Flexible Reactive Capabilities in Component-Based Autonomic Systems

Jayaprakash Nagapraveen*
HADAS Group, LIG
Saint Martin d'Hères, France
nagapraveen.jayaprakash@imag.fr

Thierry Coupaye
France Telecom R&D
Grenoble, France
thierry.coupaye@orange-ftgroup.com

Christine Collet
INP Grenoble
LIG Laboratory
Saint Martin d'Hères, France
Christine.Collet@imag.fr

Pierre-Charles David†
OBASCO Group, EMN/INRIA
Nantes, France
pcdavid@gmail.com

ABSTRACT

Reactive behaviour, the ability to (r)eact automatically to take corrective actions in response to the occurrence of situations of interest (events) is a key feature in autonomic computing. In active database systems, this behaviour is incorporated by Event-Condition-Action (ECA or active) rules. Our approach consists in defining a mechanism for the integration of these rules in component-based systems to augment them with autonomic properties. The contribution of this article is twofold. First, we propose a rule model, i.e. a rule definition model and a rule execution model, that can be coherently integrated into a component model. Second, we propose a graceful architecture for the integration of active rules into component-based systems in which the rules as well as their semantics (execution model, behaviour) are represented/implemented as components, which permits i) to construct personalized rule-based systems and ii) to modify dynamically the rules and their semantics in the same manner as the underlying component-based system by means of configuration and reconfiguration. These foundations form the basis of a framework/toolkit which can be seen as a library of components to construct events, conditions, actions, rules and policies (and their execution subcomponents). The framework implementation is extensible: additional components can be added at will to the library to render more elaborate and more specific semantics according to certain applicative requirements.

*This work was done while the author was a PhD student at France Telecom R&D and IMAG-LSR

†This work has been done while the author was a postdoctoral fellow at France Telecom R&D.

Categories and Subject Descriptors

D.2.11 [**Software Engineering**]: Software Architecture; D.2.13 [**Software Engineering**]: Reusable Software— ; H.2 [**Database Management**]: Systems

General Terms

Design, Experimentation

Keywords

autonomic systems, component-based architectures, ECA / active rules

1. INTRODUCTION

The overall motivation which underlies the emergence of autonomic computing[9] is based, from an IT industrial point of view, on the observation that the costs related to the software infrastructures (TCO) currently move from costs related to the development and licensing to costs related to the deployment and exploitation. It appears also quite clearly that a "manual administration" of pervasive environments such as "machine to machine" (automotive, home networking, etc.) software infrastructures or grid computing is, in practice, almost impossible. Autonomic computing thus aims basically at, as much as possible, automating the deployment and management (administration) of software systems in order to decrease human interventions and associated costs.

We consider in this work that an autonomic system is composed of an *autonomic infrastructure* superimposed on a *target component-based system*. The upper layer or the autonomic infrastructure, is responsible for implementing a *control loop*, i.e. instrumenting the components of the target system for monitoring, detecting and notifying events, diagnosing the system based on these events, and making decisions to determine what and how corrective actions need to be executed, and finally, executing the corrective actions on the components of the target system.

Our work focuses on the architecture and the behaviour of

control loops. We propose to use Event-Condition-Action (ECA or active) rules[12], a mechanism widely used in active database systems to provide a reactive behaviour (an elaborated form of *triggers* as found in most commercial DBMS). The fundamental idea is to "extract" the reactive functionality of active database systems[8], and to "adapt and inject" it in component-based systems so as to provide them with autonomic capabilities.

Our objective is to realise a modular and extensible framework/toolkit for the construction of ECA rule-based autonomic architectures. In this framework, the rules as well as their semantics (execution model, behaviour) are represented/implemented as components, which permits i) to construct personalized rule-based systems and ii) to modify dynamically the rules and their semantics in the same manner as the underlying component-based system by means of configuration and reconfiguration. In its basic version, the framework is a library formed of basic components (and subcomponents) which permits to construct basic rules. The framework is extensible, i.e, additional components can be added at will to the library to render more elaborate and more specific semantics according to certain applicative requirements.

The main goal of this article is to introduce the design of this framework (called Fractal ECA) through an illustrative example used throughout the paper. Due to space constraints, only the elements of the framework required for the comprehension and implementation of the example are discussed.

Sample Autonomic Scenario. Let us consider as a target system, a minimal HTTP webserver with the sole functionality of retrieving HTML documents. It is made of two main components, namely a *Request Receiver* and a *Request Processor*. The request receiver component or front-end uses a lower-level scheduler component that creates a new activity (a thread in this case) for each request. The request processor component or back-end analyses the request, logs it and responds to it. Our Comanche web server[1](shown in the bottom of Figure 4), is a multi-threaded system in Java that follows a prefork model - the parent process forks new child thread for every request. Most of the action takes place in the child threads - figuring out what the requests mean and sending the requested content back to the client. Once the processing is done, the child thread waits for the next one from its parent. Multithreading avoids the server from being idle until an I/O operation is finished. On the other hand, it introduces an overhead on the CPU due to the creation of new threads and to the commutation between existing threads. The right trade-off between these two conflicting factors reflects on the system performance. We propose a simple active rule to maintain this compromise (in complete informal format):

RULE ***DoubleThreadPoolSize***
ON each webpage request
IF (no free threads in the thread pool)
DO { Double the thread pool size }

[1]which comes with the Fractal distribution. See http://fractal.objectweb.org.

The above rule's purpose is to extend the capacity of the webserver by increasing the size of the thread pool, that is the number of possible concurrent child threads. A complementary rule can be equally envisaged that reduces the size of the thread pool, when the number of passive threads are high, thus liberating unused ressources.

The overall semantics of an ECA rule is the following: *when an event of type E occurs, if condition C is satisfied then execute action A*. Behind this apparent simplicity hides a great deal of complexity that raises many questions which we shall illustrate based upon our concrete example. Indeed, several questions are raised by the execution of this simple rule, to list a few:

- Is a new instance of a rule triggered for every occurrence of a triggering event (webpage request)? or once for several occurrences of the triggering event?
- When are the condition and action parts executed with respect to the target system execution? before the execution of the triggering operation or after?
- Is the rule executed in the same execution thread as the triggering operation or in a separate one?

For the above questions, several options exist, each representing an execution strategy of the rule. In the sequel, we discuss possible answers to these questions based on our illustrative scenario. More generally, our core work is to provide the rule programmer with an abstract framework for reasoning about rules execution and an actual architectural framework for practically programming rules and their semantics.

The rest of the paper is organized as follows. Section 2 and 3 introduce our first contribution: the proposition of an ECA rule model suited for component-based systems made of a rule definition model (Section 2) together with a rule execution model (Section 3). Section 4 and 5 introduce our second contribution: the proposal of an architectural design that allows for the graceful integration of active rules into component based-systems (Section 4) and some elements about the actual implementation of rules as Fractal components (Section 5). Section 6 discusses related works. Section 7 concludes the article.

2. RULE DEFINITION MODEL

A definition (or knowledge) model specifies how the rules are represented and manipulated. This section describes the rule definition model we propose for component-based autonomic systems.

Event. An event is a happening of interest at a given point in time. It is characterized by an event type, i.e, an expression describing a class of significant occurrences of interest. In our framework, we consider the following event types: i) *applicative* - corresponds to inter-component interactions, ii) *structural* - represents modifications (reconfigurations) of the structure (topology) of the system, like adding or removing a component, or creating new bindings between

components, iii) *system-level* - characterises events coming from the external or underlying environment or context of execution (e.g. JVM and OS events). In our illustrative scenario, we have an applicative event type: an operation (method) invocation on a component interface, and more precisely calls on the interface of the frontend component of our web server to request web pages.

Condition. Conditions are optional and express additional constraints on the state of the system that must be satisfied for the action part to be executed. For example, in our autonomic scenario, the condition predicate checks the number of available threads, These kinds of condition expressions (e.g. whether an attribute's value is bigger than a particular value or not) are simple boolean expressions built using logical operators. More complex expressions can be formed based on queries on the structure of the system as well as on its behaviour. A query that selects the various components linked to a particular one is one such example. In such a case, the condition is considered to be true when the query returns a non-empty result.

Action. The corrective (re)actions that the target system can be subjected to are expressed in the action part of the rule. The event and condition parts of the rule serve to analyse the symptoms affecting the system. In our scenario, a method call to the Request Receiver component triggers the rule. Its condition part evaluates if the number of free threads is below a limit. If yes, the action that increases twofold the size of the thread pool is performed. To rectify the anomalies, the action can range from simple parameterizations of component attributes, for example, an increase in the size of a cache or pool, to complex structural reconfiguration operations, which can include addition, removal or replacement of one or several components. Other types of actions can be envisaged, like external notifications, for example, an email or SMS notification to an administrator.

3. RULE EXECUTION MODEL

A rule execution model specifies the behavioural semantics of rules. This section introduces the design of our proposed execution model and discusses the main dimensions of this model on our illustrative scenario.

The entire execution of a single rule is comprised of the following three phases and various states:

1. Triggering and Event Processing Phase R(E): this phase begins with the notification of the event(s) that triggers ("wakes up") the rule. The notification is performed by the entity on which the event occurs. It consists in processing the event(s) based on the various rule execution parameters. The rule goes from the *triggerable* state to the *triggered* state.
2. Condition Evaluation Phase R(C): the second phase of the execution evaluates the condition expression. If the condition is satisfied then the rule transits from the *evaluable* state to *evaluated* state.
3. Action Execution Phase R(A): the last phase of the rule execution corresponds to the execution of the action part of the rule. It takes the rule from the *executable* state to the final state of *executed* state (generally confonded with the initial *triggerable* state), thus inducing a positive feedback change in the system behaviour.

It is worth mentioning that the condition of a triggered rule is not always evaluated immediately (hence the two separate states *triggered* and *evaluable*), and that a triggered rule with a satisfied condition is not always executed immediately (hence the two separate states *evaluated* and *executable*). When and how (e.g. which activity/thread) a rule is processed depends on the various dimensions of the rule execution model. Some of the most important ones are discussed later in the context of our illustrative scenario. Of course, when multiple rules are concerned, which is the case in real autonomic systems, an execution model also specifies when and how rules triggered simultaneously (by same or different events) (a.k.a. *multiple rules*)and rules triggered by other rules (a.k.a. *cascading rules*) are executed. This is handled by *rule execution strategies* (or *policies*) which basically specify the scheduling of rules (e.g. depth-first order, width-first order, flat order, by cycles in sequential or parallel settings). Due to space limitation, this article does not detail these aspects. The reader may refer to [4]. Prior to that and more fundamentally, if rules have an execution model of their own, it has to be stated that the introduction of active rules in a system (be it a database or a component-based system) has also a non negligible impact on the behaviour of that system. Indeed, there exists a dependency between the execution of the system and the execution of rules, for it is the former that triggers the latter and also the two executions are interwoven/intertwined together.

3.1 From active database systems to active component based systems

Active rule execution models in database systems have been extensively studied but cannot be directly applied to component-based systems. First, events that trigger a rule in an active DBMS are query (SQL) statements on a global data schema, so are the condition and action parts. But this it is not the case in component-based systems where we have a variety of events, condition predicates and actions (as defined in the rule knowledge model). In active database systems, all rule operations are performed on a single database, whereas in a component-based system, they may have to be performed on different components. Indeed, in a component-based system, situations of interest can happen on any component of the system. To gain a thorough understanding of the component and its execution environment, we might have to perform additional queries on it or on its neighbours and finally execute the corrective actions elsewhere. So, the distributed characteristic of component-based systems is one of the distinguishing factors.

Execution units and execution points in component-based execution models. Finally, besides the two differences we have just mentioned, a key difference between active database systems and active component-based systems is that execution models in active database systems are

based on a central concept, that of *transaction*, which is (generally) inexistant in component-based systems. A transaction in database systems is a sequence of operations that constitutes a unit of concurrency and recovery thanks to the well-known "ACID properties" (*Atomicity, Consistency, Isolation and Durability*). Transaction is a core and foundational concept of active database systems because, thanks to transaction *demarcations* (*start, commit, abort/rollback*), they provide a natural and convenient execution unit for the execution of active rules. An execution unit specifies an interval (between two execution points in a sequential flow or basically between two points in time) during which events can be detected/notified to interested rules and rules can be evaluated and executed. Hence, an execution unit specifies the *granularity* of rules execution. On the one hand, component-based systems generally do not consider transactions. On the other hand, the behaviour of a component-based system generally refers to interaction through interfaces only, thanks to operation (methods in Java) invocation. Hence, we define the execution unit in component-based systems as delimited by the interval between the reception of an operation invocation on a server interface and the emission of a response onto a client interface. For method invocations on a component's functional interfaces (produces applicative events), and operations that modify the structure of the system (produces structural events), we may signal two events: *begin* and *end*. Other forms of events (e.g. system events) can be integrated in the model that by considering their begin and end events are merged (i.e. they both represent the same execution point or point in time).

3.2 Rule execution dimensions

Based on these hypotheses, we consider our approach for defining a rule execution model, similar to the one followed in active database systems[4], where it is defined as a set of *dimensions*, with each dimension being attributed a particular *value*. The differences/issues outlined above, have been addressed in the form of a flexible rule execution model for component-based systems, adapted from rule execution models defined in active DBMS. The sequel discusses rule execution dimensions in the context of the autonomic scenario introduced earlier.

Event Processing Mode. On every webpage request, the Comanche webserver requests the scheduler service for an execution thread. So, if an instance of the thread management rule is triggered for every call, then the system ressources would be spent unnecessary resulting in a lower performance. Ideally, a rule needs to be triggered once at the appropriate moment to rectify the situation. The event processing dimension addresses this issue, with the possibility of triggering a rule for several occurrences of the event type. A rule may handle either only one event at a time or a set of events. This is specified in the *event processing mode*, having an *instance-oriented* semantics for the former, and a *set-oriented* semantics for the latter. In other words, an instance-oriented semantics suggests that a rule will be triggered for every occurrence of a triggering event. Such a kind of event processing strategy is interesting whenever each event has to be treated individually, e.g. when an exception is raised, or on every attack or on every forced entry by a malicious user which requires preventive measures to be triggered in the form of a rule. If several rules are triggered with the same purpose, system resources are bound to get depleted, affecting the performance of the application. Therefore, this strategy is beneficial when a single execution of a rule is enough to resolve the anomaly in the system. We shall opt for a set-oriented value to the rule's event processing mode: as said earlier, a single execution of the rule is sufficient to retain the performance level of the system.

Coupling Modes. Once a rule is triggered, we have to determine when and how it will continue its execution for it should not affect the system's execution. If we consider our thread management rule, should our rule be executed before processing the request or after? Should it be done in the same execution thread or in a separate one? Several options exist for the above. This is taken care of by the *coupling mode* dimension characterized by the couple : < *execution mode, activity mode*>.

The *execution mode* specifies when the condition and action parts of a rule are evaluated and executed with respect to the execution of the triggering operation. The triggering operation is the method invocation on a component's interface which produces events that trigger a rule. Such a rule that is activated and is ready for execution, is called a triggered rule. The commonly supported execution modes include (cf. Figure 1):

- *immediate (or as soon as possible - ASAP):* the triggered rule is processed immediately - its condition is evaluated, if true, the action is executed without any pause.
- *defered:* the triggered rule is processed later, awaiting the end of the triggering operation. The rule is triggered when an event indicating the beginning of the triggering operation is received. But the condition evaluation and action execution of the rule are processed only on receiving an event indicating the end of the triggering operation.
- *delayed:* here, the condition is evaluated immediately after the rule has been triggered, but the action part is executed only on the completion of the triggering operation.

Rules dealing with security issues have generally a high priority, and may typically take the *immediate* value in order to execute instantly before the damage is done. *Defered* rules are typically used in situations in which the action has to be executed on the final state of the system. In this respect, immediate rules might embed pre-conditions, while defered rules might embed post-conditions. *Delayed* rules are intermediary with a condition evaluated at the beginning of the triggering operation (i.e. before the state of the system might be changed by other rules for instance) and the action executed at its end. Our rule concerns the performance of the system. It permits the system to take preventive measures in order to maintain its performance levels. So, it is not so crucial, and can be executed once the initial task is completed, a defered value will be attributed to our rule's execution mode.

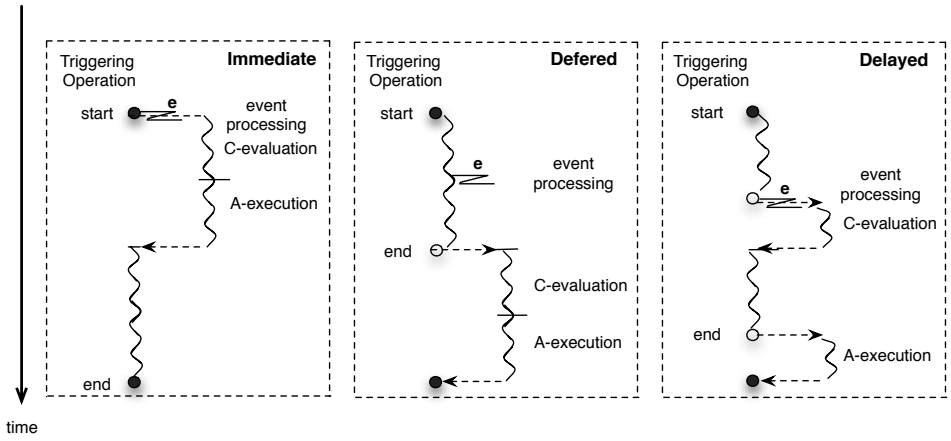

Figure 1: Execution Mode

An execution thread represents a sequential flow of control. The *activity mode* dimension determines whether the triggered rule is executed in the *same* thread as the triggering operation or in a *separate* thread. It is recommended to create a separate flow of control for aspects related to the administration of the system, e.g, logging. Thus, separating the system under control and its administration, so that its normal execution is not too disturbed. On the other hand, some administration scenarios might require the system to be paused, to enable some modification, and to resume later its execution. Depending upon the scenarios, the best method of executing the rule has to be judged and employed. For our thread management rule, we shall follow the conventional choice of executing the rule in a separate thread because there is no risk due to concurrency on the usage of threads in the pool since the rule creates only new threads.

Focus on interactions between a couple of dimensions. Since there exist some dependancies and intricacies among these dimensions, the individual semantics of the dimensions might slightly differ which one could guess at a first glance when considered as a whole. To illustrate such intricacies, we now focus on the interactions between event processing and execution modes.

Prior to presenting the possible combinations for the couple $<$ event processing mode, execution mode $>$ (including the ones that match the above choice in our example) , the following notations are employed in the sequel and in Figure 2:

- b_n: the event corresponding to the beginning of the n_{th} method invocation
- e_n: the event corresponding to the end of the n_{th} operation (method) invocation
- $R_n(E)$: depicts the event processing phase of the n_{th} occurrence of the considered triggered rule R
- $R_n(C)$: depicts the condition evaluation phase of the n_{th} occurrence of the considered triggered rule R
- $R_n(A)$: depicts the action execution phase of the n_{th} occurrence of the considered triggered rule

The events b_n and e_n are representative of the n_{th} occurrence of a triggering event type.

1. $<$ Immediate, Instance $>$: on every event b_n, a rule is triggered and processed immediately in its entirety. All e_n events are ignored.
2. $<$ Immediate, Set $>$: on the event indicating the beginning of the first triggering operation, i.e, on b_1, a rule is triggered and continues processing till its completion. All b_is that occur till the triggered rule completes evaluation, are consumed by the rule, i.e, the triggering operations are taken into consideration by the rule in execution. The next b_i when the rule is in $R_n(A)$ or after, triggers the next rule. Similarly, this rule also consumes all b_i that occurs till it completes evaluation. Two similar rules can coexist when one is in the action execution phase and the other begins execution.
3. $<$ Defered, Instance $>$: on every b_n, a rule is triggered, it processes the event and waits for a complementary e_n event to continue evaluating - $R_n(C)$ and complete execution - $R_n(A)$.
4. $<$ Defered, Set $>$: all b_is trigger each a rule. The rules complete the $R_n(E)$ phase, and wait for an end event e_i. When a e_i event is received, all rules triggered after R_i are discarded and their corresponding triggering operations consumed by the rule R_i. All events e_j where j $>$ i received later, are discarded. For any e_k where k $<$ i, the corresponding rule R_k resumes execution, and consumes all triggering operations that have triggered rules R_j where k $<$ j $<$ i.
5. $<$ Delayed, Instance $>$: on every b_n, a rule is triggered, processes its event part - $R_n(E)$, evaluates its condition part $R_n(C)$ and waits for the e_n event to execute the last part of the rule - $R_n(A)$.

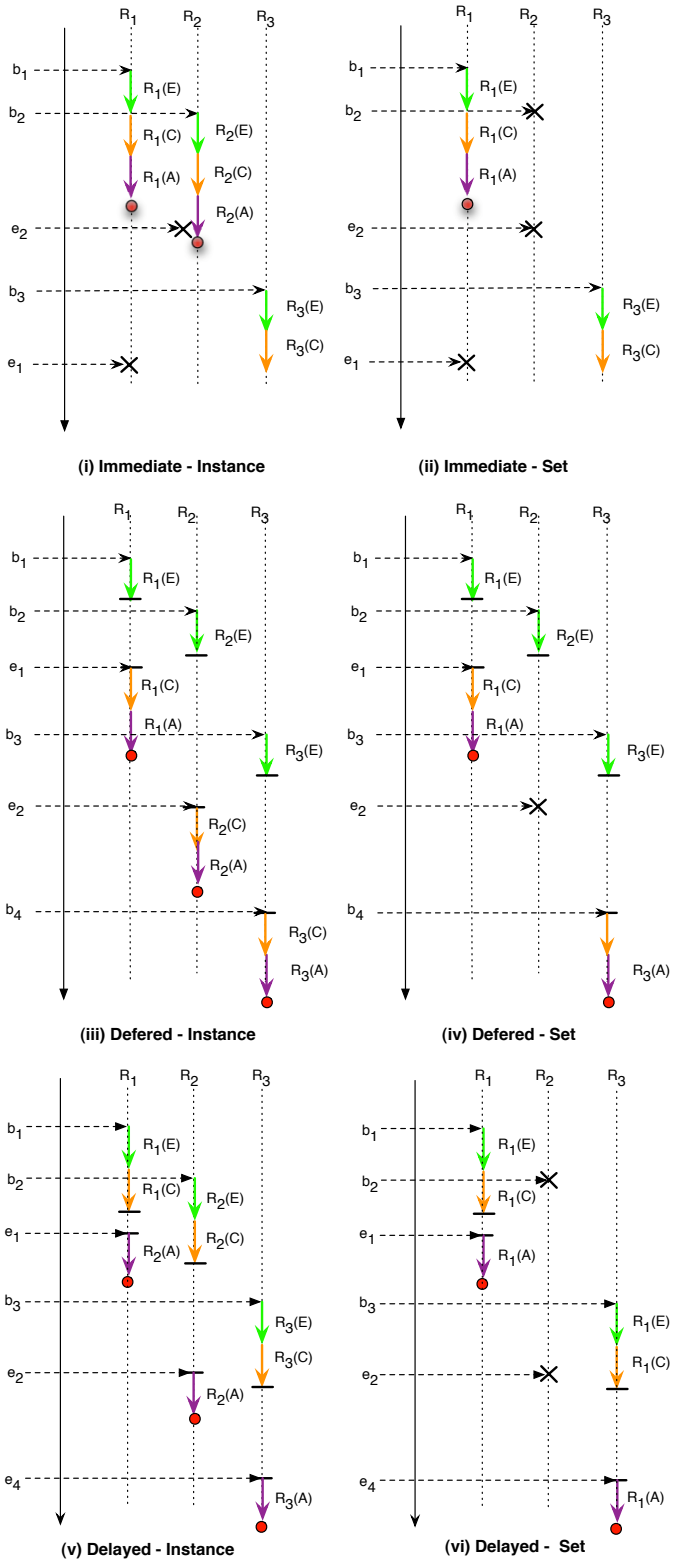

Figure 2: Execution Models

6. $<$ Delayed, Set $>$: on b_1, i.e, event indicating the start of the triggering operation, a rule R_1 is triggered and stops after condition evaluation. All b_is that occur in this period, are consumed by the R_1. Once the event e_1 is received, rule R_1 resumes execution and completes its last phase $R_1(A)$. All e_is, whose complementary b_is have been consumed by the rule R_1. The next b_i that occurs, a rule is triggered and the same strategy is followed.

Conclusion. To sum up on our example, the rule execution dimensions of the thread management rule take the following values: *Event Processing Mode : set*, *Execution Mode : defered*, *Activity Mode : separate*. In other words, the rule is executed once for a set of events, after the triggering operation's execution returns, in a separate execution thread. In our simple scenario with only one rule and the three dimensions (including one with 3 values) that we consider in this article, we already get 12 combinations of values, i.e. potentially twelve different ways of executing the rule. Note that this does not have a big impact with our sample basic rule but think of a real system with many rules. To the rule programmer facing the complexity of rules semantics (execution model), we propose a architectural framework in which rule semantics is explicitely programmed/embodied into software components.

4. ARCHITECTURAL INTEGRATION OF ECA RULES IN COMPONENT-BASED SYSTEMS

As advocated by IBM in its autonomic computing manifesto [9], a *supervision loop* (or known as *control loop* in control theory terminology [11]) has to be realized in order to provide autonomic behaviour to a target system. On similar lines, we define an autonomic system as composed of an autonomic infrastructure superimposed on a target system, where the autonomic insfrastructure is responsible for implementing the control loop. At the heart of the control loop are reaction mechanisms that, on analysis of the events of interest, determine the action operations needed to achieve the objectives. Our reaction mechanism is formed of active rules, whose structure and execution have been explained thoroughly in the previous section. The thread management rule, defined in the introduction section, is one such active rule. This section details how such rules are architecturally integrated with an underlying target system.

If it is legitimate to work towards adding autonomic behaviour a posteriori to any system and even more to tackle explicitly existing legacy systems, we believe it is likely more advantageous to build explicitly, a priori, the system in a "certain way" to be able to make them autonomous in a flexible and generic way. This "certain way" is the component-based approach - and more precisely the Fractal component model [2], which has, according to us, interesting properties for the realization of autonomic systems. It has been mentioned before that our webserver - Comanche - is implemented as Fractal components.

4.1 Canonical Autonomic Architecture

An architecture for autonomic computing[14] must accomplish some fundamental goals, outlined in IBM's autonomic computing manifesto[9]:

1. It should possess knowledge about itself and about its execution environment in order to be able to detect modifications taking place externally in its environment, or in its behaviour to subsequently undertake corrective actions. It must describe how to compose these components so that the components can cooperate toward the goals of system-wide self-management.

2. It should be adaptable, i.e., its construction should be based on a structuring model which can isolate its constituting elements, and subject them to adaptations - and on operational techniques to actually perform these adaptations (interception, programs transformation, etc.). It should be able to dynamically adapt or reconfigure itself to varying and unpredictable environments without any explicit user intervention.

These key features are present in the Fractal component model, and we believe Fractal/Julia (its implementation in Java) is a suitable substrate framework for autonomic systems development as illustrated in the sequel.

Fractal Component Model. The Fractal[2] initiative aims at supporting component-based development, deployment and management (monitoring and dynamic reconfiguration) of complex software systems, including in particular operating systems and middleware. It includes several extensions coming from research works, for management (e.g. Fractal JMX), security, transactions support, etc. Fractal is also used for developing several middlewares such as Speedo - a Java Data Object implementation, CLIF - a load injection framework, etc[2]. The Fractal component model relies on some classical concepts in CBSE: *components* are runtime entities that conforms to the model, *interfaces* are the only interaction points between components that express dependencies between components in terms of *required/client* and *provided/server* interfaces, *bindings* are communication channels between component interfaces that can be primitive, i.e. local to an address space or composite, i.e. made of components and bindings for distribution or security purposes. Fractal also exhibits more original concepts. A component is the composition of a *membrane* and a *content*. The membrane exercices an *arbitrary reflexive control* over its content (including interception of messages, modification of message parameters, etc.). A membrane is composed of a set of *controllers* that may or may not export control interfaces accessible from outside the considered component. For runtime information on the component system, the control interfaces provide with (meta) information about the its structure and also means to manipulate this structure. The model is *recursive (hierarchical) with sharing* at arbitrary levels. The recursion stops with base components that have an empty content. Base components encapsulate entities in an underlying programming language. A component can be shared by multiple enclosing components. Finally, the model is programming *language independent* and *open*: everything (e.g. controllers, type system) is optional and extensible[3] in the model, which only defines some "standard" API for controlling bindings between components, the hierarchical structure of a component system or the components life-cycle (e.g. start, stop).

The Julia Implementation. Julia is an execution support for Fractal components written in Java. It is a full-fledged implementation of Fractal that supports the highest conformance level. More fundamentally, Julia is a software framework dedicated to components membrane programming. It is a small run-time library together with bytecode generators that relies on an AOP-like mechanism based on mixins and interceptors. A component membrane in Julia is basically a set of controller and interceptor objects. A mixin mechanism based on lexicographical conventions is used to compose controller classes. Julia comes with a library of mixins and interceptor classes, the component programmer can compose and/or extend. It is worth mentioning that Julia's membranes are particularly suited to insert sensors for observing and actuators for controlling components.

(Re)Configuration Languages. For the configuration and deployment of a Fractal-based system, an Architecture Description Language (ADL), known as Fractal ADL, is used to describe the system architecture. It is XML-based and strongly typed. It describes the interfaces of components (names and signatures), the subcomponents, the bindings between the various components, the initial values of component properties and the implementation of primitive components(e.g., the name of a Java class). All static information on a component is provided by the Fractal ADL. FScript is a scripting language used to describe architectural reconfigurations of Fractal components. FScript includes a special notation called FPath (loosely inspired by XPath) to query, i.e. navigate and select elements from Fractal architectures (components, interfaces...) according to some properties (e.g. which components are connected to this particular component? how many components are bound to this particular component?). FPath is used inside FScript to select the elements to reconfigure, but can be used by itself as a query language for Fractal.

Most of our approach relies on the Fractal component model. Of course, in realistic industrial settings, we cannot assume a whole distributed system to be Fractal-based; but we argue that those non-Fractal parts (legacy components) can be wrapped into Fractal components, and extended with autonomic behaviours. In the same line of thought, we believe that it would be very advantageous to carry out the development of the autonomic infrastructure itself in the form of Fractal components so as to consider the autonomic management of the autonomic infrastructure itself. (Our work

[2]CLIF, Speedo and other middleware engineered with Fractal are available in open source at http://www.objectweb.org/.

[3]This openness leads to the need for conformance levels and conformance test suites so as to compare distinct implementations of the model.

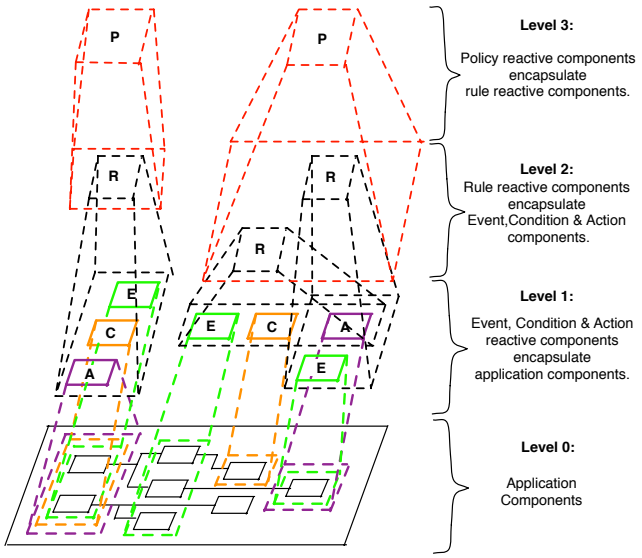

Figure 3: Architectural Vision

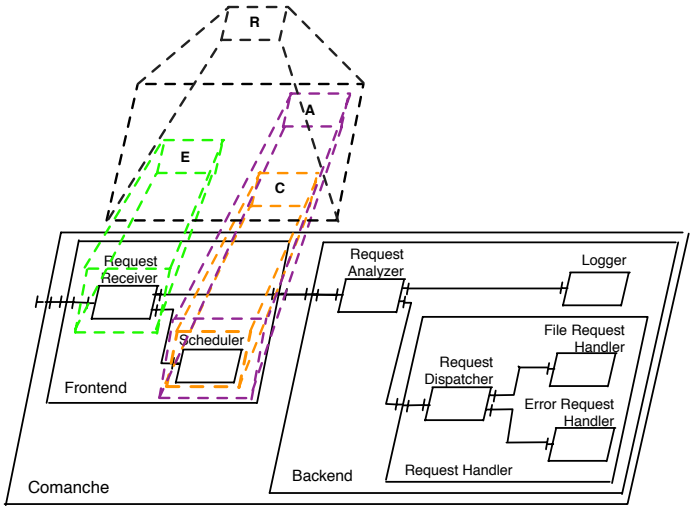

Figure 4: Thread Management Rule

can be considered as the first steps in this direction.)

4.2 Reactive part of the Canonical Autonomic Architecture

The architecture of the autonomic infrastructure is inspired from the fundamental management notion of *domain*[13], which consists in grouping the components on which the various reactive operations can be carried out. More formally, a domain is:

- *a unit of composition* to enable physical or logical partitioning of the application components, and
- *a unit of control* to define the type of control that needs to be carried out on these components.

The similarities between a Fractal component and the concept of domain suggest that a domain can be aptly modelised as a Fractal component. To incorporate reactive behaviour, several types of domains have been defined, each with a particular type of control unit applied onto its composition unit. They are each represented by a Fractal component, known as a *reactive* component. The autonomic infrastructure is formed by these reactive components. The various reactive components with their specific functionalities are listed below and illustrated by Figure 3.

- An *Event* (E) reactive component encapsulates application components where events of interest need to be detected. The membrane of this reactive component is responsible for identifying the application components that would belong to the content, instrumenting them appropriately. Further, on the occurrence of events of interest, they are notified to the appropriate controller inside the membrane of the component which processes them as defined in the rule execution model.
- A *Condition* (C) reactive component contains application components that represent the scope the queries that are to be evaluated. The functionnalities of its membrane include identifying the application components that would be in its jurisdiction, and evaluating queries on them.
- An *Action* (A) reactive component encapsulates application components, on which actions are executed. The type of control enforced by the Action component's membrane involves identifying its content's constituents, and executing some corrective operations on request.
- A *Rule* (R) reactive component coordinates the processing of a rule. It contains (exactly) one instance of the 3 above reactive components, i.e, Event, Condition (optional) and Action. The control applied by its membrane is the execution coordination of these reactive components. It is responsible for the execution of the rule embodying on a particular rule execution model.
- A *Policy* (P) reactive component's sole purpose is to coordinate the execution of the rules based on an execution strategy for a set of rules. The content part of the policy reactive component contains rule components, and its membrane controls the rights to their execution. Only, on explicit notification by the policy membrane, can the rules, once triggered, continue processing.

Figure 3 shows the relationships between the various reactive components. The architecture employs key features of the Fractal Component Model [2], notably: the containment relationship - which can be found in components in the top three levels, where each component has components from the lower level as sub-components, e.g, the Policy component at level 3 contains the Rule components of level 2; and overlapping of reactive domains, thanks to the sharing property, e.g, an application component can belong to several reactive components. Figure 4 represents the implementation of our thread management rule. The Event reactive component encapsulates the request receiver component, because

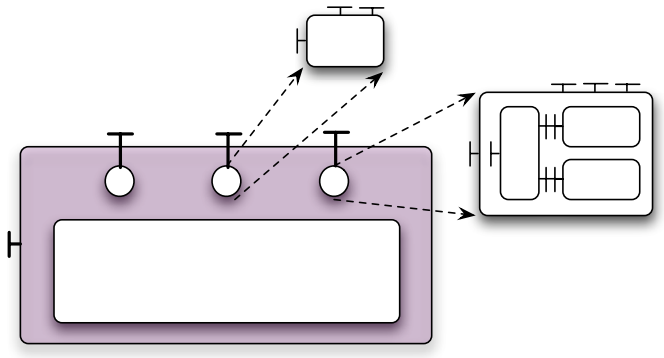

Figure 5: Extended Fractal Component

the event of interest for the rule occurs on it. The condition and action parts encapsulate the scheduler component. For their respective operations, that of verifying whether the thread pool is empty and increasing the size of the thread pool occurs on the same component. Finally, we have the Rule composite that encapsulates the reactive components (Event, Condition & Action) to coordinate their execution. The Policy reactive component is inexistant in the figure since, due to space limitations, only a single rule has been taken as an example to illustrate the framework.

5. AUTONOMIC INFRASTRUCTURE

This section presents the extensions in the prototypal implementation of our proposed framework, that offers the flexibility feature of our autonomic infrastructure.

Each of the reactive components presented above encapsulate either application components or other reactive components, where structurally both these types of components are similar. But the controller unit of each of these reactive components differs from one another because they implement different dimensions of the rule execution model. Therefore, the generic structure of a reactive component is a standard Fractal component - a composite component to be precise, with a flexible membrane formed by a set of newly defined and existing controllers (cf. Figure 5). The Fractal specification contains several examples of useful forms of controllers, which can be combined and extended to yield components with different reflective features. Likewise, additional controllers, not defined in the Fractal specification, can also be defined and incorporated in the membrane of a component. This permits the reactive components to have a membrane adapted to its respective rule execution dimensions. The membrane of a reactive component is composed of the following types of controllers : i) standard Fractal controllers as defined in the Fractal specification, ii) standard Fractal controllers represented as classical Fractal components with extended operations and finally iii) new controllers represented as Fractal components.

For instance, the membrane of the Event reactive component is composed of: i) an extended *attribute controller* to specify the parameters of the rule execution dimensions, ii) a *content controller* to add/remove the application components and iii) a *event processing controller* to process the events of interest and notify its enclosing rule component. The event processing controller is a newly defined controller represented as a Fractal component. It is a composite component with corresponding sub-components for the following dimensions: execution mode, activity mode and event processing mode. However, at the time our framework was developed, the set of controllers that constitute the membrane of a component could not be dynamically modified in the Julia implementation, nor could the functionalities of the existing controllers be modified. We have thus extended the Julia implementation to support these necessary features.

6. RELATED WORKS

Decision-making/reaction mechanisms form the core of an autonomic control loop, several systems use rules that specify conditions to be monitored and operations that should be executed when certain conditions are detected. *Production rules* (or *deductive rules*) have the following format - "IF *condition expression* THEN *action list*". These rules can also been extended, as in the DIOS++ framework[10], where an "ELSE" part as been added at the end - "IF *condition expression* THEN *action list* ELSE *action list*". ACEEL [3] uses *adaptation rules* which are couplets of the form "*OnEvent* : *Action*" with the first part defining the triggering event and the latter describing what actions to be performed. Inspired by triggers, we use active rules, which can be considered as a combinaison of the above two broad types of rules: "ON *event expression* IF *condition expression* THEN *action list*". Beyond these syntaxical differences, the main differences between production (or adaptation) rules and active rules concen their execution model. The execution model of production rules is based on the Rete algorithm: events are seen as facts which are added to the knowledge base; rules infer new facts from these facts ; the process stop when a fixed point is reached (no new facts can be inferred). Production or adaptation rule execution models are thus fixed and invariant. By contrast, active rules models are much more powerful and flexible. Also active rules models encompass the connection between the target system and the reaction mechanism while production rules systems do not (inference by production rules is disconnected from the actual execution of the target system).

For incorporating these active rules, the approach followed in SAFRAN[5], K-Components[7] consists in enhancing the computational component model with rule abstractions, where all rules concerning a particular component are injected into to it. Another approach, followed in Autonomia[6] or Automate[1], consists in implementing an autonomic computing infrastructure that acts as a control layer superimposed on the application, that provides the application as well as its individual components with the basic autonomic services to make it autonomic. In our approach, rules are not adhoc features injected into components but are themselves first-class components which can be manipulated as such. Thanks to the domain concept, the architectural connection between the application components and the rules are through containment relationships (hence a rule is not tied to a single component). Enabling thus, easy modification of the rule constituents (Event, Condition & Action).

In summary, to our knowledge, several works have tackled the architectural issues involved in the implementation of a control loop for autonomic features but none make such an explicit and extensive use of component programming for implementing the autonomic features themselves as in our proposition. As a consequence, these approaches often result in ad-hoc and not flexible management of the autonomic features. In most approaches that consider rules of a sort or another (deductive, active, etc.) as the core mechanism for autonomic features, rules execution issues have not been addressed in depth. Their execution strategy/methodology have been taken for granted, and many issues have been left under specified and ambiguous (we only find a brief mention of rule execution model in SAFRAN).

7. CONCLUSION

This article focuses on the architecture and behaviour of autonomic control loops. It proposes to use active rules as a decision-making mechanism, for which we have proposed i) a rule model, which is composed of a rule definition model and a rule execution model, to provide a clear semantics for the integration of rules, and ii) a flexible architecture that permits to dynamically add/delete new rules, to modify the rule definition model as well as the rule execution model of rules. Our rule execution model comprises of a set of dimensions, which we claim is not fully comprehensive, other dimensions can be envisaged. But we do claim that our generic architecture can incorporate new dimensions not yet identified. Further, our autonomic infrastructure takes into consideration the evolution of the target system. We would like to add that our work being positioned on components, i.e, our autonomic infrastructure as well as the underlying target system being component-based, has permitted us to benefit from CBSE properties.

The proposed autonomic infrastructure was developed in a Java implementation of the Fractal component model. The dimensions outlined in the article have been implemented, and experimented on the Fractal-based Comanche webserver. Some other execution dimensions, e.g. those related to the execution of multiple rules, which have not been discussed here due to space constraints have also been implemented.

Several future research directions are envisaged. In order to assess the validity of our proposition, we wish to apply it on more realistic applications. This would eventually permit us to determine the set of execution dimensions that are most relevant for component-based systems. Further, to enrich our proposition, we foresee a formalism for the definition of active rules that would typically be an extension of the Fractal ADL. This extension should be quite straitforward, for Fractal ADL is actually modular and extensible. On a longer term, we shall study the problem of interference between the behaviour of a control loop (i.e. the rules in our case) and the target system and the stability of the global system (target system and rules).

Acknowledgments. This work is partially supported by the French RNTL Selfware project and the European IST Selfman project. The authors thank A. Lefebvre and B. Dillenseger for their careful reading and comments.

8. ADDITIONAL AUTHORS

9. REFERENCES

[1] M. Agarwal, V. Bhat, H. Liu, V. Matossian, V. Putty, C. Schmidt, G. Zhang, L. Zhen, M. Parashar, B. Khargharia, and S. Hariri. Automate: Enabling autonomic applications on the grid. *Autonomic Computing Workshop*, pages 48–57, 2003.

[2] E. Bruneton, T. Coupaye, M. Leclercq, V. Quema, and J.-B. Stefani. The FRACTAL component model and its support in Java: Experiences with Auto-adaptive and Reconfigurable Systems. *Softw. Pract. Exper.*, 36(11-12):1257–1284, 2006.

[3] D. Chefrour. Developing component based adaptive applications in mobile environments. In *SAC '05: Proc of the 2005 ACM symposium on Applied computing*, pages 1146–1150, New York, NY, USA, 2005. ACM Press.

[4] T. Coupaye and C. Collet. Detailed sketch of a parametric execution model for active database systems. Technical report, LSR - IMAG Laboratory, University of Grenoble. France, 1997.

[5] P.-C. David and T. Ledoux. An Aspect-Oriented Approach for Developing Self-Adaptive Fractal Components. In *5th International Symposium on Software Composition (SC'06)*, Lecture Notes in Computer Science, Vienna, Austria, march 2006. Springer-Verlag.

[6] X. Dong, S. Hariri, L. Xue, H. Chen, M. Zhang, S. Pavuluri, and S. Rao. Autonomia: an autonomic computing environment. In *Proc of the 2003 IEEE Int'l Conf on Performance, Computing, and Communications*, pages 61–68, 2003.

[7] J. Dowling and V. Cahill. The k-component architecture meta-model for self-adaptive software. In *Proceedings of the Third Int'l Conf on Metalevel Architectures and Separation of Crosscutting Concerns*, pages 81–88, London, UK, 2001. Springer-Verlag.

[8] S. Gatziu, A. Koschel, G. von Bültzingsloewen, and H. Fritschi. Unbundling active functionality. *SIGMOD Record*, 27(1), Mar. 1998.

[9] P. Horn. Autonomic computing: Ibm's perspective on the state of information technology. Technical report, IBM Corporations, October 2001.

[10] H. Liu and M. Parashar. Dios++: A framework for rule-based autonomic management of distributed scientific applications. In *Euro-Par*, pages 66–73, 2003.

[11] M. Kokar and K. Baclawski and Y. Eracar. Control Theory Based Foundations of Self Controlling Software. *IEEE Intelligent Systems*, 14(3):37–45, 1999.

[12] N. W. Paton, F. Schneider, and D. Gries, editors. *Active Rules in Database Systems*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 1998.

[13] M. Sloman and K. Twidle. Domains: a framework for structuring management policy. *Network and distributed systems management*, pages 433–453, 1994.

[14] S. R. White, J. E. Hanson, I. Whalley, D. M. Chess, and J. O. Kephart. An Architectural Approach to Autonomic Computing. In *Int'l Conf in Autonomic Computing*, pages 2–9, New York, NY, 2004.