Ann. Telecommun.
DOI 10.1007/s12243-008-0073-y

FPath and FScript: Language support for navigation and reliable reconfiguration of Fractal architectures

Pierre-Charles David · Thomas Ledoux · Marc Léger · Thierry Coupaye

Received: 30 July 2007 / Accepted: 20 July 2008

Abstract Component-based systems must support dynamic reconfigurations to adapt to their execution context, but not at the cost of reliability. Fractal provides intrinsic support for dynamic reconfiguration, but its definition in terms of low-level APIs makes it complex to write reconfigurations and to ensure their reliability. This article presents a language-based approach to solve these issues: direct and focused language support for architecture navigation and reconfiguration make it easier both to write the reconfigurations and to ensure their reliability. Concretely, this article presents two languages: (1) FPath, a domain-specific language that provides a concise yet powerful notation to navigate inside and query Fractal architectures, and (2) FScript, a scripting language that embeds FPath and supports the definition of complex reconfigurations. FScript ensures the reliability of these reconfigurations thanks to sophisticated run-time control, which provides transactional semantics (ACID properties) to the reconfigurations.

This research is supported by the French RNTL project Selfware (http://sardes.inrialpes.fr/selfware) and the IST project Selfman (http://www.ist-selfman.org).

P.-C. David · T. Ledoux (✉) · M. Léger
OBASCO Group, EMN/INRIA, LINA,
Ecole des Mines de Nantes, 4 rue Alfred Kastler,
44307 Nantes, France
e-mail: thomas.ledoux@emn.fr

M. Léger · T. Coupaye
France Telecom R&D, 28 chemin du Vieux Chêne,
38243 Meylan, France

Keywords FPath · FScript · Language support

1 Introduction

Component-based software engineering is a powerful approach to build complex software [35]. Its explicit representation of dependencies between components makes it easier to reuse building blocks, to assess the validity of an architecture, and simply to understand complex systems.

In practice, a system's architecture is rarely static: the rapidly changing contexts in which applications are used today mean that systems must be adapted, or even adapt themselves [21], to changes during their lifetime. For component-based systems, this means they must support dynamic reconfigurations, including unanticipated ones [32]. This must not come at the price of the system's reliability, however: in a world of interconnected services, users (human or not) rely on services to be always available. As much as possible, reconfigurations need to happen at run-time and in a way that does not jeopardize the system's dependability.

The Fractal component model [7] provides a good base to build such adaptable and autonomic systems. Indeed, one of the most interesting and powerful features of the Fractal component model is its thorough support for reflection: it is possible both to *discover* the structure of a Fractal application (introspection) and to *modify* it (intercession) at run-time. However, Fractal is defined in terms of a relatively low-level API [8]. This has two main consequences in our context:

1. This makes it relatively *difficult to write dynamic reconfigurations*. Because Fractal does not extend

Published online: 09 December 2008

the underlying language (contrary to ArchJava [2] for example), the resulting code mixes different levels of abstractions and concepts (for example, language-level interfaces and component interfaces). This is amplified by the minimalist nature of the API and its heavy use of exceptions for error signaling, which quickly lead to verbose and convoluted programs.

2. It makes it even more difficult to write *correct dynamic reconfigurations*. Because Fractal allows almost every aspect of an application to be reconfigured at run-time, even the slightest error in the reconfiguration program can make the system unusable. The Fractal model and its implementations provide some guarantees on the architectural soundness of reconfigurations (for example, all mandatory client interfaces of a component must be bound before it can be started), but this only applies to individual operations at the API level, not to reconfigurations as a whole. In addition, most concrete applications have specific architectural invariants to verify which go well beyond what the core Fractal model can provide.

Our proposal uses a *language-based approach* to overcome both these issues. Offering direct language support for manipulating Fractal architectures makes it much *easier* to discover and modify them: programs are more concise, have a higher level of abstraction by directly manipulating the relevant concepts, and do not mix these concepts with lower-level, "technical" ones. This approach also allows us to ensure the *reliability* of the reconfigurations. First, focusing the language on the single domain of Fractal architectures and simply not allowing other concerns to be expressed in the language, makes it possible to ensure *by construction* that reconfiguration programs will only be able to manipulate the architectural concepts exposed in the language. Furthermore, because with the complete control on the language's implementation, it is possible to integrate sophisticated run-time techniques to control the reconfigurations' execution, and even—thanks to the language's restricted power of expression—static ("compile-time") analyses to validate the reconfigurations before their actual execution, both in a completely transparent way from the programmer's point of view.

Concretely, this article presents two complementary languages:

- *FPath* is a Domain-Specific Language (DSL) [26] for querying Fractal architectures. Its domain is restricted to the *introspection* of architectures, navigating inside them to locate elements of interest by their properties or location in the architecture. This focused domain allows FPath to offer a very concise yet powerful and readable syntax.
- *FScript* is a scripting language that allows for the definition of complex reconfigurations of Fractal architectures, and nothing else. FScript integrates FPath seamlessly in its syntax, FPath queries being used to select the elements to reconfigure. The restricted power of the language ensures that FScript programs can only talk about Fractal architectures and cannot execute "dangerous" and difficult to control code constructs (infinite loops, IO, etc.), as would be the case in a general-purpose scripting language. To ensure the reliability of its reconfigurations, FScript considers them as *transactions*, with all the usual ACID properties (adapted to our context). The FScript interpreter integrates a back-end system that implements this transactional semantics on top of the Fractal model.

Both languages can be used either from Java through a simple API to interact with their interpreters or using an shell-like console for interactive exploration and interaction.

The article is structured in two main parts. Section 2 presents FPath and shows how it can be used by itself as a query language for Fractal architectures. Section 3 then describes the full FScript language (which embeds FPath), including the reliability guarantees it offers on dynamic reconfiguration of Fractal systems (Section 3.3). Finally, Section 4 discusses some related work and Section 5 concludes with some hints on future work.

2 Querying Fractal architectures with FPath

FPath is a domain-specific language [26] to query and introspect Fractal-based systems. FPath expressions can be used to select architectural elements in a target system according to a combination of their properties (e.g., the state of a component) and their relations to others (e.g., the subcomponents of a composite). FPath was originally designed as a part of FScript, where it is used to select the elements to reconfigure. However, it is also usable by itself as a general querying/addressing language for Fractal systems, which is why it is described in its own section.

Both the syntax and the execution model of FPath are inspired by the XPath [38] language. XPath is the standard query language for XML documents defined by the W3C, used by many other XML technologies (XLink, XSLT, XQuery...). The use of XPath as

a model for FPath was motivated by the following features:

- XPath does not depend on the syntax of XML documents but only on an abstract graph model. This makes the approach suitable for other graph-like models, in particular, component architectures.
- Although XPath defines a fixed set of node types and relations suitable for XML documents, the syntax is open-ended and supports the definition of new kinds of nodes and relations between them without changing the language. This is important in our case to support the same level of extensibility as the Fractal model itself. If a new Fractal extension is defined that introduces new architectural elements and/or new relations between components (for example, an aspect-weaving relationship [30]), it should be possible to use it in FPath without changing the syntax of the language.
- XPath expressions can have varying degrees of precision. This make it possible in FPath to write very precise expressions, which will locate an element at a specific location in an architecture (for example, "the component which implements service S for the direct child of composite C_1 named C_2"), but also more generic and less brittle expressions, which will work on a wider range of architectures (for example, "any component which provides service S and is contained, directly or not, in C_1").
- The syntax is reasonably concise and readable, and the execution model is simple to understand for users while still allowing different implementation strategies.

It should be noted that, although FPath is *inspired* by XPath, it is not *implemented* using XPath, and it does not use any XML representation of Fractal architectures: XPath implementations are too closely tied to XML, and going back and forth between Fractal components and XML documents at run-time would be too inefficient.

2.1 Language description

This section describes the FPath language itself. The first part (Section 2.1.1) describes the conceptual model used by FPath to represent Fractal architectures. Then, Section 2.1.2 shows how path expressions on this model can be used to query these architectures.

2.1.1 Modeling Fractal architectures as directed graphs

The Fractal component model is defined in relatively technical terms using a language-independent API. As a result, the different concepts it defines and their relations are not all represented in a uniform way. Some concepts have a well-defined language-level representation, like interfaces, which are reified as `Interface` objects; others are defined through conventions, like configuration attributes, which appear only as matching pairs of getter and setter methods.

To keep the FPath language simple and uniform, the language is based on an alternative representation of Fractal architectures using a very general model: *directed labeled graphs*. In this model, core entities in Fractal are represented by *nodes* and their relations by directed *arcs*, the arc's *label* indicating what kind of relationship is modeled. One important feature of this approach is that, like Fractal, it is easily extensible in an homogeneous way: modeling an extended version of Fractal with new concepts not defined in the standard specification is simply a matter of adding new kinds of nodes and new labels. The FPath language itself supports this generality and extensibility, and even its syntax does not need to change to work on extensions or alternative representations of the model.

The three main Fractal concepts which are directly reified as FPath nodes are *components*, *interfaces*, and *attributes*:

- Each component in a Fractal architecture, be it primitive or composite, is represented by exactly one component node in the corresponding graph. Although it is used in practice to identify components, the standard `Component` controller is an *interface* of the same nature as, for example, `BindingController`. Its use both for service discovery and component identification is, in a sense, purely conventional. As a concept, however, the notion of *component* is obviously at the heart of Fractal, and components have thus been made first-class entities as nodes in our graph representation.
- Each interface of each component, be it internal or external, is also represented by exactly one interface node. This includes the `Component` interfaces, which are present in the graph as interface nodes *in addition* to the corresponding component node.
- Finally, each configuration attribute of each component is represented as an attribute node. Configuration attributes are discovered by introspection on the set of methods defined by concrete `attribute-controller` interfaces. In the Fractal API, attributes are only present implicitly in the naming conventions of the methods of a component's `attribute-controller`. Because they are used a lot in practice to represent configuration parameters of interest for FPath (discovering

how a system is configured) and FScript (tuning or reconfiguring the system), individual attributes are represented as nodes in our graph model.

Although they are a central concept in Fractal, *primitive bindings* are not reified as nodes in our representation. This choice was made for several reasons: (1) bindings represent a *relation* between interfaces and are more naturally modeled as an arc in the graph; (2) representing them as nodes would add an indirection level in the graph and in every FPath query using them, for no benefit; and (3) in most cases, when one wants to talk about the bindings themselves, one can identify them with the client-side interface (the server side is easy to obtain from there in FPath).

Each kind of node defines a set of properties to further describe the architectural elements they represent. These properties are primitive values (for example, strings or booleans) and are accessible at the language-level through unary functions on nodes named after the property. For example, each node has a *name* property, available using the `name()` function (for component nodes, the name is defined by the component's `name-controller`). Other properties include the `state()` of components (`STARTED` or `STOPPED`), the `value()` of the attributes, and all the properties of Fractal interfaces (whether they are `client()`, or `server()`, `mandatory()` or `optional()`, etc.). The examples below will illustrate the use of many of these.

To complete the graph representation of Fractal architectures, the nodes described above are connected using directed and labelled arcs, which model the structure of the architecture. The arcs are used both to connect together the different nodes which represent a given component (component nodes with their interfaces and attributes) and to model the whole architecture as relations between components and interfaces.

Figure 1 summarizes all the primitives axes defined in FPath between the different kinds of nodes. For example, it shows that component nodes have outgoing arcs labelled `attribute` to the nodes that represent their attributes ("attribute ownership" on the figure). Conversely, the attribute nodes themselves are connected to their unique owner component with an arc labelled `component`. The other primitive axes include:

- `child` (respectively, `parent`), which connect component to their direct subcomponents (respectively, direct super-components).
- `binding`, which connects client interface nodes to the server interface they are bound to.
- And, finally `interface` (resp. `internal-interface`), which connects components to their external (respectively, internal) interfaces. A reverse arc labelled `component` also connects the interfaces to their owner component.

These axes are called *primitive* because they represent all the core relationships between the elements of the model. FPath also defines *derived axes* (defined in terms of the primitive axes), which are accessible in the exact same way to the user, but enable more powerful queries:

- `sibling`, which connects component nodes to all the other component nodes that share at least one

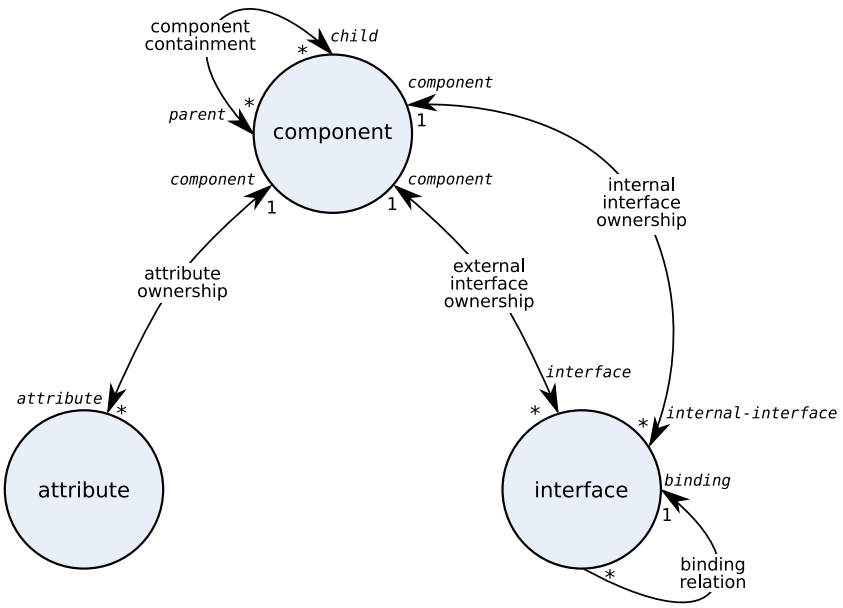

Fig. 1 Default nodes and primitive axes defined in FPath

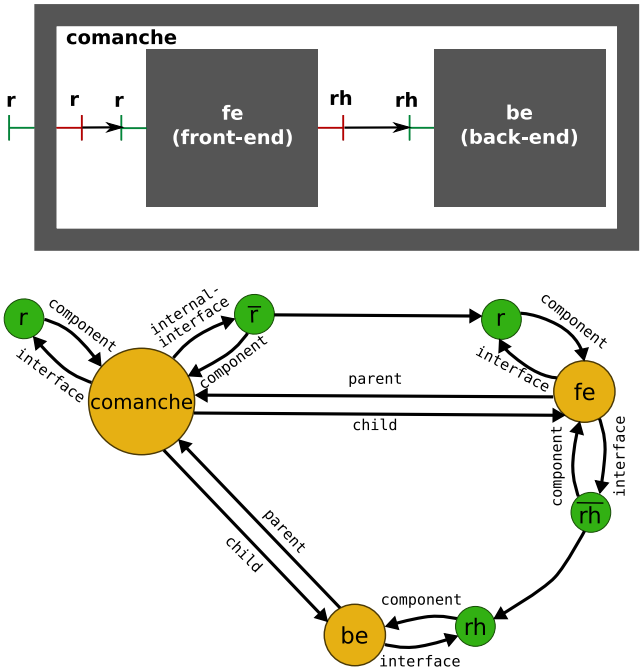

Fig. 2 The architecture of Comanche and the equivalent FPath graph

parent with them (this is useful to locate components with which one can create a connection).

- `descendant` (respectively, `ancestor`) connects a component node to *all* of its subcomponents (respectively, super-components), direct or indirect. Formally, the relation defined by this axis between the graph nodes is the transitive closure of the one defined by `child` (respectively, `parent`).
- Finally, all these derived axes have a variant named with the suffix `-or-self`, which defines the same connections as its base axis, but also connects each node to itself. This corresponds to the *reflective* versions of the relations defined by the other axes. For example, given an initial component node representing the root of an application, the `descendant-or-self` can be used to select in one step *all* the components comprising the application (including the root itself).

To conclude this section, Fig. 2 shows part of the architecture of the Comanche web server[1] (used later in the examples) and its representation as a graph. In order to keep the graph readable, only the first two levels of Comanche components are shown. The figure also omits interface nodes corresponding to control interfaces (showing only service interfaces), the `component` axis (which exist between every node and its owner component), and all derived axes.

2.1.2 FPath expressions

Given the graph representation described above, FPath expressions can be used to "walk" in the graph, starting from an initial node (or set of nodes) and successively following specific axes inside the graph to attain other nodes. The general syntax of such an expression is a sequence of *steps* separated by slashes:

```
step1/step2/.../stepN
```

Each step indicates the *axis* to follow, the *name* of the nodes to look for, and an optional list of predicates to select nodes more precisely:

```
axis::name[predicate(.)]
```

An example of a simple path expression would be:

```
$comanche/child::fe
```

The first part of the expression, `$comanche`, refers to the value of an FPath variable and is used as the initial node-set for the query. In this case it is assumed to denote the component node representing the root of the Comanche web server. The second part, after the slash character, is a *step* expression: `child::fe`. It tells FPath which axis to follow in the graph (here, `child`) and which of the resulting nodes to selects (here, the nodes named `fe`). This simple, one-step path expression thus selects the subcomponent named `"fe"` of Comanche (the server front-end).

One can also write `$comanche/child::*` to tell FPath to return *all* the subcomponents, whatever their name is. In this case, the value of the expression would be a node-set containing both Comanche's front-end and back-end (`fe` and `be`).

Of course, a path expression can be made of several steps, and each step can use a different axis. If one wants to find all the components that have an (external) interface named `r` (the `Runnable` interfaces in Comanche), one can write:

```
$comanche/descendant-or-self::
*/interface::r/component::*
```

The first step uses the `descendant-or-self` axis, which is the transitive and reflective closure of `child`. Because the expression uses a star (`*`), this will select *all* the components in Comanche, including the root (the `-or-self` variants of derived axes always include the initial node in the result). This node-set is used as input for the second step, `interface::r`. From each of the initial nodes (here, all the components in Comanche), this will try to follow the `interface` axis to find an

[1] See http://fractal.objectweb.org/tutorial/index.html.

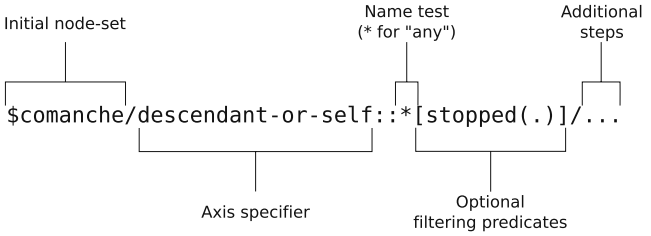

Fig. 3 Path expressions syntax

interface node named `r`. For some of the component, there is no such interface. This simply means they do not contribute anything to this step. For the others, the nodes representing the matching interfaces will be grouped in a single node-set. At this point, the intermediate result for the first two steps is the set of all the external interfaces named `r` present in Comanche, but what we want is the set of *components* that have such an interface. This is easily done in the third step using the `component` axis, to find all the owners of the interfaces named `r` (the name used for `Runnable` interfaces in Comanche).

Path expressions have one last feature that makes them very powerful: at each step in a path, it is possible to filter the intermediate result before it is fed to the next. The filtering is done using an optional list of predicates, which can access node properties and can even be complete path expressions themselves.

For example, the query

```
$comanche/descendant-or-self::*[stopped(.)]
```

finds all the components in Comanche that are stopped. The first part of the query is the same as before, but with a predicate expression in brackets added at the end of the step. Here, the predicate simply invokes the `stopped()` function to each node in the intermediate result (the dot "`.`" denotes the implicit argument passed to the predicate). Instead of returning all the components in Comanche, this will return only those that are stopped. Figure 3 summarizes the syntax of path expressions.

2.2 Examples of FPath queries

This section describes more complex FPath expressions that show the power of the language as a general query language for Fractal-based systems.

Finding configuration attributes Configuration attributes exposed by `attribute-controller` interfaces provide a convenient way to customize the behavior of components. Complex architectures with dozens of components can expose lots of these attributes, scattered in the many composition levels of the system, and it is not always obvious what configuration parameters are available to tune an application. However, with a very simple FPath query, it is possible to discover *all* the configuration attributes supported by a system:

```
$root/descendant-or-self::*/attribute::*
```

This query, where `$root` denotes the top-level component of the system (or a subsystem), will return a set of attribute nodes, each representing one of the configuration attributes of any component inside `$root` (directly and indirectly).

Checking architectural invariants Fractal is a very flexible model and makes it possible to reconfigure applications dynamically. However, it also imposes some constraints to ensure that the resulting architectures are consistent. For example, before a component can be started, all its client interfaces that are not optional must be bound to a compatible server interface. Such a constraint can be checked easily using the following FPath query, which will find all the components violating the constraint (and can, thus, not be started immediately):

```
$root/descendant-or-self::*/interface::
*[client(.)][mandatory(.)]
[not(./binding::*)]
```

This query illustrates the use of multiple predicates. It also show how it is possible to access the properties of nodes using predefined functions. Here, the expression uses the functions `client()` and `mandatory()` to select only client interfaces that *must* be bound (i.e., are not optional). The last predicate uses an embedded path expression to select the server interface bound to the candidate client. The predicate will be *true* if and only if the client interface is not currently bound (`./binding::*` returns an empty set, considered *false*, which is transformed in *true* by the `not()` function).

Shared components Fractal supports component sharing, i.e., components that have multiple direct super-components. This is a powerful feature, but it can make an architecture more difficult to understand, as each of the parents try to control the shared component, requiring some coordination. Once again, FPath makes it very easy to identify all the shared components in a system, which will need to be checked further:

```
$root/descendant-or-self::
*[size(./parent::*) > 1]
```

This query can almost be read as is: "Starting from the system's root component, select all its descendants in

the architecture, but keep only those which have strictly more than one parent." For comparison, the Java code equivalent to this simple and readable expression takes 25 lines of code.

3 Programming reliable architectural reconfigurations with FScript

This section describes the FScript language, which embeds the FPath query language presented above. While FPath only supports querying Fractal architectures (i.e., pure introspection without modification), FScript enables the definition of complex reconfiguration scripts that can modify the structure and configuration of a Fractal application. In FScript, FPath is used as an embedded sub-language to select the elements on which to apply the reconfigurations.

The overall design objective of FScript is to be a scripting language to express reliable architectural reconfigurations of Fractal-based systems. More precisely:

- FScript is *focused* on the manipulation of architecture-level concepts. It provides complete control of the architecture and configuration of Fractal-based systems, to the full extent of what is defined by the model, but nothing else. This means all the concepts defined by the Fractal model are available in the language, both for introspection and, where supported by the specification, modification. Because its domain is limited to the architecture level ("wiring" and configuration), it also means that FScript is explicitly not designed to *implement* Fractal components. It is not a complete programming language and does not deal with business logic. For example, although the service interfaces of components are visible and their bindings controllable, the language itself does not support the invocation of business methods.
- The second important objective is to ensure the *reliability* of the reconfigurations. When a system is dynamically reconfigured, it is critical to ensure that reconfigurations will not result in inconsistent systems. In FScript, this is guaranteed by giving a *transactional semantics* to the reconfigurations, with the standard ACID properties. Implementing our own language instead of reusing an existing one makes it possible to integrate the run-time controls required to offer these guarantees in a completely transparent way for the FScript programmer. The restricted power of expression of the language compared to general-purpose languages also allows in the future the use of static analyses to validate reconfigurations before they are actually executed.
- Finally, a secondary but important objective for FScript is to be able to support any extension to the Fractal model seamlessly, i.e., without requiring changes to the core language. This constraint is already visible in the design of the FPath model and syntax, which supports the definition of new kinds of nodes and axes.

The rest of this section first describes the different constructs of the language (Section 3.1). More involved reconfigurations are then described in Section 3.2. Finally, the section concludes with a description of the reliability guarantees offered by FScript, and an overview of the techniques used to implement them (Section 3.3).

3.1 Language description

This section describes the different constructs of the language. The language's structure is that of an imperative, scripting-like language with a syntax mostly taken from the C family of languages, except for FPath expressions. This syntax was chosen for its simplicity and familiarity, making it easier for most users to learn the language.

The next subsections describe how FScript programs are structured (Section 3.1.1) then show the different control structures available to program the reconfigurations (Section 3.1.2), and finally, which primitive (predefined) reconfiguration actions are available (Section 3.1.3).

3.1.1 Procedures definitions

An FScript program is made of a sequence of top-level procedure definitions, which can be either *functions* or *actions*. Moreover, FScript has a dynamic type checking system where the types of procedure arguments and return values are checked at invocation time relating to the procedure signatures. Functions can only use introspection features from FPath and are hence guaranteed to be side-effect free. They can be used in FPath expression (for example, in predicates). Actions, on the other hand, are allowed to modify the target architecture. Concretely, the body of a function can use only other functions (be they predefined FPath functions or user-supplied), while actions can make use of functions and other actions (primitives or user-defined).

The only syntactic difference between functions and actions appears in their definitions. Functions are

introduced by the keyword `function`, while actions use the keyword `action`:

```
/* Locate the request scheduler in a
Comanche instance. */
function scheduler(comanche) {
  return $comanche/child::fe/child::s;
}

/* Replace the request scheduler of a
Comanche instance. */
action
replace-scheduler(comanche, newSched)
{
  //Invoke user-defined function.
  prev = scheduler($comanche);
  //  replace() action defined below.
  replace($prev, $newSched);
  return $prev;
}
```

The first procedure is a function named `scheduler()`; it takes a single argument named `comanche`, which should represent the top-level component of a Comanche instance. The function simply evaluates and returns the value of an FPath expression. As FPath can only introspect components but cannot modify them, the whole procedure can be declared as a safe `function`. Once loaded in an FScript interpreter, it will then be usable in other procedures (both functions and actions) or in FPath queries.

The second procedure, `replace-scheduler()`, is declared as an action, and can thus use any other procedure, including functions (as in the first line) and other actions (as in the second line). Here, the action first locates the current scheduler component of the Comanche instance passed in parameter. It then invokes a generic reconfiguration action named `replace()` to do the actual replacement.

An important property of reliable reconfiguration is that they should always terminate in finite time. One of the ways FScript guarantees this is by forbidding recursive definitions of user-defined procedures (either direct or indirect).

3.1.2 Control structures

FScript procedures support a limited set of control structures so that it is possible to ensure that they will eventually terminate. In addition to simple sequencing of instructions (terminated by semicolons ";"), the control structures are:

Variables assignment New local variables can be created inside the body simply by assigning them an initial value. Their values can be changed by reassigning them. For example, the function `bound-to()` defined below introduces a new local variable named `$servers` to make the code easier to follow. The second function, `bindings-to()`, also introduces a local variable `$itfs` but then changes its value before returning it.

```
/* Tests whether the client interface $itf
 * is bound to (an interface of)
 * component $comp.
 */
function bound-to(itf, comp) {
 servers = $itf/binding::*/component::*;
 return size(intersection($servers,
 $comp/component::*)) > 0;
}

/* Finds all the client interfaces which are
 * bound to the component $comp.
 */
function bindings-to(comp) {
 itfs = $comp/parent::*/internal-interface::
 *[bound-to(., $comp)];
 itfs = union($itfs, $comp/sibling::*/
 interface::*[bound-to(., $comp)]);
 return $itfs;
}
```

Conditionals Conditional execution is supported using the standard C syntax. For example, the following action uses the life-cycle state of a component, accessible through the `started()` *function* to make a choice:

```
/* Makes sure that $dest is in the same
 * lifecycle state as $src.
 * Assumes the standard lifecycle
 * (STARTED, STOPPED).
 */
action copy-lc-state(src, dest) {
  if (started($src)) {
    start($dest);
  } else {
    stop($dest);
  }
}
```

Iteration FScript supports a restricted form of iteration, which ensures that the execution will always terminate. The `for` loop executes a given block repeatedly with a local iteration variable successively bound to each element in a node-set (which is always finite):

```
/* Copies the value of all the attributes
 * of component $src to the attributes
 * of the same name in component $dest.
 */
action copy-attributes(src, dest) {
  for oldAttr : $src/attribute::* {
    newAttr = $dest/attribute::
    *[name(.) == name($oldAttr)];
    set-value($newAttr, value($oldAttr));
  }
}
```

The iteration expression (here `$src/attribute::*`) must evaluate to a node-set.

Explicit return At any point during its execution, a procedure can stop its execution and immediately return to the caller (optionally yielding a value) using a `return` statement.

3.1.3 Primitive actions

All the standard reconfiguration operations defined in the Fractal specification are available in FScript as primitive, predefined actions:

Content The operations defined in the standard `content-controller` interface are available in FScript as actions `add()` and `remove()`. They correspond, respectively, to the `addFcSubComponent()` and `removeFcSubComponent()` methods.

Because FScript is imperative and not object-oriented, each of these actions take two arguments instead of one. For example, the FScript code `add($parent, $child)` adds the component denoted by the `$child` (a component node) into the composite denoted by `$parent` (also a component node). It corresponds to the following Java code

```
Fractal.getContentController(parent).
addFcSubComponent(child);
```

where `parent` and `child` are objects of type `Component`. The `remove()` action is similar, except that `remove($parent, $child)` invokes the `removeSubComponent()` control method.

Bindings Bindings between interfaces can be controlled in FScript using the `bind()` and `unbind()` actions. They correspond, respectively, to the `bindFc()` and `unbindFc()` methods of the `binding-controller`.

`bind()` takes as arguments the client and server interfaces to connect, in this order. This is slightly different than in the Java API, where the client interface is referenced by name. In Java, the receiver object of the `bindFc()` method and the interface name are used to identify the client interface, but FScript directly uses the interface node. `unbind()` takes only the client interface to disconnect as an argument.

Life-cycle The default component life-cycle defined by the default `lifecycle-controller` can be controlled using the actions `start()` and `stop()`. Each takes a single argument denoting the component to start or stop.

Some reconfiguration operations in Fractal require that the components implied are stopped. When such an action is called in an FScript reconfiguration, the FScript interpreter makes sure they are stopped before executing the action, without needing an explicit call to `stop()`. They are restarted automatically at the end of the reconfiguration. This feature simplifies a lot the task of the programmer, who does not have to test the components' state at each step and keep track of which must be restarted.

Attributes The values of configuration attributes defined in `attribute-controller` can also be changed in FScript, using the `set-value()` primitive action. It takes two arguments: the attribute itself, as an FPath attribute node, and the new value to set.

Name The name associated to Fractal components through their `name-controller` can be modified using the `set-name()` primitive action: `set-name ($component, "newName")`.

Component creation Finally, FScript also supports the creation of new components through the action called `new()`. It corresponds to the `newComponent()` method of Fractal Architecture Description Language's (ADL's) `Factory` interface. In its simplest form, it takes a single-string argument, which must be the full name of a Fractal ADL component definition. Some component definitions in Fractal ADL are parametrized, and the parameter values must be supplied to instantiate them. This is also supported by a second form of `new()`, where parameter names and values are supplied as additional parameters. Because FScript does not support complex data structures like arrays of dictionaries, the pairs of parameter names and values are simply put in sequence.

```
// Simple form
comanche = new("comanche.Comanche");
// Second form with parameters.
adl = new("org.objectweb.fractal.adl.
          BasicFactory",
          "fractaladl.backend",
          "org.objectweb.fractal.adl.
          JavaBackend");
```

In addition to these predefined actions, new primitive actions can be easily added to reflect new customized Fractal operations and controllers. For instance, to deal with Fractal RMI bindings, two new primitive actions have been developed to bind and unbind component references in a Fractal RMI registry.

3.2 Sample reconfiguration scripts

This section describes in more detail two complex FScript programs, which illustrate all the constructs of

the language and how they can be combined to create complete reconfiguration scripts.

3.2.1 Adding a cache to Comanche

The Comanche web server already presented above is designed to be very simple. In particular, it does not include a page cache and needs to reread files' content on each request. Because of its component-based architecture, however, it is easy to create a new cache component and to introduce it at the right place in the architecture. Thanks to Fractal's dynamicity, it is even possible to do so while the server is running.

The following FScript program does just this. It is split into two actions:

- `get-or-create-cache()` takes in argument the Comanche request handler (a subcomponent of the server's backend). This is the composite that includes the different handler components (for example, the file and error handler) and a dispatcher, which sends the requests to all the available handlers in turn until one accepts it. The action first looks inside the handler composite to see if an instance of the cache component (named `"cfh"`) is already there. If it is found, it is returned immediately. Otherwise, a new cache component in instantiated using the `new()` action, renamed `"cfh"`, and placed inside the handler.
- The `enable-cache()` action itself first tests whether the cache is already installed by following a binding on the dispatcher component. If the cache is not installed, it removes the direct connection from the dispatcher to the file handler, uses the `get-or-create-cache()` to obtain a cache, and inserts it between the dispatcher and the file handler. Once this is done, the next requests sent by the dispatcher will be intercepted by the cache, which can either answer them directly if it has the file content in memory or use its connection to the original file handler component to read (and store) the content otherwise.

```
action get-or-create-cache(handler) {
  if ($handler/child::cfh) {
    return $handler/child::cfh;
  } else {
    cache = new("comanche.
    CachingFileRequestHandler");
    set-name($cache, "cfh");
    add($handler, $cache);
    return $cache;
  }
}
```

```
action enable-cache(handler) {
  dispatcher = $handler/child::rd;
  if (not($dispatcher/interface::
  h0/binding::*/component::cfh)) {
   unbind($dispatcher/interface::h0);
   file-handler = $handler/child::frh;
   cache = get-or-create-cache($handler);
   bind($dispatcher/interface::h0,
   $cache/interface::request-handler);
   bind($cache/interface::handler,
   $file-handler/interface::rh);
   start($cache);
 }
}
```

3.2.2 Replacing a component (hot-swapping)

The last example action is a more complex—and useful—one, which reuses some of the procedures defined earlier. It takes two components as arguments and replaces the first one with the second everywhere it appears in the architecture. It also reproduces the configuration (attributes) and state (started or stopped) of the original into the replacement component.

The structure of the action is easy to follow:

1. First, it adds the new component inside all the parents of the old one. This step must be done before the next ones to avoid the creation of nonlocal bindings.
2. Next, it updates all the bindings that involved the old component with an equivalent binding with the new one. Updating the bindings *from* the old component (step 2.1 in the code) is trivial, using a simple FPath query to loop on the client interfaces that are bound. The action simply has to remember to take a reference on the original server interface (the `server` variable in the code) before destroying the original binding. Updating the bindings that pointed *to* the original component (step 2.2) is a little more involved since it is not possible to directly follow bindings "backwards." Finding all the relevant client interfaces is done in the `bindings-to()` function, which was explained earlier. Thanks to the power of FPath, this was done in three lines. Once this task is abstracted away in a function, using the resulting set of client interfaces in `replace()` is actually even simpler than updating the bindings from the old component.
3. Then, the actions reproduce the configuration and state of the old component into the new one using previously defined actions `copy-attributes()` and `copy-lc-state()`.

4. Finally, the action can completely remove the old component from the architecture since it is not connected to it anymore.

```
/* Replaces $oldComp with $newComp
* everywhere in the architecture */
action replace(oldComp, newComp) {
  // 1. Add the new component everywhere
  // the old one is present.
  for p : $oldComp/parent::* {
    add($p, $newComp);
  }
  // 2. Update bindings involving oldComp
  // to use newComp instead
  // 2.1. Bindings *from* oldComp
  for client : $oldComp/interface::
  *[client(.)][bound(.)] {
    itfName = name($client);
    server = $client/binding::*;
    unbind($client);
    bind($newComp/interface::$itfName,
    $server);
  }
  // 2.2. Bindings *to* oldComp
  // (uses bindings-to() defined above)
  for client : bindings-to($oldComp) {
    itfName = name($client/binding::*);
    unbind($client);
    bind($client, $newComp/interface::
    $itfName);
  }
  // 3. Reproduce configuration and state
  copy-attributes($oldComp, $newComp);
  copy-lc-state($oldComp, $newComp);
  // 4. Remove the old component
  for p : $oldComp/parent::* {
    remove($p, $oldComp);
  }
}
```

Comparison with Java FScript, as a domain-specific language, has not only a more readable syntax but it is also more compact than a general-purpose language like Java for programming reconfigurations in Fractal. The complete definition of the `replace()` action above, including the support procedures defined earlier (`bound-to()`, `bindings-to()`, `copy-lc-state()`, and `copy-attributes()`) and comments takes 68 lines. The equivalent Java code takes 114 lines, an increase of about 67%. The difference may not seem that large, except for the fact that the Java version uses a helper method to deal with attributes in a generic way,[2] and most importantly, the total of 114 lines does not include *any* error handling: exceptions are not simply caught and ignored, they are not considered at all.

Table 1 Code size comparison between FScript and Java

Procedure	LOC in FScript	LOC in Java	Increase
`bound-to`	4	7	×1.4
`bindings-to`	5	26	×5.2
`copy-lc-state`	7	10	×1.42
`copy-attributes`	6	7	×1.16
`replace`	21	37	×1.76
Total	43	86	×2

Not surprisingly, the biggest gain in terms of size is found in the `bindings-to()` equivalent, which uses relatively sophisticated FPath expressions. The FScript version is five lines long and could easily be made shorter, even to the point of a simple one-liner. The Java version, however, takes 26 lines (an increase of 420%, again with no error handling) and includes up to five levels of imbrication, three of which are `for` loops, making the code very difficult to understand. Table 1 gives the detail, procedure-by-procedure. The number of lines in the table does not include comment lines.

There is no explicit error handling construct in FScript, but this is not because errors are ignored. On the contrary, it is because error detection and correction is handled completely automatically and transparently to ensure the reliability of the reconfigurations and of the target applications. The next section shows how this is done using a powerful and uniform approach that handles several kinds of errors in a completely transparent way for the FScript programmer.

3.3 Ensuring the reliability of reconfigurations

Reliability (a key concept of dependability [5]) is a main problem in systems subject to dynamic reconfigurations, especially with open models like Fractal because reconfigurations can be unanticipated at compile time and, consequently, static analysis of reconfigurations is difficult. Indeed, dynamic reconfigurations may be invalid and leave a system in an inconsistent state, i.e., no more available/usable from a functional point of view. FScript aims to guarantee the reliability of reconfigurations with automatic and transparent error detection and correction thanks to transactional semantics for reconfigurations.

3.3.1 FScript reconfigurations as transactions

A reconfiguration script must be a consistent transformation of the system; however, it is possible to specify

[2] The `AttributesHelper` class (provided by FScript) is about 450 lines long, although only part of the features are used here.

invalid actions in FScript. For instance, the following action is invalid:

```
action strictly-invalid(c1, c2) {
  add($c1, $c2);
  add($c2, $c1);
}
```

Whichever application it is applied to, it will fail because this would create a cycle (*$c1* would end up containing itself), which is forbidden by the Fractal model. Some actions can be valid only on some specific system architectures, i.e., with some preconditions on components of the system:

```
action conditionally-valid(c1, c2) {
  add($c1, $c2);
  bind($c1/internal-interface::foo,
  $c2/interface::bar);
}
```

To be valid, this action requires that:

- Component *$c1* is a composite component
- *$c1* and *$c2* have, respectively, *foo* and *bar* interfaces
- *foo* and *bar* interfaces are type compatible

Executing these invalid actions in a system would violate its consistency, so we use transactions to be able to recover the system from failed reconfigurations. The execution backend of the FScript interpreter aims to ensure the reliability of dynamic, distributed, and concurrent reconfigurations in the Fractal component model. This backend consists essentially on an extension of the Julia [7] implementation of the Fractal model in Java associated to a transaction manager dedicated to dynamic reconfigurations. We use a flat transaction model for managing reconfiguration transactions. Several more complex transaction models have been defined for specific applicative domains [37] but flat transactions have proven to be sufficient and efficient in applications such as databases where transactions are relatively short-lived and the number of concurrent transactions is relatively small and the transactional processing system is centralized. Dynamic reconfigurations we consider appear to satisfy these hypotheses. Indeed, reconfigurations are essentially short-lived FScript actions, and the level of concurrency for reconfigurations should be moderate: the system is not constantly reconfigured; otherwise, it would reduce its availability. Moreover, even if reconfigurations can be distributed, our transaction processing system is rather centralized. This model appears sufficient for short-lived reconfiguration transactions and the moderate level of concurrency we consider here. FScript reconfigurations are automatically demarcated by the FScript frontend, then each top-level FScript action and function is executed as a separated transaction (subactions/functions are inlined as explain in the sequel).

ACID properties in the context of reconfigurations can improve reliability by making systems fault-tolerant, i.e., compliant with the specification in spite of faults due to dynamic reconfigurations. These properties are unifying concepts of transactions for distributed computation [36] used for supporting concurrency and recovery and guaranteeing system consistency. The definition of the ACID properties in the context of dynamic reconfigurations in component-based systems is as follows:

- **Atomicity** Either the system is reconfigured and the reconfiguration transaction commits (all the operations forming the transaction are executed) or it is not and the transaction aborts. If a reconfiguration transaction fails, the system comes back in a previous consistent state as if the transaction never started.
- **Consistency** A reconfiguration transaction is a valid transformation of the system state, i.e., it takes the considered system from a consistent state to another consistent state. A system is a consistent state if and only if it conforms to our consistency criteria: it does not violate the component model and possibly more specific constraints.
- **Isolation** Reconfiguration transactions are executed as if they were independent. Results of reconfiguration operations inside a noncommitted reconfiguration transaction are not visible from other transactions until the transaction commits or never if the transaction aborts. It must be noticed that synchronization between the functional and the nonfunctional levels to reach a "quiescent state" (i.e., a stable state) [22] currently relies on the implementation of the *LifeCycleController* as in Julia [7].
- **Durability** The results of a committed reconfiguration transaction are permanent: once a reconfiguration transaction commits, the new state of the system (both the architecture description and the component state) is persisted so that it can be recovered in case of major failures (e.g., hardware failures).

Each ACID property and associated mechanisms will be detailed more precisely in the next sections.

3.3.2 Atomicity of reconfiguration transactions

The execution of a reconfiguration transaction, which corresponds to a top-level FScript action or function, boils down to the execution of primitive reconfiguration operations (in Fractal controllers), i.e., *introspection operations* without side effects and *intercession operations* that modify the system. In our approach, to ensure atomicity of reconfiguration transactions, operations performed in transactions that abort are undone. Actually, only intercession operations need to be taken into account, as they are the only operations modifying the system; functions are considered as read-only actions, which must be isolated. When operations are reversible, undoing a reconfiguration transaction is equivalent to sequentially undoing primitive intercession operations (in reverse order) in a compensation transaction. The undo model is linear and there is no permutation between operations, and then the commutativity of operations is not required. On the other hand, it is possible to define nonreversible intercession operations, but they need a customized treatment during the transaction rollback. Compensation operations can be associated to these operations but without any guarantee on the atomicity since the resulting state of a rollback can be not exactly the same as the initial one, but the system consistency is ensured.

An extensible library of primitive operations from the Fractal API is proposed where each intercession operation is associated to its inverse intercession operation (cf. Fig. 4) in the semantics, we choose (typically, *bindFc* and *unbindFc* for two component interfaces are inverse operations). Moreover, an operation can keep a state so as to be undone (e.g., changing the value of an attribute requires to keep the old value of this attribute). When a transaction rollback occurs, a notification is sent by means of a *RollbackException* caught by the reconfiguration client.

The target component architecture is managed as a single transactional resource with only one centralized transaction manager dedicated to dynamic reconfigurations. Then, the transaction manager can use internally a One-Phase-Commit protocol [1] without a voting phase since there is no need for synchronization between several transactional resources and the manager has only to coordinate its own modifications, even if the system architecture is distributed. Nevertheless, the transaction manager also supports the two-phase-commit protocol [36] so as to be able to participate in global transactions involving several transactional resources; then it can be coordinated as a distributed transaction participant by another transaction manager.

By default, as any operation could potentially lead the system to an inconsistent state, they must always be included in transactions (if there is no transaction, a new one is created). Transaction demarcation is automatic for FScript actions and functions (no explicit language demarcation is needed); they are always included inside a transaction. As there is no nested transaction, two nested actions or functions are always executed in the same transaction. A top-level action is always composed of a single transaction and nested actions are then linearized. It corresponds to the FScript semantics in which actions encapsulated in other actions are simply inlined in the top-level actions.

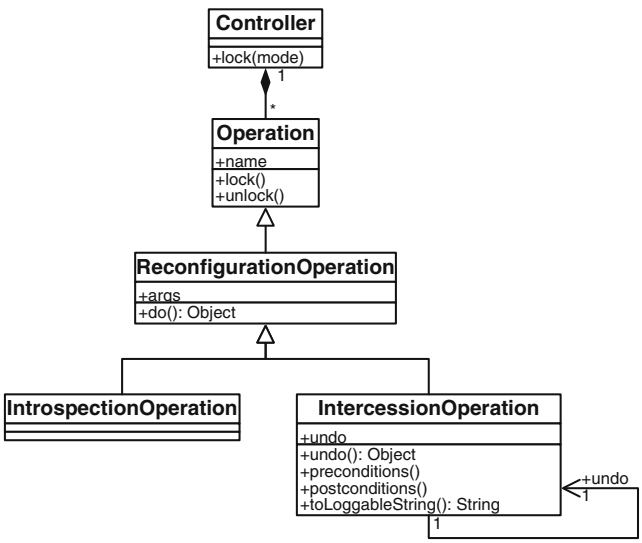

Fig. 4 Reconfiguration operation model

3.3.3 Integrity constraints to ensure system consistency

In our proposal, system consistency relies on integrity constraints, and we want to express these constraints both on the component model and on applications. An *integrity constraint* is a predicate on assemblies of architectural elements and component state [23]. Therefore, a reconfiguration transaction can be committed only if the resulting system is consistent, i.e., if all integrity constraints on the system are satisfied. To express constraints including invariants, preconditions, and postconditions, we use FPath as a constraint language "à la OCL" [28]. Some advantages of FPath are that:

- It can navigate and select Fractal elements at runtime with just introspection capacities.
- It is based on a simple graph representation of the system during execution, which can be easily

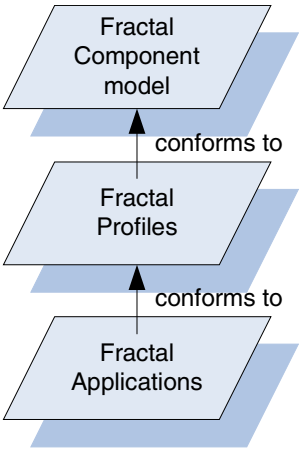

Fig. 5 The three levels of integrity constraint

extended with new concerns (e.g., new relations can be added between Fractal elements).
- Its powerful syntax allows to specify both generic constraints for all elements in the system by using the star token and recursivity and specific constraints where Fractal elements are designated by their name.

We distinguish three different levels of constraint specification (cf. Fig. 5) corresponding to three different abstraction levels.

The model level This a set of generic constraints associated to the component model. These constraints apply to all instances of some elements of the Fractal model. Examples of such constraints are hierarchical integrity (bindings between components must respect the component hierarchy) or cycle-free structure (a component cannot contain itself to avoid recursion). We specified the semantics of reconfiguration operations in the model with preconditions and postconditions that should never be violated. An example of such a precondition for the remove operation is to check that all interfaces of the component to remove are unbound:

```
Fractal API: void
removeSubComponent(Component child);
// preconditions:
not(\$child/interface::*[bound(.)]);
```

The profile level This is a set of generic constraints to refine the component model for a family of applications. The profile level conforms to the model level. A profile may, for instance, forbid component sharing in applications with the constraint `size(./parent::*)<=1`, which says that every component cannot have more that one super-component.

The application level Application constraints are specific to a given architecture and apply directly to instances of Fractal elements designed by their names. The application level conforms to the profile and to the model. Invariants can concern cardinality of subcomponents in a super-component, two-component interfaces that can never be unbound, etc. Application constraints are specified in Fractal ADL with the *constraint* tag. For instance, we may want to add a structural invariant on a simple component *ClientServer* (Fig. 6), saying that it must always contain only one component providing the *service* interface, it will be specified in the ADL definition of the component as follows:

```
<definition name="ClientServer">
 <interface name="main" role="server"
  signature="java.lang.Runnable" />
 <component name="client">
  <interface name="main" role="server"
   signature="java.lang.Runnable" />
  <interface name="service"
   role="client" signature="Service" />
  <content class="ClientImpl" />
 </component>
 <component name="server">
  <interface name="service"
   role="server" signature="Service" />
  <content class="ServerImpl" />
 </component>
 <binding client="this.main"
  server="client.main" />
 <binding client="client.service"
  server="server.service" />
 <constraint value="size(./child::
  *[./interface::service])=1" />
</definition>
```

Constraints must be checked both at compile time on the component static configuration and at runtime, and only new application constraints can be dynamically added or removed by means of a *ConstraintController*. There is currently no check of conflicts between

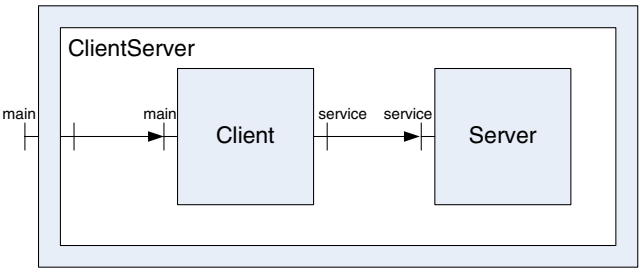

Fig. 6 A simple client–server Fractal application

constraints. Pre/postconditions of primitive intercession operations are checked at each operation execution, whereas invariants are only checked at commit of transactions. That is to say a system can temporarily violate invariants during a transaction but it must be in a correct state after commit. When it is detected, a constraint violation makes the transaction involved rollback.

3.3.4 Isolation of reconfigurations to support concurrency

Several administrators (humans or machines) may want to reconfigure the same system at the same time or a single reconfiguration may be composed of parallel reconfigurations to optimize the reconfiguration process. There are two available scheduling policies adapted to the level of concurrency for reconfigurations in Fractal applications: either reconfigurations are concurrently executed and accesses to Fractal elements in the system must be synchronized or reconfigurations are serially executed, i.e., only one reconfiguration is executed at a time while others are simply queued.

For concurrency management, we adopt a pessimistic approach with strict two-phase locking (2PL) [36] to provide strong concurrency (corresponding to *serializable* ANSI isolation level). In 2PL, locks acquired during a transaction are only released when the transaction is committed to avoid *phantoms* (lost updates). Several elements in the model corresponding to FPath nodes are lockable with a hierarchical algorithm: for instance, a lock acquisition on a component for instance will also lock all its interfaces. Locks are reentrant (a transaction can acquire several times the same lock) and there are basically two types of locks: read (or shared) locks and write (or exclusive) locks. A *LockController* is provided with every component to lock its subelements; locks can be also acquired by means of an FPath expression.

We use this locking model at a finer granularity to avoid inconsistent state for reconfiguration operations based on their semantics. Basically, an introspection operation in a transaction will take read locks on other reconfiguration operations to avoid *dirty reads* and *fuzzy reads* (nonrepeatable reads): it should not see modifications in the system due to other transactions that have not been committed yet. However, two operations in two different transactions can introspect the same element in the system by sharing a read lock. On the other hand, intercession operations on a given element will take write locks to avoid conflicts between state modifications: other transactions can neither introspect the same element nor modify it. For example, if we want to add a component B in a component A, it will write lock the following operations in the *ContentController* of the parent (A): *addFcSubComponent*, *getFcSubComponents*, and *removeFcSubComponent*. The *getFcSuperComponents* operation in the *SuperController* of the child (B) will also be write locked.

3.3.5 Durability of reconfigurations to support recovery

Transactions are units of recovery, so for every transaction, demarcation and primitive intercession operations are logged in a journal so that it can be undone in case of rollback. The journal is kept both in memory and persisted on disk and it can be redone in case of software or hardware failure. We use write-ahead logging [36] with checkpointing, which means that all modifications are written to the log before they are applied in the system and both redo and undo information is stored in the log. In case of major failure such as a hardware crash, the undo/redo recovery protocol is applied by the transaction manager: all transactions that are not committed are canceled and all committed transactions since the last checkpoint are redone. The log format currently follows the FPath syntax and Fractal elements in arguments of operations are designed with one absolute path in the system architecture (path is not unique due to sharing).

In addition to the journalization of transactions, the system state is periodically checkpointed. The state of a component-based system that is considered here is its architectural description and the set of its component state. Checkpointing at commit time allows to recover any component in its last known consistent state resulting from the last successful reconfiguration. A component architectural state is checkpointed by a persistence manager with Fractal ADL dumps, the ADL dumper is extensible so as to consider eventual extensions of the ADL language. Thus, some new state information can be included in the ADL such as, for instance, the lifecycle state of components. The functional state that is considered for a component is basically the set of its attribute values of components. Attribute values that are of primitive types are automatically saved in ADL dumps. There are two implementations for other attribute values that are Java Object: either a Java serialization in a file named with an absolute path of the component or they can be saved in a database if an object relational mapping is defined for the component attributes (our implementation uses Hibernate[3]). In case of recovery, the component state

[3] http://www.hibernate.com.

will be updated automatically with its last available persisted state. An extension of the attribute module in Fractal ADL allows to define which attributes are persistent. Regarding the functional state persistence, state transfer is the responsibility of the *StateController* for each component.

3.3.6 An instrumented Fractal implementation for reconfiguration transactions

The Julia implementation of the Fractal model has been extended to provide transactional semantics to Fractal reconfiguration operations. It must be noted that this implementation does not depend on FScript so that reconfigurations that are programmed only with the Fractal API can also benefit from transactional properties by means of an explicit transaction demarcation. New controllers have been added to the standard Fractal controllers to implement the ACID properties:

- The *ConstraintController* is used to set and get integrity constraints to specify the component consistency. It is bound to a *Consistency Manager* component for constraint checking before validating a transaction.
- The *LockController* is used for concurrency management to lock the component, its interfaces, and operations. It interacts with a *Concurrency Manager* component, which implements the locking protocol and notably keeps a waiting graph of transactions.
- The *StateController* allows to get and set the functional state of the component, i.e., its architecture description and its functional state. A *Persistence Manager* component is responsible to get and serialize the architectural and functional state of components involved in a transaction.
- The *TransactionController* checks preconditions and postconditions of operations and registers the invoked operations in the journal of the *Transaction Manager*.

Moreover, an interceptor is added to each standard reconfiguration controller so as to notify the transaction manager when an operation is invoked on the control interface. It acts as a proxy for each operation of the controller and it reports any error that could occur during the execution of the operation and lead to a cancellation of the transaction. For example, a reconfiguration operation `m()` is intercepted before and after its execution while exceptions are caught and registered in the transaction manager:

```
void m () {
  try {
    checkPreconditions();
    register(m, PRE);
    impl.m();
    checkPostconditions();
    register(m, POST);
  } catch(Exception e) {
     register(m, FAILED, e);
  }
}
```

When an exception is thrown by the code of the Fractal operation or if the check of pre/postconditions fails, the reconfiguration is rollbacked and the FScript interpretor is notified of the error.

4 Related work

Navigation and query languages Object Constraint Language (OCL) [28] is a standardized language used by the OMG in UML and related technologies. It is used primarily to annotate UML models with model-specific constraints (for example, pre and postconditions on operations), but can be used for navigation and query. OCL provides a rather concise and readable syntax and has good support to handle collections of elements (filtering, set operations…). However, the language cannot navigate along transitive relations like FPath does with derived axes (`descendant` for example). This means the "depth" of a query must be fixed and hard-coded, limiting the usefulness of queries in large or unknown architectures.

Object Query Language (OQL) [9] is a declarative language derived from SQL92 and defined by the ODMG. It includes several extensions to SQL to support object databases, including complex objects with attributes and references to each other, path expressions, operation invocation, and a rich collection of data structures (sets, bags…). Contrary to OCL, it supports navigation to arbitrary depth using a built-in function `reachables()`, which finds all the objects reachable from a given source. However, this operation is not very selective and returns all the reachables at once. Compared to FPath, it would correspond to following *all* the axes at the same time and letting the user write the appropriate predicates to restrict the result afterwards. Compared to FPath, OQL is a much more powerful (and complex) language, but its very generality makes it less adapted to our specific domain (Fractal

architectures), where FPath provides tailored support to the relevant concepts.

Logic programming, in the form of Prolog or Datalog [10] or languages inspired by them like [19], can also be used as query languages. Graph structures like the one used by FPath are encoded naturally in logic terms. This family of languages has the advantage of being formally defined and based on rigorous mathematical theories. As for SQL-derived languages, however, their syntactic structure can make them difficult to integrate with other languages. We are currently using Datalog as part of our work on formalising the semantics of FPath and FScript. As Datalog is more powerful than FPath, this may lead to extensions in future versions of FPath.

The version of FPath presented in this article was inspired by XPath 1.0 [38], but an updated version of XPath [39] has recently been standardized. Some of the new features, like existential and universal quantifiers, might be interesting to add to FPath, especially when FPath is used to express architectural constraints (as, for example, in Section 3.3.3 to check for consistency).

Alia et al. [3] discussed infrastructure support in Fractal for sophisticated querying of the system architectures. Its scope was larger than FPath's, as it also included the querying of component repositories in addition to run-time components. The underlying model was also based on a directed graph representation of Fractal components, which is very similar to the one used in FPath. However, this work was more concerned with infrastructure issues (how to build a query service), and less on the language aspect of queries. To our knowledge, the query language proposed in the paper was never fully implemented.

Component configuration and reconfiguration Several component models support dynamic reconfiguration, but most of them, however, do not provide direct language support for these reconfigurations and rely directly on the implementation programming language instead (OpenCOM [11], K-Component [16]). Also, few of them take into account the reliability of the reconfigurations. Instead, most work on reliability and validation of component-based architectures is done in the context of static ADLs [25]. Wright [4] use a process algebra to ensure that there will be no deadlocks in components interactions. C2 [24] can check the conformance of an architecture description relative to *style constraints*. However, as identified by [29], ADLs are not enough. To support dynamic architectures, one also needs what the author calls an *Architecture Modification Language* (AML) to describe modification operations and an *Architecture Constraint Language* (ACL) to describe the constraints under which architectural modifications must be performed. In the Fractal ecosystem, Fractal ADL[4] already fills the role of a classic (although extensible) ADL to describe initial/static configurations, while FScript and FPath complement it by filling these two additional roles (AML and ACL, respectively).

One interesting work that supports dynamic reconfigurations while still offering strong guarantees is ArchJava [2], which extends Java with component-based concepts. ArchJava architectures can be reconfigured dynamically and the language guarantees communication integrity during execution. However, these guarantees are only possible because the reconfigurations are hard-coded in the program. This is an important limitation because in practice adaptations required during the lifetime of an application can rarely be predicated at the time it is built. Mae [33] proposes an architecture evolution environment using xADL [12]; it has another approach different from ours to reconfigure applications as it is goal-based oriented: an ADL configuration is given as an objective and the difference with the current configuration is automatically performed; the resulting patch is applied on the system.

More recently, component models relying on reflexive architectures to allow nonanticipated (also called ad hoc) reconfigurations while supporting some kinds of guarantees have appeared. FORMAware [27] is relatively close to our work. This framework to program component-based application gives the possibility to constrain reconfigurations with architectural-style rules. A transaction service manages the reconfiguration by stacking operations. The main difference with our proposal is that our integrity constraints are more flexible than styles as they can be applied at the model level or directly to specific instances with pre/postconditions and invariants. Plastik [6] is the integration of the OpenCOM component model and the ACME/Armani ADL [18]. As in our approach, architectural invariants can be checked at run time and constraints are expressed at two levels (style and instance). However, we propose a full support of concurrency for reconfiguration. We define more formally reconfiguration operations to identify conflicts between them so that our locking algorithm is finer than simple locking components; moreover, it also considers introspection operations with shared locks, notably so as to avoid dirty reads.

[4] http://fractal.objectweb.org/fractaladl/index.html.

5 Conclusion and future work

This paper has presented two related (and complementary) languages that make it easier to manage Fractal architectures. FPath is a general query language for Fractal applications inspired by XPath [38]. It provides a convenient notation to navigate inside Fractal architectures and select elements based on their properties or location in the architecture. FPath is based on a uniform and extensible representation of Fractal architectures as directed graphs, and although its syntax is relatively simple, it supports sophisticated queries. FPath was originally designed as a part of FScript, but it can also be used by itself as a general addressing language for Fractal. FPath has been used in [14] and [20] to write the condition part of event–condition–action rules in these autonomic extensions to Fractal. It is also used in [30] as a point-cut language for an aspect-oriented extension of the Fractal model and in [34] to define and verify architectural invariants on Fractal architectures.

FScript relies on FPath for its addressing needs but supports the definition of architectural reconfigurations of Fractal architectures. It is modeled as a simple scripting language with an imperative flavor to make it easy to learn for potential users. FScript supports all the reconfiguration features defined in the standard Fractal model, including structural (bindings and containment), state (attributes), and life-cycle reconfigurations. Like FPath, it is extensible along the same ways as the Fractal model itself. Beyond the convenience of a custom syntax, the main feature of FScript is that its constructs and run-time semantics have been designed to make reconfigurations *reliable*: its restricted (focused) power of expression enables us to guarantee that FScript reconfigurations terminate and have *transactional semantics* and, hence, keep the target system in a consistent, usable state. This makes it possible to apply unanticipated reconfigurations to running systems in a way that minimizes disruption and maximizes the continuity of service. FScript[5] has been used as part of the Safran extension to Fractal [13, 14] to program the reaction part of Safran's autonomic rules, which can automatically adapt a Fractal architecture to context changes. Jayaprakash et al. [20] also uses FScript for similar purposes. Finally, [31] has shown how FScript can be used with the Think [17] implementation of Fractal (targeted at Operating System kernels) to take advantage of Think's reconfiguration features.

A *formal specification* of FPath/FScript is currently being defined. This should enable to provide *static analyses* of FScript programs in order to detect some kinds of errors earlier [15]. This would allow us to reject certain kinds of invalid reconfigurations before actually modifying the system. This formal definition will also be used as a basis for a new, more sophisticated implementation of FPath and FScript, which will include optimizations (something completely missing from the current interpreter).

[5]The FScript interpreter is available for downloading at http://fractal.objectweb.org/fscript/.

References

1. Abdallah M, Guerraoui R, Pucheral P (1998) One-phase commit: does it make sense? In: ICPADS '98: proceedings of the 1998 international conference on parallel and distributed systems. IEEE Computer Society, Washington, DC, p 182
2. Aldrich J, Chambers C, Notkin D (2002) ArchJava: connecting software architecture to implementation. In: International conference on software engineering, ICSE 2002, Orlando, May 2002
3. Alia M, Lenglet R, Coupaye T, Lefebvre A (2004) Querying on reflexive component-based architectures. In: Proceedings of the 30th EUROMICRO conference, Rennes, Sept 2004
4. Allen RJ (1997) A formal approach to software architecture. PhD thesis, Carnegie Mellon University, May 1997. Technical Report Number: CMU-CS-97-144
5. Avizienis A, Laprie J-C, Randell B, Landwehr C (2004) Basic concepts and taxonomy of dependable and secure computing. IEEE Trans Dependable Secur Comput 01(1):11–33
6. Batista T, Joolia A, Coulson G (2005) Managing dynamic reconfiguration in component-based systems. In: 2nd European workshop on software architectures (EWSA 2005). Lecture notes in computer science, vol 3527. Springer Berlin, Heidelberg
7. Bruneton R, Coupaye T, Leclercq M, Quéma V, Stefani J-B (2006) The Fractal component model and its support in Java. Softw Pract Exp 36(11–12):1257–1284 (special issue on Experiences with Auto-adaptive and Reconfigurable Systems)
8. Bruneton R, Coupaye T, Stéfani J-B (2003) The Fractal component model. Technical report, The ObjectWeb Consortium, Sept 2003, version 2.0
9. Cattell R, Barry DK, Berler M, Eastman J, Jordan D, Russell C, Schadow O, Stanienda T, Velez F (eds) (2000) The object data standard – ODMG 3.0. The Morgan Kaufmann series in data management systems. Morgan Kaufmann, San Francisco
10. Ceri S, Gottlob G, Tanca L (1989) What you always wanted to know about Datalog (and never dared to ask). IEEE Trans Knowl Data Eng 1(1):146–166
11. Coulson G, Blair GS, Grace P, Joolia A, Lee K, Ueyama J (2004) A component model for building systems software. In: Proceedings of IASTED software engineering and applications (SEA'04), Cambridge, Nov 2004
12. Dashofy EM, van der Hoek A, Taylor RN (2001) A highly-extensible, XML-based architecture description language. In: Proceedings of the working IEEE/IFIP conference on software architectures (WICSA 2001), Amsterdam
13. David P-C (2005) Développement de composants Fractal adaptatifs : un langage dédié à l'aspect d'adaptation. Ph.D.

thesis, Université de Nantes / École des Mines de Nantes, July 2005
14. David P-C, Ledoux T (2006) An aspect-oriented approach for developing self-adaptive Fractal components. In: 5th international symposium on software composition (SC'06), Vienna, Mar 2006
15. David P-C, Léger M, Grall H, Ledoux T, Coupaye T (2008) A multi-stage approach for reliable dynamic reconfigurations of component-based systems. In: Proceedings of the 8th IFIP international conference on distributed applications and interoperable systems (DAIS'08), Oslo, Norway, June 2008. LNCS. Springer, Heidelberg
16. Dowling J, Cahill V (2001) The K-Component architecture meta-model for self-adaptive software. In: Yonezawa A, Matsuoka, S (eds) Proceedings of reflection 2001, the third international conference on metalevel architectures and separation of crosscutting concerns, Kyoto, Japan (Sept 2001), vol 2192 of Lecture Notes in Computer Science, AITO. Springer, Heidelberg, pp. 81–88
17. Fassino J-P, Stefani J-B, Lawall J, Muller G (2002) THINK: a software framework for component-based operating system kernels. In: Proceedings of the general track: 2002 USENIX annual technical conference, June 10–15, 2002, Monterey, California, USA. USENIX 2002, ISBN 1-880446-00-6
18. Garlan D, Monroe RT, Wile D (2000) Acme: architectural description of component-based systems. In: Leavens GT, Sitaraman M (eds) Foundations of component-based systems. Cambridge University Press, New York, pp 47–67
19. Hajiyev E (2005) CodeQuest – source code querying with datalog. Msc thesis, St. Anne's College, University of Oxford, Oxford University
20. Jayaprakash N, Coupaye T, Collet C, Rivierre N (2005) Des régles actives au sein d'une infrastructure logicielle autonomique. In: Journées Composants 2005, 4éme conférence francophone autour des composants logiciels, Le Croisic, Apr 2005
21. Kephart J, Chess DM (2003) The vision of autonomic computing. IEEE Comput 36(1):41–50
22. Kramer J, Magee J (1990) The evolving philosophers problem: dynamic change management. IEEE Trans Softw Eng 16(11):1293–1306
23. Léger M, Coupaye T, Ledoux T (2006) Contrôle dynamique de l'intégrité des communications dans les architectures à composants. In: Rousseau R, Urtado C, Vauttier S (eds) LMO (2006), Hermés Lavoisier, pp 21–36
24. Medvidovic N, Oreizy P, Robbins JE, Taylor RN (1996) Using object-oriented typing to support architectural design in the C2 style. In: Proceedings of the ACM SIGSOFT'96 fourth symposium on the foundations of software engineering, San Francisco, CA, USA, Oct 1996. ACM SIGSOFT, New York, pp 24–32
25. Medvidovic N, Taylor RN (2000) A classification and comparison framework for software architecture description languages. IEEE Trans Softw Eng 26(1):70–93
26. Mernik M, Heering J, Sloane AM (2005) When and how to develop domain-specific languages. ACM Comput Surv 37(4):316–344
27. Moreira RS, Blair GS, Carrapatoso E (2004) Supporting adaptable distributed systems with FORMAware. In: ICDCSW '04: proceedings of the 24th international conference on distributed computing systems workshops. IEEE Computer Society, Washington, DC, pp 320–325
28. OCL 2.0 Specification (2005) http://www.omg.org/docs/ptc/05-06-06.pdf
29. Oreizy P (1996) Issues in the runtime modification of software architectures. Technical report UCI-ICS-TR-96-35, Department of Information and Computer Science University of California, Irvine
30. Pessemier N, Seinturier L, Coupaye T, Duchien L (2006) A model for developing component-based and aspect-oriented systems. In: Löwe W, Südholt M (eds) Software composition, 5th international symposium, SC 2006, Vienna, Austria, Mar 2006, vol 4089 of Lecture Notes in Computer Science. Springer, Heidelberg, pp 259–274 (Revised papers)
31. Polakovic J, Mazaré S, Stefani J-B, David P-C (2007) Experience with implementing safe reconfigurations in component-based embedded systems. In: The 10th international ACM SIGSOFT symposium on component-based software engineering (CBSE 2007), Boston, MA, USA, July 2007. Lecture Notes in Computer Science, ACM. Springer, Heidelberg
32. Redmond B, Cahill V (2002) Supporting unanticipated dynamic adaptation of application behaviour. In: Proceedings of ECOOP 2002, Malaga, Spain, May 2002, vol 2374 of Lecture Notes in Computer Science. Springer, Heidelberg, pp 205–230
33. Roshandel R, Hoek AVD, Mikic-Rakic M, Medvidovic N (2004) Mae—a system model and environment for managing architectural evolution. ACM Trans Softw Eng Methodol 13(2):240–276
34. Rouvoy R (2006) Une démarche à granularité extrêmement fine pour la construction de canevas intergiciels hautement adaptables : application aux services de transactions. PhD thesis, Université des Sciences et Technologies de Lille, Lille
35. Szyperski C (1997) Component software. ACM, New York
36. Traiger IL, Gray J, Galtieri CA, Lindsay BG (1982) Transactions and consistency in distributed database systems. ACM Trans Database Syst 7(3):323–342
37. Weikum G, Schek H-J (1992) Concepts and applications of multilevel transactions and open nested transactions. Database transaction models for advanced applications. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, pp 515–553
38. World Wide Web Consortium (1999) XML path language (XPath) version 1.0. W3C Recommendation, Nov 1999. http://www.w3.org/TR/xpath/
39. World Wide Web Consortium (2007) XML path language (XPath) version 2.0. W3C Recommendation, Jan 2007. http://www.w3.org/TR/xpath20/