FPath and FScript Reference Manual

Pierre-Charles David
pcdavid@gmail.com

November 20, 2007

Contents

Chapter 1

Introduction

Note: This document is a work in progress. If you have any feedback, please contact `pcdavid@gmail.com`. Thank you.

This document is the reference manual for the FPath and FScript Domain-Specific Languages [David, 2005]. Both languages are designed to make it easier to deal with software systems built using the Fractal Component Model [Bruneton et al., 2003, 2006]. More specifically:

- FPath is a *navigation* language for Fractal architectures. An FPath expression is a query which is run against a set of Fractal components, returning the set of architectural elements (for example components or interfaces) which match the query.
- FScript is a language to program architectural *reconfigurations* of Fractal systems. It builds upon FPath to select the elements which must be reconfigured, but otherwise looks like a simple "scripting language", with a few restrictions. These restrictions are designed so that it is possible to guarantee that FScript reconfigurations are "well-behaved".

One of the main strengths of the Fractal component model is its support for reflection. In particular, the model provides standard ways to introspect Fractal components to discover their internal structure, state and relationships with each other (architecture), and to reconfigure these dynamically. However, Fractal is specified as a set of APIs, and does not rely on syntactic extensions to the implementation languages. In addition, these APIs are designed to be minimal and orthogonal. In practice, this means that using these "raw" APIs can quickly lead to writing very verbose and error-prone code. The main goal of FPath and FScript is to provide more convenient ways to access the full power of the Fractal model, both for introspection (FPath) and reconfiguration (FScript).

Note that the languages are not designed to implement Fractal component, but to introspect and manipulate (instantiate, reconfigure) already implemented components. FPath and FScript only deal with the aspects of components specified in the model. In particular, they do not expose or allow to manipulate implementation-specific information (e.g. the internal structure of a membrane) or business-level behaviour and state (except insofar that this state is exposed using standard Fractal interfaces). The current version of FScript has the additional limitation that it relies on Fractal ADL descriptors to instantiate new components dynamically (see 3.3.9). This limitation may be removed in the future.

This document assumes a good knowledge of the Fractal model. See the reference manual [Bruneton et al., 2003] or the official web site at `http://fractal.objectweb.org/` for details about Fractal. Both the definition and implementation of FPath and FScript are independant on the specifics of a particular implementation of Fractal (Julia, AOKell...), although parts of this specification are currenlty Java-specific. As both languages make heavy use of Fractal's reflective capabilities, the current implementation, which is fully interpreted, requires taht components are compliant with level 3.3 (see page 28 of the specification for the definition of compliance levels).

This document only describes the languages syntax and semantics. Instructions on how to use the standard implementation to invoke FPath and FScript programs from Java can be found in the implementation's Javadoc and in the tutorial on the web[1], also available in the project source.

Chapter 2 first describes FPath as a standalone query language. Chapter 3 then present the full FScript language, which builds on FPath.

[1] `http://fractal.objectweb.org/tutorials/fscript/index.html`

Chapter 2

FPath

FPath is an expression language to query live Fractal architectures. By *expression language* we mean that FPath is side-effect free: executing an FPath program can not modify the target system. By *query* we mean that an FPath expression is used to select architectural elements in a target system according to some criteria. By *live Fractal architectures* we mean that FPath is primarily designed to target instanciated Fractal components, as opposed to static descriptions of such components like Fractal ADL definitions[1].

The design objectives of FPath are the following:

- the syntax should be *simple to read and understand*, and reasonably *compact*;
- the language should provide *full access to all the introspection capabilities* of the Fractal model. Note however that FPath only deals with components at the architecture level, and hence only "knows" about concepts defined in the Fractal specification. In particular, FPath does not give access to information specific to an implementation (e.g. the mixin structure of a Julia membrane) or to the implementation code of primitive components;
- the language (especially its syntax), should be *extensible* to transparently support extensions to the Fractal model which are not defined in the base specification. By transparently we mean that it should not be possible to distinguish between standard features and extensions.
- the language semantics should be defined at a *high level of abstraction*, to be *simple to understand* while supporting *efficient implementations*.

[1] It might be possible to use FPath or a variant of it to query static architecture descriptions. This version of the language does not try to support this use case.

Both the syntax and the execution model of FPath are inspired by the XPath World Wide Web Consortium [1999] language. XPath is the standard query language for XML documents defined by the W3C, used by many other XML technologies (XLink, XSLT, XQuery...). XPath is defined on an abstract representation of XML documents as sets of nodes which represent the content of an XML document (elements, attributes, textual content...), and relations between these nodes which represent its structure (an element contains another, which itself has some attributes). XPath expressions can navigate along these relations to select the nodes representing parts of the document based either on their properties or their location in the document.

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 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 than the Fractal model itself. If a new Fractal extension is defined which introduces new architectural elements and/or new relations between components (for example an aspect weaving relationship), 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 C1 named C2"), but also more generic and less brittle (but also less precise) 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 C1").

- The syntax is reasonably consise 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 does not use any XML representation of Fractal architectures.

In addition to the path expressions used to navigate inside the architecture, FPath also supports more standard kinds of expressions and primitive values, and

a standard library of functions to operate on those . In the rest of this chapter, we first describe primitive values and expressions before going into the details of nodes and path expressions.

2.1 Primitive Values and Expressions

2.1.1 Numbers and arithmetic operations

FPath supports only one kind of numbers, which correspond to Java's `double` type. They have the following litteral syntax:

⟨*number*⟩ ::= ⟨*opt-sign*⟩ ⟨*digits*⟩ ⟨*opt-decimal-part*⟩

⟨*opt-sign*⟩ ::= '`+`' | '`-`' | ⟨*empty*⟩

⟨*digits*⟩ ::= ⟨*digit*⟩ | ⟨*digit*⟩ ⟨*digits*⟩

⟨*opt-decimal-part*⟩ ::= '`.`' ⟨*digits*⟩ | '`.`' | ⟨*empty*⟩

⟨*digit*⟩ ::= '`0`' | '`1`' | '`2`' | '`3`' | '`4`' | '`5`' | '`6`' | '`7`' | '`8`' | '`9`'

Numbers without a decimal part are automatically converted to floating point representation, and all operations in FPath are done using `doubles`.

Here are a few examples:

```
1
-0.0
3.14159
-2.
+42
01234.56789
```

The four arithmetic operations (addition, substraction, multiplication, division) are available, with the standard precendence rules, but the division operator is `div` instead of `/` to avoid confusion with the path separator. Their semantics is the same as in Java, including the treatment of negative zero, infinities, and *NaN* results. A division by zero aborts the evaluation of the whole FPath expression with an error.

```
3+4*5        // => 23.0
(3+4)*5      // => 35.0
3 div 4      // =>  0.75
3 * -1.0 + 7 // =>  4.0
```

2.1.2 Strings

FPath supports Unicode strings as a base data type, and provides a set of standard functions to operate on them. Both single-quoted and double-quoted string litterals are supported, with some standard escape characters:

⟨*string*⟩ ::= ⟨*sq-string*⟩ | ⟨*dq-string*⟩

⟨*sq-string*⟩ ::= ‘’’ ⟨*sq-chars*⟩ ‘’’

⟨*dq-string*⟩ ::= ‘"’ ⟨*dq-chars*⟩ ‘"’

⟨*sq-chars*⟩ ::= ⟨*escape*⟩ ⟨*sq-chars*⟩ | ⟨*sq-char*⟩ ⟨*sq-chars*⟩ | ⟨*empty*⟩

⟨*dq-chars*⟩ ::= ⟨*escape*⟩ ⟨*dq-chars*⟩ | ⟨*dq-char*⟩ ⟨*dq-chars*⟩ | ⟨*empty*⟩

⟨*escape*⟩ ::= ‘\n’ | ‘\r’ | ‘\t’ | ‘\"’ | ‘\’’

where ⟨*sq-char*⟩ (resp. ⟨*dq-char*⟩) is the set of all Unicode characters except for backslash (‘\’) and the single-quote character ‘’’ (resp. the double-quote character ‘"’).

For example:

```
"a simple string"
'another one'
""
'He said "Hello".'
'It\'s not what I expected...\n'
```

2.1.3 Booleans

FPath supports booleans and the corresponding operations. There is no litteral syntax for boolean values, but they can be obtained by calling the built-in functions `true()` and `false()`.

The boolean operations supported are: conjunction, disjunction and negation. The first two are supported through the operator `and` and `or`, conjunction having a higher priority. Negation is supported using the predefined function `not()`.

For example:

```
true() and false()                // = false()
true() or false() and true()      // = true()
not(true() and false()) or true() // = true()
```

2.1.4 Variables

Although FPath expression can not *define* variables or change variables values, they can *use* variables defined elsewhere. When using FPath as a standalone query language, variable definition is done in the calling program using the evaluator's API (not specified here). When using FPath as part of FScript, variables can be defined and modified by the enclosing FScript program.

Referencing a variable is done by prefixing its name with a dollar sign (`$`):

⟨*variable*⟩ ::= '`$`' ⟨*name*⟩

Examples of valid variable names include:

```
$var
$x3
$a_long_variable_name
$anotherOneInADifferentStyle
$yet-another-style
$_
```

2.1.5 Function calls

FPath expressions can invoke functions. These functions can be used either to manipulate primitive values (numbers, strings and booleans) or to introspect node values (representing elements in the architecture). However, no function call can modify these architectural elements.

The syntax of function call is the following:

⟨*call*⟩ ::= ⟨*name*⟩ '`(`' ⟨*opt-arguments*⟩ '`)`'

⟨*opt-arguments*⟩ ::= ⟨*arguments*⟩ | ⟨*empty*⟩

⟨*arguments*⟩ ::= ⟨*expression*⟩ | ⟨*expression*⟩ '`,`' ⟨*arguments*⟩

For example:

```
ends-with("-controller", name($anInterface))
compatible($clItf, $srvItf)
bound($clItf)
```

2.1.6 Opaque Object Values

Although they do not have a litteral syntax, it is possible for an FPath function to return a raw value from the underlying implementation language (for example an object in Java). These values can be stored in variables and used with compatible, user-defined functions. This is intended to enable convenient extension of the library of functions. For example, one could imagine an extension to support regular expressions matching:

```
re = make-regexp("fo+");
isFoo = matches("foo", $re);
```

Where `make-regexp` is a user-supplied function which returns a compiled regurlar expession object as an opaque value which is then used by `matches`.

None of the predefined functions produces or uses such values.

2.1.7 Comparisons

General equality an inequality tests are available for all primitive and node types, while ordering oprators are available only for numbers.

⟨*comparison*⟩ ::= ⟨*equality-test*⟩ | ⟨*ordering-test*⟩

⟨*equality-test*⟩ ::= ⟨*expression*⟩ '`==`' ⟨*expression*⟩ | ⟨*expression*⟩ '`!=`' ⟨*expression*⟩

⟨*ordering-test*⟩ ::= ⟨*expression*⟩ ⟨*ordering-operator*⟩ ⟨*expression*⟩

⟨*ordering-operator*⟩ ::= '`<`' | '`>`' | '`<=`' | '`>=`'

For example:

```
"foo" != "bar"
name($itf) == "component"
1 + 2 == 3
42 <= 43
```

2.2 Nodes and Path Expressions

2.2.1 Graph Representation of Fractal Architectures

A Fractal architecture (i.e. a set of Fractal components) is modeled in FPath as a directed graph with labeled arcs. The nodes in the graph represent the different kinds of architectural entities which are visible in FPath, while the labeled arcs

represent their relations. Note that this graph representation is just a model; implementations do not have to use a graph representation of the architecture as long as they conform to the language semantics

FPath defines a standard set of node types and their relations, which correspond to the concepts defined in the Fractal specification. However, neither the semantics nor the syntax of the language is specific to these particular node types and relations. This is the very feature which makes FPath extensible in the same way that Fractal is: if a new model extension is defined, supporting it in FPath amounts only to adding support to the appropriate new node types and relations. It does not require changes in the syntax or in the implementation. Indeed, an FPath implementation must support the addition of new types of nodes and relations without requiring to be modified.

The following subsections describe the standard, predefined node types and relations (arcs).

2.2.2 Nodes

Nodes represent elements of interest in a Fractal architecture. These do not correspond exactly to the kinds of "objects" reified by the Fractal APIs, but more to the concepts that users want to be able to talk about. For example, Fractal does not have a specific concept of component: the `component` interface is just one of the standard control interfaces (it is *used* to identify components, but it is of the same nature as, for example, the `content-controller` interface). In contrast, FPath has a specific kind of node to represent the components themselves, in addition to the nodes representing their interfaces (including the `component` interface).

The standard kinds of nodes supported by FPath are:

- *component* nodes, which represent the components themselves. Exactly one such node exist in the graph for each component in the Fractal architecture it represents.

- *interface* nodes, which represent the component interfaces. Interface nodes are used to represent *all* the interfaces, be they control or service interface, client or server, internal or external. Exactly one interface node exist for each interface in the architecture. If two components have an "identical" interface (for example a `name-controller`), two different nodes are used in the graph.

- *attribute* nodes, which represent the configuration attributes of the components, as defined by their (optional) `attribute-controller` interface. Configuration attributes are not reified explicitely in the Fractal specification,

but available only through naming conventions – namely the JavaBeans conventions – on the methods of the `attribute-controller` interface. One attribute node exist for each pair of setter/getter method in this interface.

Each node has a name (accessible through the predefined `name()` function), and each node type defines how a node can be converted into each of the three primitive value types (numbers, strings, booleans).

Unless specified otherwise for a particular node type, two nodes are equal (as defined by the `=` operator) if and only if they represent the same architectural element.

In the following discussion we will sometimes identify nodes with the architectural elements they represent for simplicity when there is no risk of confusion.

Component nodes

Component nodes represent the actual Fractal components in the target architecture. They are different from the interface node which represent their `component` interface. Each component in the architecture, whether a primitive, composite or another kind, is represented by exactly one component node.

The *name* of a component node is defined as:

1. The value returned by the `NameController#getFcName()` method of the standard `name-controller` interface if the component provides this interface (or a compatible extension of it).
2. The empty string (`""`) otherwise.

When converted into a *string*, the value of a component node is the node's name. When converted into a *boolean*, its value is always `true()`. Finally, when converted into a *number*, the value of a component node is Java's `Double.NaN` (*Not a Number*).

The following pre-defined functions give access to additional information on component nodes:

- The `state(a`*`CompNode`*`)` function takes a component node in argument and returns a string indicating the current lifecycle state of the corresponding component. More precisely, it returns:
  1. The value returned by the `#getFcState()` method of the standard `lifecycle-controller` interface if the component provides this interface (or a compatible extension of it).
  2. The empty string (`""`) otherwise.

- The `started(a`*`CompNode`*`)` function takes a component node in argument and returns `true()` if `state(aCompNode) = "STARTED"`, `false()` otherwise.

- The `stopped(a`*`CompNode`*`)` function takes a component node in argument and returns `true()` if `state(aCompNode) = "STOPPED"`, `false()` otherwise.

Interface nodes

Interface nodes represent component interfaces. Each interface of each component present in the architecture, whether it is internal or external, client or server, is represented by exactly one interface node.

The *name* of an interface node is defined as the result of the `Interface#getFcItfName()` method of the `Interface` object representing the component interface.

When converted into a *string*, the value of an interface node is the node's name. When converted into a *boolean*, its value if always `true()`. Finally, when converted into a *number*, the value of an interface node is Java's `Double.NaN`.

The following pre-defined functions give access to additional information on interface nodes:

- The `client(`*`anItfNode`*`)` function takes a single interface node as argument and returns `true()` if the interface is a client (i.e. required) interface, and `false()` otherwise. More precisely, it returns the same value as the Java expression `((InterfaceType) itf.getFcType()).isFcClientItf()` where `itf` denotes the interface represented by *`anItfNode`*[2].

- The `server(`*`anItfNode`*`)` function takes a single interface node as argument and returns `true()` if the interface is a server (i.e. provided) interface, and `false()` otherwise. It is equivalent to `not(client(anItfNode))`.

- The `optional(`*`anItfNode`*`)` function takes a single interface node as argument and returns `true()` if the interface is optional, and `false()` otherwise. More precisely, it returns the same value as the Java expression `((InterfaceType) itf.getFcType()).isFcOptionalItf()` where `itf` denotes the interface represented by *`anItfNode`*.

- The `mandatory(`*`anItfNode`*`)` function takes a single interface node as argument and returns `true()` if the interface is mandatory (i.e. not optional), and `false()` otherwise. It is equivalent to `not(optional(`*`anItfNode`*`))`.

- The `collection(`*`anItfNode`*`)` function takes a single interface node as argument and returns `true()` if the interface is a collection interface, and `false()`

[2]Like most of the functions described in this section, this assumes that the type of the interface is (a subtype of) the standard `org.objectweb.fractal.api.type.InterfaceType`.

otherwise (if it is a single interface). More precisely, it returns the same value as the Java expression `((InterfaceType) itf.getFcType()).isFcCollectionItf()` where `itf` denotes the interface represented by *`anItfNode`*.

- The `single(`*`anItfNode`*`)` function takes a single interface node as argument and returns `true()` if the interface is a single interface (i.e. not collection), and `false()` otherwise. It is equivalent to `not(collection(`*`anItfNode`*`))`.

- The `internal(`*`anItfNode`*`)` function takes a single interface node as argument and returns `true()` if the interface is an internal interface, and `false()` otherwise (if it is an external interface). More precisely, it returns the same value as the Java expression `itf.isFcInternalItf()` where `itf` denotes the interface represented by *`anItfNode`*.

- The `external(`*`anItfNode`*`)` function takes a single interface node as argument and returns `true()` if the interface is an external interface, and `false()` otherwise (if it is an internal interface). It is equivalent to `not(internal(`*`anItfNode`*`))`.

- The `signature(`*`anItfNode`*`)` function takes a single interface node as argument and returns a string representing the fully qualified name of the Java interface implemented (or required) by the interface. More precisely, it returns the same value as the Java expression `((InterfaceType) itf.getFcType()).getFcItfSignature()` where `itf` denotes the interface represented by *`anItfNode`*.

- The `bound(`*`anItfNode`*`)` function takes a single interface node as argument and returns `true()` if the interface is a client interface and it is currently bound to (at least) one server interface, and `false()` otherwise.

- The `compatible(`*`anItfNode1`*`, `*`anItfNode2`*`)` function takes two interface nodes as parameters and returns `true()` if:

  1. *`anItfNode1`* is a client interface;
  2. *`anItfNode2`* is a server interface;
  3. it would be possible to create a direct binding from *`anItfNode1`* to *`anItfNode2`* (see [Bruneton et al., 2003] for the detailed conditions).

  The function returns `false()` otherwise.

Attribute nodes

Attribute nodes represent configuration attributes/parameters of Fractal components, as exposed through the standard `attribute-controller` interface. These attributes are not explicitely reified by the Fractal model, as they are only defined through method naming convention. However, as they provide one of the main ways to dynamically configure components, FPath represents them explicitely.

If a component does not provide the standard `attribute-controller` interface, it does not contribute any attribute node.

FIXME: *The following description is badly worded, and does not support read-only attributes.* Given a component which provides the standard `attribute-controller` interface, let `T` be the signature of that interface (i.e. `T` is a Java interface type which extends `AttributeController`). Each pair of methods of `T` of the form *`V`* `get`*`X`*`()` / `void set`*`X`*`(`*`V`*`)` or *`B`* `is`*`X`*`()` / `void set`*`X`*`(`*`B`*`)`, where *`X`* is a valid Java identifier, *`V`* represents any Java type (including primitive types) and *`B`* represents either `boolean` or `Boolean`, defines one attribute node. The name of this attribute node is the identifier *`X`* where the first character (always a letter) is converted to lower case. If for some identifier *`X`* the three methods *`B`* `is`*`X`*`()`, *`B`* `get`*`X`*`()` and `void set`*`X`*`(`*`B`*`)` exist, only one attribute node represents the attribut *`X`*.

TODO: *The Fractal specification supports write-only attributes (see p. 13). Should we support these too? Are there any cases when they might be useful?*

When converted into a *string*, the value of an attribute node is the node's name. When converted into a *boolean*, its value if always `true()`. Finally, when converted into a *number*, the value of an attrubute node is Java's `Double.NaN`.

The following pre-defined functions give access to additional information on attribute nodes:

- The `value(`*`anAttrNode`*`)` function takes a single attribute node as argument and returns the current value of the corresponding attribute. More precisely, if the name of the attribute node is *`x`*, the `value()` returns the same value as the following Java expression:

  ```
  ((T) c.getFcInterface("attribute-controller")).getX()
  // or
  ((T) c.getFcInterface("attribute-controller")).isX()
  ```

  where:

  - `c` is the `component` interface of the component which contributed *`anAttrNode`*;
  - `T` is the actual type of that component's `attribute-controller`;
  - `X` is the capitalized version of the attribute's name.

2.2.3 Axes

The different kinds of nodes defined in the previous section represent the individual elements which are visible in FPath. The *structure* of the application, i.e. its architecture, is represented by how these elements are connected to each other in a oriented graph. Each arc in the graph is annotated by a label which denotes a particular relation between the two nodes it connects. The different possible relations between nodes are called *axes*, and each is identified by a label. FPath defines a standard set of axes which include all the relations defined in the Fractal specification which can exist between the pre-defined node types. As for node types, the set of available axes can be extended without changing the language syntax or semantics to take into account extensions to the Fractal model.

- The `component` axis connects every single node (including component nodes) to the component node which represent their "owner" Fractal component.
- The `interface` axis connects component nodes to the interface nodes which represent external interfaces of the corresponding component.
- The `internal-interface` axis connects component node to the interface nodes which represent internal interfaces of the corresponding component.
- The `attribute` axis connects component node to the attribute nodes which represent configuration attributes of the corresponding component.
- The `binding` axis connects client interface nodes to the server interface node they are bound to, if any.
- The `child` axis connects component nodes representing composites to the component nodes representing their direct sub-components. The `child-or-self` axis is a superset of the `child` axis which also connects each component node to itself.
- The `parent` axis connects component nodes to the component nodes representing their direct parents. The `parent-or-self` axis is a superset of the `parent` axis which also connects each component node to itself.
- The `descendant` axis is the transitive closure of the `child` axis. It connects a given component node to all the components it contains, directly or indirectly (including itself in the case of `descendant-or-self`).
- The `ancestor` axis is the transitive closure of the `parent` axis. The `ancestor` axis connects a given component node to all its direct or indirect parents.

- The `sibling` axis connects a component node to all the components which share at least one direct parent with it, i.e. all the components which are "at the same level" in the architecture.
- The `descendant-or-self`, `ancestor-or-self`, and `sibling-or-self` axes are variants of respectively `descendant`, `ancestor`, and `sibling` which also connect the origin component node to itself.

2.2.4 Nodesets

A nodeset is a set of node values of the same type (i.e. only component nodes, or attribute nodes, but not component and attribute nodes in the same set). Nodesets are used to offer aggregate operations and avoid explicit loops. For example, path expressions (described in the next section) take nodesets as input and produce nodesets. Functions can also take nodesets as arguments and produce nodesets as a result.

When a single node value is present in a context where a nodeset value is expected, the node is automatically wrapped in a singleton nodeset. On the other hand, when a nodeset is present in a context where a single node value is expected, then:

- if the nodeset is a singleton, the node it contains is automatically unwrapped;
- otherwise, the behaviour depends on the operator or function which expected the node value. It can either:
  - abort with an error;
  - if the nodeset is non-empty, select one of the node (based on any criteria) to use as the value to operate on;
  - apply itself repeatedly for each of the values in the nodeset (providing an implicit loop).

FIXME: *This irregular behaviour in how nodeset conversion is handled is very bad design. It should be fixed in future versions of the language.*

2.2.5 Path Expressions

Path expressions are used to navigate inside the graph structure described in the previous sections. A path is evaluated relative to an *initial nodeset*, and produces a *result nodeset* containing the architectural elements matching the query. This evaluation is a multi-step process, each step producing an intermediate nodeset result, which is then used as input to the next step, if any, or as the final result for the last step. Each step is itself (conceptually) split into up to three phases:

1. *selection* of a set of candidate nodes, by navigating inside the graph, from each of the initial nodes along a specified axis;
2. (optional) *testing* of the candidate nodes on their names;
3. (optional) *filtering* of the matching candidates using predicate expressions.

The set of candidate nodes which match both the name test and all the predicates is the result nodeset for that step.

Syntax

A path expression is made of an initial, non-path, expression whose value will be the initial nodeset, and a sequence of at least one step, each prefixed by a slash character:

⟨*path*⟩ ::= ⟨*expression*⟩ ⟨*steps*⟩

⟨*steps*⟩ ::= '`/`' ⟨*step*⟩ | '`/`' ⟨*step*⟩ ⟨*steps*⟩

Each step is itself made of two to three parts: an *axis specifier*, a *test*, and an optional sequence of *predicates*:

⟨*step*⟩ ::= ⟨*axis*⟩ '`::`' ⟨*test*⟩ ⟨*opt-predicates*⟩

⟨*axis*⟩ ::= ⟨*name*⟩

⟨*test*⟩ ::= '`*`' | ⟨*name*⟩

⟨*opt-predicates*⟩ ::= ⟨*empty*⟩ | ⟨*predicates*⟩

⟨*predicates*⟩ ::= '`[`' ⟨*expression*⟩ '`]`' | '`[`' ⟨*expression*⟩ '`]`' ⟨*predicates*⟩

Syntaxically, the axis specifier can be any valid identifier. This is what makes FPath extensible in terms of the node types and relations it can handle. Although this specification defines a fixed set of predefined axes (`component`, `child`...), the language can handle extensions to the Fractal model which define new relations between components. For example, one could define an `aspect` axis which connects base components to aspect components Pessemier [2006] which affects them without requiring changes to the FPath language syntax or semantics.

Semantics

Given a step of the from $axis::test[pred_1]\ldots[pred_n]$ and *current* the initial nodeset, the result nodeset produced by this step is computed by the following algorithm:

S1. [Initialisation] $result \leftarrow \emptyset$.

S2. [Selection] Select every node connected to any of the current ones through an arc whose label matches the *axis* part: $result \leftarrow \bigcup\{n : c \xrightarrow{axis} n, c \in current\}$.

S3. [Test] If the *test* is an identifier (as opposed to $*$), remove from *result* the nodes whose name do not match: $result \leftarrow \{n \in result : name(n) = test\}$.

S4. [Filtering] Only keep the elements for which all predicates hold: $result \leftarrow \{x \in result : pred_1(x) \wedge \cdots \wedge pred_n(x)\}$.

S5. [End] The algorithm finishes and returns *result*.

Given a full path of the form $expr/step_1/\ldots/step_n$, the result nodeset produced by the full path is computed by the following algorithm:

P1. [Initialisation] Evaluate the initial expression *expr*, which must produce a nodeset and put the result in *current* (if *expr* produces a single node, it is automatically converted into a singleton nodeset first). $i \leftarrow 1$.

P2. [Step] Evaluate $step_i$ using algorithm S above, using *current* as the initial nodeset, and put back the result into *current*.

P3. [Loop] Increment i: $i \leftarrow i + 1$. If $i \leq n$, goto P2.

P4. [End] The algorithm finishes and returns *current* as the value of the full path expression.

Chapter 3

FScript

3.1 Procedures Definitions

An FScript program is made of a sequence of top-level procedure definitions. FScript supports two kinds of procedures: *functions* and *actions*. Functions can only use introspection features and are hence guaranteed to be side-effect free. They can be used in FPath expression (for example in predicates) to supplement the standard library of FPath functions. 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 syntaxic difference between functions and actions appears in their definitions (Note: maybe we should change this to make it clearer by reading source code which calls correspond to pure functions or actions. Maybe something like the `mutation!` convention used by Scheme. On the other hand maybe this would make FScript code too ugly and actually harder to read). Functions are introduced by the keyword `function` while actions use the keyword `action`:

```
function my-function(<parameters>) { <body> }

action   my-action(<parameters>) { <body> }
```

Recursive definitions, direct or indirect, are forbidden:

```
function f()  { ...; f(); ...; } // ERROR: direct recursiion

function f1() { ...; f2(); ... }
function f2() { ...; f1(); ... } // ERROR: indirect recursiion
```

Inside the body of a procedure, the values of the arguments are available as variables with names matching the parameter names in the signature:

```
action do-something(x, y) {
  do-this($x);
  do-that($y);
}
```

3.2 Control Structures

FScript procedures support a limited set of control structures so that it is possible to ensure that they will enventually terminate (there is no bound on the time they can take however).

3.2.1 Variables Assignment

New local variables can be created inside the body simply by assigning them an initial value:

```
x = <some-expression>;
```

Variable values can be changed simply by re-assigning them:

```
x = <some-expression>;
... // do something with $x
x = <another-expression>;
... // do something else with the new value of $x
```

Invocation parameters behave exactly like local variable except that they are automatically assigned before entering the body of the procedure. In particular, this means their value can be reassigned:

```
action f(x) {
  do-something($x);
  x = ...;
  do-something-else($x);
}
```

FIXME: *Would there be any advantages to forbid the redefinition of variable?*

TODO: *It would probably be a good idea to have a* `var v = ...;` *statement to explicitely introduce new variables as it would catch some typos. It could be optional at first, with a simple warning when a variable is created without using it.*

3.2.2 Sequence and Blocks

Sequences of instructions can be grouped in blocks, where each instruction is ended by a semicolon ';'. Blocks themselves are enclosed in braces `{ ... }`. Blocks can not be introduced anywhere where an instruction is expected; the FScript grammar uses blocks only at certain points:

- the body of a procedure must be a block;
- the "then" and "else" parts of a conditional must be blocks;
- the body of an iteration must be a block.

3.2.3 Conditionals

Conditional execution is supported using the following two constructs:

```
if (<test>) {
  <block-body>
}

if (<test>) {
  <block-body>
} else {
  <block-body>
}
```

where `<test>` must be an FPath expression (i.e. it can not be an action invocation, which can have side-effects). The test expression is first evaluated, and its result converted into a boolean value. If it is *true*, the `then` block is exectuded. If it is *false*, the first construct does nothing, while the second executes the `else` block.

3.2.4 Iteration

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

```
for <var> : <expression> {
  <block-body>
}
```

The `<expression>` must evaluate to a nodeset (or a single node value which will be automatically converted to a singleton nodeset). The body of the block is executed exactly once for each node in the nodeset, with the corresponding value bound to `<var>`.

3.2.5 Explicit return

At any point during its execution, a procedure can stop its execution and immediatly return to the caller, optionally yielding a value, using a `return` statement:

```
return;
return <expression-or-action>;
```

3.3 Primitive Actions

FScript includes a standard library of primitive actions which correspond to all the reconfiguration actions defined in the Fractal specification [Bruneton et al., 2003]. This library can of course be extended to support new operations introduced by extension to the model. For example, one could imagine new primitives to support weaving and unweaving of *aspect components* as supported by FAC [Pessemier et al., 2004].

An implementation of FScript must provide a standard way to add new primitive operations.

The rest of this section describes all the primitive actions included in the standard library.

TODO: *For each action: be more explicit on the error conditions, and show the equivalent Java code.*

3.3.1 Changing attribute values: `set-value()`

The `set-value(`*`attr`*`, `*`value`*`)` primitive action is used to change the value of a component's attribute. It takes two parameters:

- *`attr`* is an FPath expression which must evaluate to a non-empty nodeset containing attribute nodes;
- *`value`* is an FPath expression which must evaluate to a basic value (i.e. not a node or a nodeset).

As mentioned earlier (Sec. 2.1.1), FPath, and hence FScript, normally only deals with floating point numbers. The only exception is when a numeric value is

used to set the value of a Fractal component attribute: if the attribute's type (as defined by the signature of the corresponding *setter* method in the components `attribute-controller`) is an integral type[1], or a corresponding *wrapper* reference type, the number's value is automatically converted into the appropriate type. This conversion is done using the method `Number.`*x*`Value()`, where *x* is the target integral type (i.e. `byte`, `short`...).

3.3.2 Creating bindings between interfaces: `bind()`

The `bind(`*clitf*`, `*srvitf*`)` primitive action is used to bind a client interface to a compatible server interface. It corresponds to the `bindFc(string, any)` method of the standard `BindingController` interface (see [Bruneton et al., 2003, Sec. 4.3, p. 14]).

3.3.3 Destroying bindings between interfaces: `unbind()`

The `unbind(`*clitf*`)` primitive action is used to destroy a binding from client interface to a server interface. It corresponds to the `unbindFc(string)` method of the standard `BindingController` interface (see [Bruneton et al., 2003, Sec. 4.3, p. 14]).

3.3.4 Adding sub-components to composites: `add()`

The `add(`*parent*`, `*child*`)` primitive action is used to add a new sub-component to a composite component. It corresponds to the `addFcSubComponent(Component)` method of the standard `ContentController` interface (see [Bruneton et al., 2003, Sec. 4.4, p. 15])

3.3.5 Removing sub-components from composites: `remove()`

The `remove(`*parent*`, `*child*`)` primitive action is used to remove a sub-component from a composite component. It corresponds to the `removeFcSubComponent()` method of the standard `ContentController` interface (see [Bruneton et al., 2003, Sec. 4.4, p. 16])

3.3.6 Changing a component name: `set-name()`

Text `set-name(`*comp*`, `*name* primitive action is used to change the name of a component, as defined by its `name-controller` interface. If the component denoted by

[1] Java's integral type are `byte`, `short`, `int`, `long` and `char` [Gosling et al., 2005, Sect. 4.2.2, p. 36]

comp does not implement `name-controller`, calling `set-name()` results in an error. Otherwise, it corresponds to the `setFcName(String)` method of the standard `NameController` interface (see [Bruneton et al., 2003, Sec. 4.4, p. 17]).

3.3.7 Starting a component: `start()`

The `start(`*comp*`)` primitive action is used to ensure that the component passed in argument is in the `STARTED` state. If the component denoted by *comp* does not implement `lifecycle-controller`, calling `start()` results in an error. Otherwise, it corresponds to the `startFc()` method of the standard `LifeCycleController` interface (see [Bruneton et al., 2003, Sec. 4.5, p. 17]).

3.3.8 Stopping a component: `stop()`

The `stop(`*comp*`)` primitive action is used to ensure that the component passed in argument is in the `STOPPED` state. If the component denoted by *comp* does not implement `lifecycle-controller`, calling `stop()` results in an error. Otherwise, it corresponds to the `stopFc()` method of the standard `LifeCycleController` interface (see [Bruneton et al., 2003, Sec. 4.5, p. 17]).

3.3.9 Instantiating new components: `new()`

The `new(`*tmplName*`)` primitive action is used to instanciate a new component. Its argument *tmplName* must be the fully qualified name of a Fractal ADL component definition, as a string, and returns a component node representing the newly instanciated component (this action is the only pre-defined action which returns a value). It corresponds to the `newComponent(String, Map)` method of the Fractal ADL `Factory` interface, the second parameter being `null`[2].

```
// Fscript
c = new("com.myapp.SomeComponent");

// Java
Factory fact = FactoryFactory.getFactory(FRACTAL_BACKEND);
Component c = fact.newComponent("com.myapp.SomeComponent", null);
```

[2]Future versions of FScript may provide more control on the instanciation process, and perhaps independance from Fractal ADL definitions.

Bibliography

Éric Bruneton, Thierry Coupaye, and Jean-Bernard Stéfani. The fractal component model. Technical report, The ObjectWeb Consortium, September 2003. version 2.0.

Éric Bruneton, Thierry Coupaye, Matthieu Leclercq, Vivien Quéma, and Jean-Bernard Stefani. The Fractal Component Model and its Support in Java. *Software Practice and Experience, special issue on Experiences with Auto-adaptive and Reconfigurable Systems*, 36(11-12):1257–1284, 2006.

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.

James Gosling, Bill Joy, Guy Steele, and Gilad Bracha. *The Java Language Specification.* Addison-Wesley, third edition, 2005.

Nicolas Pessemier. FAC: Fractal aspect component. Séminaire France Télécom R&R, 23 mai 2006, May 2006. Présentation des travaux de Jacquard financés par le CRE 46131097 de France Télécom.

Nicolas Pessemier, Lionel Seinturier, Laurence Duchien, and Olivier Barais. Une extension de fractal pour l'AOP. In *Première Journée Francophone sur le Développement de Logiciels Par Aspects (JFDLPA'04)*, Paris, September 2004.

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