An Aspect-Oriented Approach for Developing Self-Adaptive Fractal Components

Pierre-Charles David[1] and Thomas Ledoux[2]

[1] France Télécom, Recherche & Développement
28, chemin du vieux chêne
F-38243 Meylan
PierreCharles.David@francetelecom.com
[2] OBASCO Group, EMN / INRIA, Lina
École des Mines de Nantes
4 rue Alfred Kastler
F-44307 Nantes CEDEX 3
Thomas.Ledoux@emn.fr

Abstract. Nowadays, application developers have to deal with increasingly variable execution contexts, requiring the creation of applications able to adapt themselves autonomously to the evolutions of this context. In this paper, we show how an aspect-oriented approach enables the development of self-adaptive applications where the adaptation code is well modularized, both spatially and temporally. Concretely, we propose SAFRAN, an extension of the Fractal component model for the development of the adaptation aspect as reactive adaptation policies. These policies detect the evolutions of the execution context and adapt the base program by reconfiguring it. This way, SAFRAN allows the modular development of adaptation policies and their dynamic weaving into running applications.

1 Introduction

Nowadays, application developers have to deal with increasingly variable execution contexts. On the one hand, we find a large diversity of platforms covering a wide spectrum in terms of available resources (from embedded systems to grids), these heterogeneous machines being increasingly interconnected, and hence interdependent. On the other hand, even on a particular host the execution context of an application changes during its execution (hardware and software resources availability, mobility...). This situation makes application development more and more complex, as it is often difficult to know at development-time the conditions in which applications will be used, especially when these conditions can change unpredictably during execution. Instead of trying to hide the execution context under an abstraction layer (middleware), we believe that applications must become *context-aware* so that they can adapt to their context [1]. Such *self-adaptive applications* are able to adapt themselves autonomously [2] to the evolutions of their execution context, not only to continue functioning but also to leverage new possibilities which can appear dynamically.

The need to build applications which adapt to their environment is not new. However, the ad hoc techniques generally used, in which adaptation decisions are hardwired in applications, are not sufficient: they mix business concerns with adaptation policies, which makes both initial development and maintenance more difficult [3]. Furthermore, it is generally impossible to predict during the development phase the actual circumstances in which applications will be used, even less the appropriate reaction. Ideally, we would like to be able to develop the adaptation code *separately* and then *integrate* it dynamically inside the business code so as to decouple these two kinds of code, both spatially and temporally.

In this paper, we use an aspect-oriented approach [4] to modularize the adaptation code in self-adaptive applications. Aspect-Oriented Programming (AOP) gives us an interesting framework to separate the adaptation concern from business code and then to dynamically weave and un-weave them. The system we propose, SAFRAN, allows to develop self-adaptive applications based on the Fractal component model [5]. SAFRAN is designed around three main principles: *(i)* the use of a dynamic component model (Fractal) to build applications which can be adapted at runtime; *(ii)* the use of AOP concepts and techniques to develop the adaptation logic separately from business code and then to dynamically weave them to yield self-adaptive applications; *(iii)* and finally the use of a Domain (or Aspect) Specific Language [6] to express this adaptation logic.

Section 2 shows how the software adaptation concern can be – conceptually – considered as an aspect. Section 3 then presents our contribution, SAFRAN, showing how this approach translates in the concrete design and architecture of SAFRAN. We finally illustrate the use of SAFRAN on a simple example (Section 4), and discuss some related work (Section 5) before concluding (Section 6).

2 Software Adaptation as an Aspect

2.1 Adaptation as a Cross-Cutting Concern

In the most general sense, an adaptation is a *modification* triggered by *changing circumstances*, by which a system becomes better suited to its new environment. In the case of software, an adaptation will be implemented by a program responsible for *(i)* observing the environment in which the target software is running to detect new conditions, *(ii)* deciding about the appropriate modifications to apply to the target software, and *(iii)* applying these modifications, adapting the target to the new conditions. With the advent of ubiquitous computing, new applications must be able to adapt themselves autonomously [2] to the various execution contexts in which they can be running. Such *self-adaptive software* applications are both the agent and the target of the adaptation.

The main issue with building such self-adaptive software is that integrating the code dealing with the adaptation concern into the application increases its complexity: the business code becomes "polluted" by non-functional concerns like observing the environment and deciding which reconfiguration is more appropriate. This also impedes the reusability of the system, which can then function

properly only in the few, fixed set of situations which have been anticipated during its development. To solve these issues, we need a looser and more dynamic coupling between business code and adaptation logic.

Software adaptation thus appears as a cross-cutting concern relative to business code, which we would like to modularize so as to offer more reusability and maintainability of the business code. Aspect-Oriented Programming (AOP) [4] gives us adequate abstractions and composition mechanisms to solve these issues.

2.2 Towards an Adaptation Aspect

In "traditional" AOP systems (e.g. AspectJ [7]), an aspect is a module which regroups pairs of the form (*pointcut*, *advice*) where *pointcut* denotes a set of *join-points*, i.e. points of interest in the execution of a base program (in which the aspect is to be weaved) and *advice* is a code fragment to be executed whenever the pointcut matches, i.e. at each of its join-points. Together, these constructs can be used to implement in a well-defined module a concern which can modify the semantics of a base program incrementally and transparently (from the base program's point of view) [8]. The base program and the aspects are *weaved* into a consistent whole, either statically or dynamically. In the following, we propose to "aspectize" the adaptation concern.

The event-based nature of the adaptation process (when a significant change occurs, an adaptation decision is made taken and then applied) relates with the EAOP approach [9] in which point-cuts are defined in terms of sequences of runtime events in the execution of the base program (method invocation, object creation...). In EAOP, runtime events are only *internal*, i.e. related to the base program execution. This is not sufficient to trigger adaptations in the more general setting of context-aware applications, which must also react to *external* events regarding the evolutions of their execution context, like the appearance of a new device or the sudden decrease of the available bandwidth. Despite their different origin (the context instead of the application itself), we believe these events can also be considered as join points, as they trigger adaptation actions. Our join point model thus extends the domain of possible join points beyond internal events ("traditional" join points) to the whole execution context, which increases the expressive power of our system by allowing us to react to changes in the execution context.

Concerning the advice model, actions (triggered by events) indicate how to reconfigure the base program in order to adapt it to the new conditions. The role of the advice language is thus to adjust the target application (tuning, parameterization, architectural configuration...) in order to make it more adapted. Note that contrary to AspectJ [7] which is a general purpose Aspect-Oriented language, the advice language in SAFRAN is a domain-specific language whose expressive power is reduced so that it is not possible to reconfigure the application in an inconsistent state.

As for the aspect weaving model, our choice of considering adaptation in open and dynamic systems lead us to choose a dynamic approach, which is much more

flexible than static weaving because the separation of concerns remains at runtime. This means that the adaptation aspect does not have to be anticipated, but can be loaded, modified and tuned at runtime, without stopping the application. This dynamic weaving process allows us to fully decouple the base program and the adaptation aspect (both spatially and temporally).

3 An Adaptation Aspect in SAFRAN

SAFRAN (Self-Adaptive FRActal compoNents) [10] is an extension to the Fractal component model [5] allowing the creation of self-adaptive applications. One of the key principles in the design of SAFRAN is the treatment of the adaptation concern as an aspect. Following the structure of a generic AOP system, SAFRAN's main elements are:

- a *base program* corresponding to a configuration of Fractal components (architecture);
- *point-cuts* corresponding to the notification of internal events (message invocations on Fractal interfaces, changes in the architecture) or external events (thanks to a framework we designed to create context-aware applications);
- *advices* voluntarily restricted to architectural reconfigurations;
- and finally the adaptation *aspect* itself, linking join points to advices, and represented by modular *adaptation policies* dynamically weaved and unweaved into target components.

The rest of this section will present in more details each of these points.

3.1 Fractal Components: The Base Program

Fractal [5] is a component model developed by France Télécom R&D and INRIA, and distributed through the ObjectWeb consortium. We chose Fractal over other component models because it is designed around a minimal but very extensible core, and is highly dynamic.

A Fractal application (see Fig. 1) is seen as an assembly of components, each made of two parts: a *controller* (in grey on the figure) and its *content*. This content can be either made of other components (*composite*) or of a single object of the underlying programming language (*primitive*). For example, the figure shows a single composite containing two primitive sub-components. The controller part of a component manages all the interactions of its content with the outside. To do this, it exposes internal and external *interfaces* (ports), which can represent services provided or required by a component. Two compatible interfaces can be connected together to create a one-way binding through which all communications must pass. On the figure, the rightmost sub-component provides a service of type "S" through an interface named "s". The other sub-component uses this service through a binding from its own required interface of a compatible type, and exposes another service "m" of a different type. This service is exported to the outside of the composite using a binding from a matching internal interface.

When the composite receives an invocation on its interface "m", its controller intercepts the message, executes optional control behavior (depending on the controller configuration), and then forwards it to the sub-component through the internal binding.

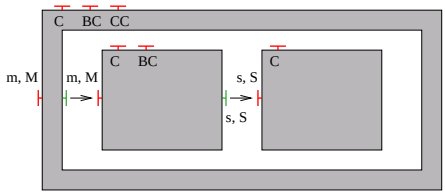

Fig. 1. Example of a simple Fractal architecture.

In addition to *service interfaces*, which depends on each application ("s" and "m" on the figure), Fractal components can offer a variety of standard control interfaces. These interfaces, represented on top of the components in the figure, enable dynamic introspection and modification of various aspects of the components: discovery of the set of interfaces of a component (`component` interface, `C` on the figure), lookup, creation and destruction of bindings (`binding-controller`, or `BC`), addition and removal of sub-components from composites (`content-controller`, or `CC`), etc.

Fractal offers a predefined set of such control interfaces to reflectively manipulate aspects of the components. This support for architectural reflection allows us to reconfigure the architecture of an application during its execution. Compared to other component models like ArchJava [11], which supports runtime reconfigurations only if they have been programmed at compile-time, Fractal's support for reflection enables the discovery and unanticipated modification of the structure of components. This feature is essential for the creation of self-adaptive applications [12] as most of the adaptation we will want to perform are not known during the initial construction of the software.

Another advantage of Fractal is that the set of control interfaces is not fixed. Although there is a predefined set of such interfaces, all of them are optional. More importantly, Fractal and its default implementation are designed so that is is easy to add new control interfaces, thus extending the component model. We use this feature in SAFRAN to seamlessly integrate our extension into the standard model by adding a new control interface named `adaptation-controller` to manage the adaptation aspect associated to a component. Beyond the advantages inherent to the component-based approach, the specific features of Fractal make it an ideal candidate for the construction of *adaptable* applications, the first step towards fully autonomous *self-adaptive* applications.

3.2 Reconfiguration with FScript: The Advice Language

FScript is a domain-specific language [6] we designed to program the Fractal components reconfigurations. FScript is a simple procedural language with dynamic typing and lexical scoping, which gives access to all the standard operations supported by Fractal components: creation of new components, architecture introspection and reconfiguration of this architecture by manipulating composites' content and bindings between interfaces. The main features of FScript are *(i)* a special notation to navigate easily in the Fractal architecture of the base program, and *(ii)* the guarantee that reconfigurations always leave the application in a consistent state. Although it has been designed to be used in SAFRAN, FScript can also be used by itself as a scripting language to program consistent Fractal components reconfigurations.

The FPath Notation FScript uses a special syntax, FPath (inspired by the XPath language [13]), to easily *navigate* in Fractal architectures without modifying it and *select* elements (components, interfaces or configuration attributes) matching certain criteria. The language is based on a model of Fractal architectures as a (virtual) directed graph where nodes represent components, their interfaces and attributes, and where arcs are annotated by *labels* to denote the kind of relation between two nodes ($C1$ "*is a sub-component of*" $C2$, $I1$ "*is bound to*" $I2 \ldots$). In addition to basic expressions (arithmetic, boolean and comparison operators...), FPath expression can denote *relative paths* (starting from an initial node). Such a path is a series of steps, each made of three elements: `axis::test[predicate]`. On each step, an initial set of nodes is converted to a new set by following all the arcs with a label corresponding to the axis, then filtering the result using the *test* (on the node names) and optional *predicates* (boolean FPath expressions applied to each candidate). More precisely, the evaluation algorithm for one step is the following:

P1. [Initialisation] $result \leftarrow \emptyset$.
P2. [Selection] Select every node connected to any of the current ones through an arc whose label matches the `axis` part: $result \leftarrow \cup \{n : c \xrightarrow{axis} n, c \in current\}$.
P3. [Test] If the test part is an identifier (as opposed to `*`), remove from $result$ the nodes whose name do not match: $result \leftarrow \{n \in result : name(n) = test\}$.
P4. [Filtering] Only keep the elements for which all predicates hold: $result \leftarrow \{x \in result : pred_1(x) \wedge \cdots \wedge pred_n(x)\}$.
P5. [End] The algorithm finishes and returns $result$.

For a multi-step path, this algorithm is repeated with the result of the previous step as the current node-set of the next.

FPath offers a set of axes to navigate in Fractal architectures, by selecting a component's interfaces (`interface` axis), configuration attributes (`attribute`),

direct sub-components (`child`) or parents[3] (`parent`), and following the binding of an interface (`binding`). It is also possible to select in one step all the direct and indirect sub-components (resp. parents) of a component with the `descendant` (resp. `ancestor`) axis, which is the transitive closure of `child` (resp. `parent`).

For example, the FPath expression `child::server/attribute::cacheEnabled` first selects all the sub-components of the initial node(s) named `server` (test on the node name), then selects its configuration attribute named `cacheEnabled`. Using the same logic, the expression `count(interface::*[required(.) and not(bound(.))]) > 0` returns *true* if and only if the initial component has required interfaces which are not yet connected (the dot "`.`" in predicates denote the current node to which it is applied).

FScript Actions FScript is used to define *reconfiguration actions*, combining FPath expressions, primitive actions, simple control structures (sequence, choice, finite iteration) and variables manipulation. All the dynamic reconfiguration operations supported by Fractal components are available to FScript program as predefined, primitive actions, including the `attach()` and `detach()` actions introduced by SAFRAN to control the (runtime) weaving of adaptation policies to components. The following example shows an FScript action which could be used to adapt a component.

```
// Changes a cache's replacement strategy.
action select-strategy(cache, strat) = {
  // Gets the cache's client interface to the strategy
  itf := $cache/interface::strategy;
  if (bound($itf)) { // Is it already bound to a server interface?
    // Unbind it and stop the now unused component.
    previous := $itf/binding::*;
    unbind($itf);
    stop($previous/component::*);
  }
  // Binds the cache client interface to the
  // appropriate server interface on $strat.
  bind($itf, $strat/interface::replacement-strategy);
  // Make sure the strategy component is started.
  start($strat);
}
```

This action can be used to change the replacement strategy used by a cache component by modifying the binding between the cache and the strategy component. It uses FPath expressions to navigate in the application's structure, and primitive actions corresponding to operations supported by Fractal components (`bind()`, `stop()`...). Although this action is relatively specific to a given application, FScript can be used to program more generic reconfigurations (replacing a component by another for example) which can then be reused in multiple application (architectural patterns).

[3] Fractal supports component sharing, so a component can have multiple parents.

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

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

3.3 Internal & External Events as Join-Points

We now describe the join-points supported by SAFRAN to trigger the adaptation actions' execution. Following the EAOP approach [9], we consider these join-points as event occurrences. Although traditional join-points only account for the execution of the base program, we extended the domain of events to consider with external events corresponding to changes in the execution context.

Whether they are internal or external, all event occurrences in SAFRAN are represented as objects with a set of properties. Some of these properties are present on every event while some are specific to certain kinds of events. Common properties are: the `type` of the event, as a string; the `source` of the event, which can be either a component or an element of the execution context (see below); and a `timestamp` indicating the time of occurrence of the event.

Event specification and detection is realized by *event descriptors*, for which the exact syntax depend on the type of event, but always follow the same

general form `event-type(parameters)`. Thus, the descriptor `changed(sys://storage/memory#free)` allows to detect the variations in the quantity of memory available on the system.

Internal Events Internal events are execution points in the base program, which in our case is a set of Fractal components. The first three types of internal events, `message-received`, `message-returned` and `message-failed`, correspond respectively to the reception of a message, the successful return of a message and the throwing of an exception. The descriptors for these three kinds of events share the same parameters, expressed using FPath, to indicate which interfaces and methods should be monitored. For example, `message-received($c/interface::logger)` can be used to detect invocations on any method of the `logger` interface of component `$c`, while `message-failed($c/interface::*)` detects errors on any interface of the same component.

The other internal event types correspond to the possible reconfigurations of Fractal components : component creation, life-cycle changes (component started or stopped), configuration (changes in configuration parameters), content manipulation (addition and removal of sub-components) and finally creation and destruction of bindings. Each of the corresponding descriptor takes arguments to specify which components, interfaces or attributes to monitor. Thus, the descriptor `component-started($c/child::*)` detects when any direct sub-component of `$c` is started.

The implementation of these events is based on the instrumentation of Fractal controllers, for example the components' `lifecycle-controller` is instrumented to generate `component-{started,stopped}` events.

External Events In order to detect the occurrence of external events we first need to reify the application's execution context, which is normally implicit. To do this, we use WildCAT [14], a system we designed to ease the creation of *context-aware* applications [1]. WildCAT is used by SAFRAN to observe the execution context and to notify the occurrence of the external events which can trigger the execution of reconfigurations. As was the case for FScript, WildCAT can actually be used independently.

WildCAT models the execution context as a set of *context domains*, each representing a particular aspect of the context, for example hardware resources, network, geo-physical information, etc. Each of these context domains is itself modeled as a tree of *resources* described by a set of attributes (simple $(name, value)$ pairs). The syntax used to denote resources and attributes is inspired by that of URIs: `domain://path/to/resource#attribute` (`#attribute` being optional). For example, `sys://storage/drives/hdc#removable` indicates whether the `hdc` drive is removable.

The context model provided by WildCAT changes dynamically to reflect changes in the actual execution context: attributes values can change, attributes and resources can appear or disappear at any moment. All these modifications

generate *external events* which can be detected by an adaptation policy. The different types of external events supported by SAFRAN are:

`changed(expression)` : detects any modification of the value of the expression, which can reference any attribute or resource in the context[4], for example `changed(geo://location/logical#room)`. Expressions to monitor are written in a simple language which, in addition to references to context locations, supports strings, numbers, arithmetic and boolean operations, comparisons and function calls.

`realized(condition)` : detects the occurrence of a boolean condition, for example `realized(sys://storage/memory#free > 2*sys://storage/swap#used)`. This is actually a particular case of `changed` which only detects changes from *false* to *true*.

`appears(path)` **and** `disappears(path)` : detects the appearance or disappearance of a resource or attribute in the context. The path expression can be a *joker* character "`*`" as its last element. For example `appears(sys://devices/input/*)` detects the apparition of any new input device.

3.4 Adaptation Policies: The Adaptation Aspect Language

Adaptation Aspect Syntax Conforming to the reactive nature of the adaptation process, adaptation policies in SAFRAN are structured as sets of *reactive rules* of the form

```
when <event> if <condition> do <action>
```

where `<event>` is an (internal or external) event descriptor[5] (cf. Sect. 3.3) corresponding to a point-cut, `<condition>` is a boolean FPath expression (without side-effects), and `<action>` is an FScript reconfiguration (cf. Sect. 3.2) corresponding to the aspect's advice.

This type of rules is inspired by what can be found in Active Databases [15] under the name of ECA (Event, Condition, Action) rules. An adaptation rule indicates that *when* an event corresponding to the `<event>` expression occurs, *if* the `<condition>` expression holds, *then* the `<action>` reconfiguration is applied, thus adapting the target application to the new conditions resulting from the event.

In the SAFRAN system, the adaptation policies which are dynamically attached to Fractal components are made of (ordered) sequences of adaptation rules:

```
policy example = {
  rule { when <event1> if <cond1> do <action1> }
  rule { when <event2> if <cond2> do <action2> }
```

[4] WildCAT automatically re-evaluates expressions when any element it depends on changes.

[5] In the future, we plan to extend this model to support more complex point-cuts, especially hybrid point-cuts which mix internal and external events and would allow finer coordination between the execution of adaptation code and the base program.

```
  ...
}
```

As an adaptation policy is always executed when attached to a target component, a special variable named `$target` can be used inside rules to access the component to which the policy is attached; it is akin to `self` of `this` in object-oriented languages.

Figure 2 summarizes the event/control flow between the different parts of SAFRAN. Internal events are generated by instrumentation code inside Fractal components, and external events are detected by WildCAT. These events are routed to the appropriate adaptation controllers, which uses its current rules to decide which adaptations to perform. These decisions are finally applied by executing FScript reconfigurations.

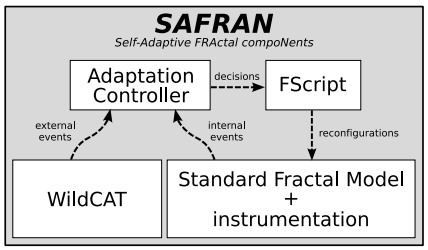

Fig. 2. Flow of events in SAFRAN.

Weaving The Adaptation Aspect SAFRAN introduces an extension to the Fractal model which enables the dynamic attachment (weaving) of adaptation policies (aspects) to components (base program). Like most Fractal extensions, it takes the form of a new control interface, in this case `adaptation-controller`. It is this controller, present on each self-adaptive component, which implements the weaving of adaptation policies into the target component, thus making it *self-adaptive*: whereas a standard Fractal component can *be adapted* by an external entity (through its standard control interfaces), a SAFRAN component embeds the adaptation code itself and becomes *autonomous*, actor of its own adaptation.

The `AdaptationController` interface (see below) enables the dynamic attachment (weaving) of one or several adaptation policies to each SAFRAN component. This interface can be seen as a special case of an aspect weaving interface, where `attachFcPolicy()` and `detachFcPolicy()` correspond to specialized versions of more general `weave(Aspect)` and `unweave(Aspect)` operations:

```
public interface AdaptationController {
  void attachFcPolicy(AdaptationPolicy policy);
  void detachFcPolicy(AdaptationPolicy policy);
  AdaptationPolicy[] getFcPolicies();
}
```

When a policy is attached to a component, the component's adaptation controller analyzes it, and depending on the join-points mentioned in the rules, instruments the target component to generate the appropriate internal events and registers itself with WildCAT to be notified of the external events. After this initialization, when the adaptation controller receives events, be they internal or external, it determines the appropriate reaction according to the current set of policies and rules on the target component (see below), and then executes this reaction in order to adapt the component to the new circumstances. This execution schema matches the reactive nature of the adaptation process, with the same three phases: observation, decision, action.

Aspect Composition Model To handle multiple advices affecting the same join-point, SAFRAN provides an ad hoc aspect composition model. Indeed, a policy (aspect) can be made of several rules, a component can have multiple policies attached at the same time, and of course an application can contain many self-adaptive components. SAFRAN defines the following composition rules to manage the interactions between these different elements when several rules are triggered by the same event:

- Inside a given policy, the rules' reactions are composed in sequence, in the textual order of their definition, and executed in a single reconfiguration transaction. The rationale is that a given policy should implement a consistent, self-contained adaptation, and its (single) author can be expected to foresee the rules' interactions.
- On a single component, the competing reactions of multiple policies are also executed in sequence, but each in its own reconfiguration transaction. The effects of a single policy's failure is thus isolated. This is important as policies developped independently can be attached to the same component. The order in which the policies' reactions are executed depend in the order of their attachment: the oldest policies are executed first. The rationale is that once a policy P is attached to component C, the resulting component C' must be considered as a self-contained black-box by the next policies, and hence P has a greater priority over the policies attached later.
- Finally, when multiple components must react to a single event, their reactions are executed in an order defined by the components' composition relations: subcomponents are adapted before their parents. The rationale is similar to the previous one: in a component-base approach, when a composite includes a subcomponent, it should treat it as a black-box.

Although these rules are designed to be the most general possible, there are situations in which they are not appropriate. One of the main future directions of our work is the extend the execution model of our reactive rules to provide more flexibility on the semantics of composition. The challenge is to do this while without making the policies language too complex for the end users.

4 Example

The example application we chose to illustrate the use of SAFRAN is a small web server named Comanche, implemented by É. Bruneton as a tutorial on the use of Fractal. Comanche, being extremely simple, does not integrate a file cache mechanism. In order to improve its performances, we thus add a new cache component in Comanche. The cache performances depends on the amount of memory it can use. If this amount is too low, the system will not use all the cache potential. If it is too high, performances can be even lower, as the cache will force the operating system to use slow virtual memory (swap). The amount of memory we should allocate to the cache depends on the amount of free memory available on the host system, which varies dynamically and unpredictably. Our adaptation policy will thus have to dynamically adapt the maximum amount of memory allocated to the cache component in order to guarantee good performances in every circumstances. The introduction of a cache component in Comanche is very simple, as it only requires to modify the application architecture defined using Fractal's ADL (Architecture Description Language), after having coded the cache component itself, of course.

The cache component exposes two parameters accessible through its `attribute-controller` interface, `currentSize` and `maximumSize`, indicating respectively the current and maximum amount of memory the cache uses; only `maximumSize` is writable. The policy works by adjusting the value of `maximumSize` depending on the amount of free memory on the host system, which WildCAT makes available as `sys://storage/memory#free`. We now have all the information we need to write the adaptation policy:

```
policy adaptive-cache = {
  rule {
    when realized(sys://storage/memory#free < 10*1024)
    do { to-free := 10*1024 - sys://storage/memory#free;
         size := $target/cache/attribute::currentSize - $to-free;
         if ($size < 500) {
           set-value($target/cache/attribute::maximumSize, 0);
           disable-cache($target);
         } else {
           set-value($target/cache/attribute::maximumSize, $size);
         }
       } }
  rule {
    when mem:changed(sys://storage/memory#free)
    if (sys://storage/memory#free >= 10*1024)
    do { enable-cache($target);
         current := $target/cache/attribute::currentSize;
         size := 0.8 * ($mem.new-value + $current);
         max := sys://storage/memory#used - $current + $size;
         if ($max < sys://storage/memory@total - 10*1024) {
           set-value($target/cache/attribute::maximumSize, $size);
         }
       } } }
```

This file uses two user-defined FScript actions (code not shown for space reasons): the first one, `disable-cache`, disables the cache component by disconnecting it while the second action, `enable-cache`, re-introduces it in the components' pipeline. The first rule is triggered when the total amount of available memory drops below 10Mb. When this happens, the reconfiguration action

tries to free memory by reducing the size of the cache, or even disabling it completely below a certain size. The second rule is triggered whenever the amount of memory changes[6] but is more than 10 Mb. In this case, the reconfiguration adjusts the maximum cache size to use 80% of the total amount available, but only if this leaves enough free memory to the rest of the system.

This example policy illustrates *(i)* a point-cut based on two types of external events (`realized` and `changed`); *(ii)* two kinds of reconfiguration actions: parameterization and bindings manipulation. Not only the reconfiguration is dynamic, but thanks to the dynamic weaving process in SAFRAN, the policy can be updated during the execution of the base application, which is essential when developing open systems.

5 Related Work

In the last few years, numerous works have tried to make software more adaptable, in particular to take into account the needs of mobile computing and autonomous applications [2]. The most promising approach seems to be the use of dynamic and extensible component models, which enable the integration of non-functional services in a way that is adapted to the specific needs of applications, and most importantly allow dynamic reconfigurations of the application itself [16]. Some works, like ACEEL [17] or K-Components [3] are based on custom component model which impose a specific way of structuring applications. Others use existing component models but restrict themselves to particular application domains: for example PLASMA [18] which is based on Fractal like SAFRAN but limited to multimedia stream processing.

Concerning the adaptation aspect itself, Cilia *et al.* [19] have shown the links existing between AOP and reactive rules from active databases, particularly in the context of autonomous applications. Indeed, applications must be reactive in order to adapt themselves to their context, and the underlying principles of AOP allow us to introduce this reactivity in base programs in a non-invasive way. However, the authors only present abstract concepts where SAFRAN provides a concrete implementation.

We can also note the existence of FAC [20] and Fractal-AOP [21], two extensions of the Fractal model for general AOP. Although SAFRAN is heavily inspired by AOP, SAFRAN's goal is to enable the creation of self-adaptive applications, and AOP is simply a convenient framework used to structure and describe the system. The difference between the FAC/Fractal-AOP approach and SAFRAN' approach is essentially the same as between a general-purpose programming language, powerful but generic, and a DSL, more limited but better suited to its particular objective.

[6] In practice, such an event is not generated each time the amount of free memory changes, but only when such a change is detected. The sampling rate and hence the system performance depends on how the corresponding sensor is configured.

6 Conclusion And Future Works

In this paper, we have shown how AOP principles can be used to ease the creation of self-adaptive applications. On a conceptual level, we have shown that adaptation can be considered as a cross-cutting concern and that it is possible to use AOP's concepts (base program, point-cuts, advices and weaver) in this particular case to model the adaptation aspect. In order to support self-adaptive applications, we have *extended* the traditional notion of join-points beyond internal events related to the program's execution to include external events corresponding to changes in the execution context. Regarding the advices, we have on the contrary chosen to *restrict* the expressive power of our reconfiguration actions by designing a Domain-Specific Language (FScript) which can offer guarantees on the consistence of adaptations.

On a more concrete level, we have then described SAFRAN, an extension of the Fractal model which implements this approach and enables the modular development of reactive adaptation policies. The main features of SAFRAN are *(i)* the decoupling of adaptation policies from business components, *(ii)* a Domain-Specific Language based on reactive rules to express these policies, and *(iii)* a completely dynamic approach, where policies and reconfiguration actions – even ones which where not anticipated at compile-time – can be defined, loaded and applied during the execution of the target application without stopping it. Another interesting feature of SAFRAN is its modular design, with subsystems (WildCAT and FScript) which can be reused independently.

One of our future goals is to extend the principles of SAFRAN to allow the adaptation of distributed applications. We do not anticipate major structural changes in the system, but incremental evolutions of its different parts. A first step would be to extend FScript to support distribution-aware reconfigurations, like for example component migration and distributed bindings. New WildCAT context domains will have to be implemented to share information between remote nodes; different strategies are possible with varying degrees of invasiveness (see [14]). Finally, the execution model of adaptation policies itself will have to be extended to support coordinated adaptation of remote components.

References

1. Dey, A.K., Abowd, G.D.: Towards a better understanding of context and context-awareness. In: Workshop on The What, Who, Where, When, and How of Context-Awareness, as part of CHI 2000, The Hague, The Netherlands (2000)
2. Kephart, J.: A vision of autonomic computing. In Gabriel, R.P., ed.: Onward! proceedings from an OOPSLA 2002 track, Seattle, WA, USA, ACM (2002) 13–36
3. Dowling, J., Cahill, V.: The K-Component architecture meta-model for self-adaptive software. In: Proceedings of Reflection 2001, The Third International Conference on Metalevel Architectures and Separation of Crosscutting Concerns. Volume 2192 of LNCS., Springer-Verlag (2001) 81–88
4. Kiczales, G., Lamping, J., Mendhekar, A., Maeda, C., Lopes, C.V., Loingtier, J.M., Irwin, J.: Aspect-oriented programming. In: European Conference on Object-Oriented Programming (ECOOP). Volume 1241 of LNCS., Springer-Verlag (1997)

5. Bruneton, E., Coupaye, T., Leclercq, M., Quema, V., Stefani, J.B.: An open component model and its support in java. In: Proceedings of the 7th International Symposium on Component-Based Software Engineering (CBSE 2004). Volume 3054 of LNCS., Edinburgh, Scotland, Springer-Verlag (2004) 7–22
6. van Deursen, A., Klint, P., Visser, J.: Domain-specific languages: An annotated bibliography. ACM SIGPLAN Notices **35**(6) (2000) 26–36
7. Kiczales, G., Hilsdale, E., Hugunin, J., Kersten, M., Palm, J., Griswold, W.G.: An overview of AspectJ. In Knudsen, J.L., ed.: ECOOP 2001. Volume 2072 of LNCS., Springer-Verlag (2001) 327–353
8. Filman, R.E., Friedman, D.P.: Aspect-oriented programming is quantification and obliviousness. In: Workshop on Advanced Separation of Concerns. (2000) Minneapolis.
9. Douence, R., Fradet, P., Südholt, M.: A framework for the detection and resolution of aspect interactions. In: Generative Programming and Component Engineering GPCE 2002. Volume 2487 of LNCS., Pittsburgh, PA, USA, Springer-Verlag (2002) 173–188
10. David, P.C.: Développement de composants Fractal adaptatifs : un langage dédié à l'aspect d'adaptation. PhD thesis, Université de Nantes / École des Mines de Nantes (2005)
11. Aldrich, J., Chambers, C., Notkin, D.: Architectural reasoning in ArchJava. In: Proceedings of ECOOP'2002, Malaga, Spain, AITO (2002)
12. Redmond, B., Cahill, V.: Supporting unanticipated dynamic adaptation of application behaviour. In: Proceedings of ECOOP 2002. Volume 2374 of LNCS., Malaga, Spain, Springer-Verlag (2002) 205–230
13. World Wide Web Consortium: XML path language (XPath) version 1.0. W3C Recommendation (1999) `http://www.w3.org/TR/xpath`.
14. David, P.C., Ledoux, T.: WildCAT: a generic framework for context-aware applications. In: Proceeding of MPAC'05, the 3rd International Workshop on Middleware for Pervasive and Ad-Hoc Computing, Grenoble, France (2005)
15. Dittrich, K.R., Gatziu, S., Geppert, A.: The active database management system manifesto: A rulebase of a ADBMS features. In: International Workshop on Rules in Database Systems. Volume 985., Springer-Verlag (1995) 3–20
16. McKinley, P.K., Sadjadi, S.M., Kasten, E.P., Cheng, B.H.: Composing adaptive software. IEEE Computer **37**(7) (2004) 56–64
17. Chefrour, D., André, F.: Développement d'applications en environnements mobiles à l'aide du modèle de composant adaptatif ACEEL. In: LMO 2003, Vannes, Hermès (2003)
18. Layaïda, O., Hagimont, D.: Designing self-adaptive multimedia applications through hierarchical reconfiguration. In: Distributed Applications and Interoperable Systems (DAIS). Volume 3543 of LNCS., Athens, Greece, Springer-Verlag (2005) 95–
19. Cilia, M., Haupt, M., Mezini, M., Buchmann, A.: The convergence of AOP and active databases: Towards reactive middleware. In: Proceedings of GPCE'03. Volume 2830 of LNCS., Erfurt, Germany, Springer-Verlag (2003) 169–188
20. Pessemier, N., Seinturier, L.: Components, ADL & AOP: Towards a common approach. In: Reflection, AOP and Meta-Data for Software Evolution Workshop at ECOOP 2004 (RAM-SE'04), Oslo, Norway (2004)
21. Fakih, H., Bouraqadi, N.: Les aspects et les composants logiciels : Etude de cas avec le modèle de composant Fractal. Numéro spécial de la revue L'Objet sur les aspects **11**(3) (2005) 1–17 In French.