Towards a Framework for Self-adaptive Component-Based Applications*

Pierre-Charles David and Thomas Ledoux

OBASCO Group, EMN/INRIA
École des Mines de Nantes – Dépt. Informatique
4, rue Alfred Kastler – BP 20722
F-44307 Nantes Cedex 3, France
{pcdavid,ledoux}@emn.fr

Abstract. Nowadays, applications must work in highly dynamic environments, where resources availability, among other factors, can evolve at runtime. To deal with this situation, applications must be *self-adaptive*, that is adapt themselves to their environment and its evolutions. Our goal is to enable systematic development of self-adaptive component-based applications using the Separation of Concerns principle: we consider *adaptation to a specific execution context and its evolutions* as a concern which should be treated separately from the rest of an application. In this paper, we first present the general approach we propose and the corresponding development framework and tools we are developing to support it. Then, in order to validate this approach, we use it on a small component-based application to show how it can be made self-adaptive.

1 Introduction

Given today's fast pace of technological evolutions and diversity of computing platforms (both hardware and software), building applications which can work in such a wide range of systems is becoming more and more challenging. Moreover, even on a specific platform, the execution context and available resources tend to vary a lot at runtime. This is particularly the case with distributed applications, as those have to deal with the heterogeneity of different hosts and rely on network communication, which are often highly variable (especially with newer technologies like wireless networking).

The result of this situation is that applications should be *adaptable* (i.e. able to be adapted), or even better *self-adaptive* (i.e. adapting themselves) to their environment [1]. Adaptation in itself is nothing new, but it is generally done in an ad hoc way, which involves trying to predict future execution conditions at development time and embedding the adaptation decisions in the application code itself. This creates several problems: increased complexity (business logic polluted with non-functional concerns) and poor reuse of components caused

* This research is supported by the RNTL project ARCAD (http://arcad.essi.fr/)

J.-B. Stefani, I. Demeure, and D. Hagimont (Eds.): DAIS 2003, LNCS 2893, pp. 1–14, 2003.

by a strong coupling with a specific environment. We believe that some kinds of adaptations, most notably those related to resource usage, can be decoupled from pure functional concerns, and that doing so does not have the same drawbacks as the ad hoc approach.

The long term goal of our work is to enable a more systematic (as opposed to ad hoc) way to develop self-adaptive component-based applications. Our approach is based on the Separation of Concerns [2] principle: we consider *adaptation to a specific execution context and its evolutions* as a concern which should be treated separately from the rest of an application. Ideally, application developers should be able to concentrate on pure business logic, and write their code without worrying about the characteristics and resource limitations of the platform(s) it will be deployed on. Then, the *adaptation logic* [1], which deals specifically with the adaptation concern, is added to this non-adaptive code, resulting in a self-adaptive application able to reconfigure its architecture and parameters to always fit its evolving environment.

In this paper, we first present the general approach we propose and the corresponding development framework and tools we are developing to support it. This framework is based on the Fractal component model [3], chosen for its flexibility and dynamicity. To supplement this model, we propose a *context-awareness* service to provide information about the execution context. This information is used by *adaptation policies*, which constitute the third part of our framework and capture the adaptation concern. Then, in order to validate this approach, we present how a small component-based application can be made self-adaptive using it. In this example, we show how different kinds of performance-related adaptations can be applied non-invasively to the original application depending on its dynamic execution context.

This paper is divided in two main parts: in the first one (Sec. 2), we describe our approach in more details and present the three main parts of our framework (Sec. 2.1, 2.2, 2.3); in the second one (Sec. 3), we present a very simple application developed using our approach. Finally, we compare our approach to some other related work (Sec. 4) before concluding (Sec. 5).

2 General Approach

The general idea of our approach is to consider adaptability as a concern [2] which can be separated from the core concerns (i.e. business logic) of an application. We think that regardless of the specific domain of an application, adaptability concerns can often be expressed in the same way. Our goal is thus to identify a general form of these concerns and to capture it in a framework and accompanying tools. In practice, we do not pretend we have found the one and only mean to express adaptability concerns, but we believe that the approach we propose is general enough to be used in a wide range of adaptations.

More concretely, the development model we envision is as follows:

1. First, the application programmers create the core of the application (pure business logic), without worrying about the characteristics of the environment(s) it will/may be deployed in.

2. Then, once the deployer or administrator knows more precisely both the execution context(s) the application will be deployed in and its non-functional requirements , he can specify (using the tools we provide) how the application must be made to adapt itself to this context.

What we need as the output of the first phase is an *adaptable application*: an application which exhibits enough flexibility so that it *can* be reconfigured to fit different execution conditions. In the second phase, we take this adaptable application and make it *self-adaptive* by incorporating the additional logic required to drive these reconfigurations appropriately in response to the evolutions of the execution context. This integration of the adaptation logic can be done by source code transformation of the application, but although it enables adaptations to *happen* at run-time, this approach can not deal with *unanticipated adaptations* [4,5] which will only be known during the execution. The solution we adopted is to host the adaptable application inside a suitable execution platform which is responsible for interpreting the adaptation logic and reconfigure the application when required. This way, not only do the adaptations happen at run-time, but the adaptation logic itself can be modified during the execution, without stopping and restarting the application.

The kind and extent of flexibility the adaptable application provides determine the nature of the adaptations that will be possible. In our case, we deal with component-based applications which are explicitly structured through their *architecture*. The first kind of reconfigurations we support is thus *structural*: modifications of the connections between components and of their containment relationships. This allows for example to disconnect a component and replace it with another one, compatible with it but better suited to a specific usage context. As we will see, the component model we chose, Fractal [3], also supports the notion of component configuration through parameters. The second kind of reconfiguration we provide is thus *parameterization* (for example, changing the colors of GUI components if we know the user is color-blind). These two kinds of reconfiguration already enable powerful adaptations, but they both have the same drawback: they must be anticipated, at least partially, at development time. It is the application programmer (or assembler) who defines the architecture and which component parameters will be reconfigurable. In order to support unanticipated adaptations [5] at run-time, we needed a mechanism to transparently modify the behavior of components. As the Fractal component model did not support this natively, we had to extend it. The extension we added uses reflection to provide a meta-programming [6] interface to Fractal components (see Sec. 2.1 for details).

Having an *adaptable* application with the required level of flexibility and a set of reconfiguration operations is only the first step. In order to make this application *self-adaptive*, we need to add adaptation logic [1] whose role is to determine *when* to apply *which* specific reconfigurations. In our approach, this adaptation logic takes the form of *adaptation policies* which can be attached to or detached from individual components during the execution. An adaptation policy is a set of rules modeled after the Event – Condition – Action rules from

active databases [7]: when the event described in the first part of a rule occur, the condition is evaluated, and if it holds, the action is then applied. This allows to react to changes in the execution context by reconfiguring the application.

To summarize the global picture of our approach, the development and execution of a component-based self-adaptive application can be decomposed in two phases:

1. Development of an adaptable application by application programmers and assemblers using an appropriate component model, but without worrying about the details of the execution context it will be deployed in.
2. Definition of adaptation policies and binding of these policies to the components of the application by the deployer. This binding is dynamic and happens at run-time so adaptation policies can be defined and attached to components when the real execution conditions are better known.

The resulting application is executed in an appropriate software platform which interprets the adaptation policies and ensures that all the adaptations are applied correctly (no disruption in the execution of the application, no conflicts between policies...).

2.1 Using Fractal to Build Adaptable Applications

In this section we present the Fractal component model that we have chosen as a base for our framework and show how it supports our requirements for adaptable applications.

Fractal [3] is a general (i.e. not domain-specific) component model for Java part of the ObjectWeb consortium[1]. It supports the definition of primitive and composite components, bindings between the interfaces provided or required by these components, and hierarchic composition (including sharing). Unlike other Java-based component models, like Jiazzy [8] or ArchJava [9], Fractal is not a language extension, but a run-time library which enables the specification and manipulation of components and architectures.

What makes Fractal particularly suited in our case is that because of its nature as a run-time library, it is highly dynamic and reflective. In practice, Fractal is presented as an API which can be used to create and manipulate complex architectures using plain Java classes as building blocks. Its programmatic approach makes it an ideal base to build tools on top of it.

Fractal distinguishes two kinds of components: primitives which contain the actual code, and composites which are only used as a mechanism to deal with a group of components as a whole, while potentially hiding some of the features of the subcomponents. Primitives are actually simple, standard Java classes conforming to some coding conventions. Fractal does not impose any limit on the levels of composition, hence its name. Each Fractal component is made of two parts: a controller which exposes the component's interfaces, and a content which can be either a user class in the case of a primitive or other components in the

[1] `http://www.objectweb.org/`

case of a composite. All interactions between components passes through their controller.

The model thus provides two mechanisms to define the architecture of an application: bindings between interfaces of components, and encapsulation of a group of components into a composite. Because Fractal is fully dynamic and reflective (in the sense that components and interfaces are first-class entities), applications built using it inherently support structural reconfiguration. Fractal also supports component parameterization: if a component can be configured using parameters, it should expose this feature as a Fractal interface identified by a reserved name so that it is available in a standard way. This gives us the second kind of adaptations we want to support.

In order to support adaptations not anticipated at development time [4,5], we needed a mechanism to modify transparently the behavior of Fractal components. Fractal does not provide this kind of mechanism by default. However its reference implementation is very extensible and it was possible to add the required feature, which can be thought of as a simple "Meta Object Protocol" [6] for Fractal. As every interaction between components passes through the controller part, we extended the default Fractal controller so that it can reify all the messages it receives (see Fig. 1). Instead of being sent to its original target, the reified message is sent to a subcomponent which implements a meta-level message invocation interface. The component can then process the message in a generic way, doing pre- or post-processing, or even completely replacing the original behavior. If this meta-level component is a composite, it is then relatively easy to support dynamic addition and removal of multiple generic behavior modifications (though the correct composition of meta-level behaviors is an open problem [10] which we do not address).

2.2 Context-Awareness Service

The role of the context-awareness service is to provide the rest of the framework precise and up-to-date information about the execution context of the application [11]. This information is required to decide when it becomes necessary to trigger an adaptation. We can distinguish three parts in context-awareness:

1. *Acquisition* of raw data from outside the application into a form that can be further manipulated.
2. *Representation* or *structuring* of these informations using an ontology.
3. *Reasoning* on the resulting knowledge to detect interesting situations which ask for a reaction/adaptation.

The remaining of this section describes how these functions are realized in our framework.

Acquisition. In our framework, acquisition is delegated to libraries of probes. A probe is simply a Java object which, when invoked, return an object regrouping named samples (values with a time-stamp indicating the moment it was

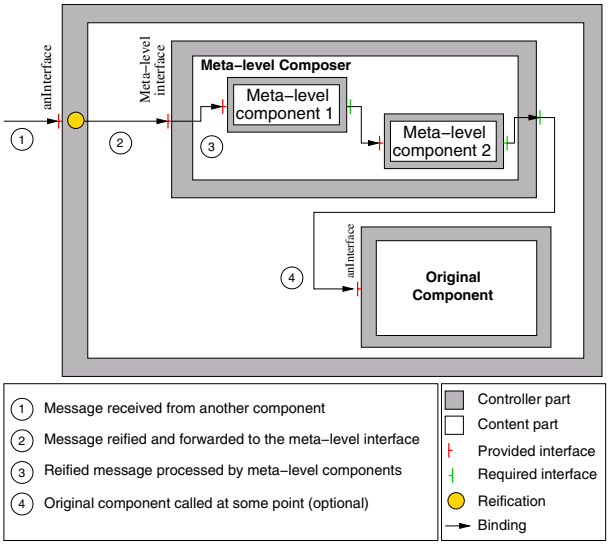

Fig. 1. MOP-like extension for Fractal.

acquired). The system maintains a set of instantiated probes, each with a particular sampling rate. A scheduler running in its own thread invokes each probe in turn, according to its sampling rate, and collects the results of the measures, to be consumed by the other parts of the framework. Libraries of probes to observe different kinds of context can be developed independently of an application, although some would probably depend on the underlying platform (OS, hardware). For example, our current implementation uses Linux's `/proc` pseudo file-system to gather information about the CPU load, available memory, and usage of the network connection.

Representation. The raw data collected by the probes is just a set of unrelated measures, not in a form appropriate to support any kind of sophisticated reasoning. The second step towards context-awareness is to structure this information into a model of the environment using an appropriate ontology. Because our framework is domain-independent, we chose a rather simple and generic meta-model to structure these informations. The context is represented as a tree of *resources* identified by a name. Resources can represent actual elements of the context (like a disk if the domain is hardware resources, or a room if the domain deals with physical informations) or a logical group of other resources (for example, a category named `storage` to regroup disk, memory...). Each resource is described by a set of named attributes, whose value change over time.

The structure of the resource tree is defined by a configuration file read at startup, which also indicates the mapping between the informations gathered by probes and the resources attributes. The current design assumes that the hierarchical structure of resources is fixed at startup time and that neither resources

nor attributes can appear or disappear. Future versions will remove these limitations to reflect more accurately the dynamic nature of real execution contexts.

The system supports multiple parallel resource trees, identified by a name. Each such tree is called a *context-domain* and models one particular aspect of the application context. Examples of possible context-domains inlude hardware and software resources (CPU, memory, disk, libraries...), network topology and performance, physical environment (geographic position, temperature...) and user preferences (should applications use sound or stay quiet...). Given the appropriate library of probes and ontology to structure them, all these domains can be treated in a uniform way by our framework.

Reasoning. Even structured, information gathered by the probes are generally very low level and not appropriate for deciding when an adaptation is required. Indeed, which particular information is available to probes can vary from system to system.

In addition to primitive attributes containing raw data collected by probes, it is also possible to define *synthetic attributes*. These attributes are defined by an expression over the whole context domain the attribute belongs to. This feature allows to define more abstract attributes derived from the low level data collected by probes. For example, if probes can get from the underlying OS the number of bytes sent and received by a network interface, these values can be used to compute synthetic attributes representing incoming and outgoing traffic in MB/s. Combining this with the expected maximum throughput of the network card (available thanks to another probe), another attribute could be defined to represent the percentage of the current bandwidth usage.

The system uses the data regularly collected by the scheduler to update the values of all the synthetic variables, automatically taking care of dependencies. This way, an up-to-date model of the execution context of the application is always available to the rest of the system. However, what we have described until now gives us only snapshots (albeit regularly updated) of the state of the context. More sophisticated reasoning requires not only awareness of the immediate state of the context, but also of how it evolves over time. This is supported by the possibility to define composite event in the rules of adaptation policies (see Sec. 2.3).

The context-awareness service is available to the rest of the system through two interfaces: a simple query interface and an asynchronous notification interface. To use the query interface, a client object simply sends a request in the form of an expression, similar to those used to define synthetic variables. The system immediately evaluates the expression relative to the current state of its knowledge and returns the current value of the expression. To use the notification interface, an object also sends the expression it is interested in, but this is used to register the client as a listener to the expression. From this moment, until the client unregisters itself, the context-awareness service will notify the client each time the value of the expression changes. These expressions are actually managed almost exactly like normal synthetic variables, except that they are not associated to a resource but to a client object.

2.3 Adaptation Policies

The role of adaptation policies is twofold: first, to detect significant changes in the execution context of the application using the information made available by the context-awareness service, and then to decide which reconfigurations must be applied to the application when these changes occur.

To do this, an adaptation policy consists in a set of rules, each of the form Event – Condition – Action (modeled after the ECA paradigm used for example in active databases [7]):

- the *event* part describes the circumstances in which a rule must be triggered, using primitive events which can be combined to reason on the evolution of the context over time (see below);
- *conditions* are simple guards, evaluated when a rule is triggered to determine if the action should be applied;
- *action* is a (sequence of) reconfiguration operation(s) among those presented in Sec. 2.1 which are applied to the system when the rule is activated.

Each Fractal component in the application can have one or more adaptation policy dynamically attached to (or detached from) it. The policies are actually contained inside the controller part of the component, and are accessible through a specific Fractal interface automatically added to every self-adaptive component.

We distinguish two kinds of *primitive events* depending on their source (external or internal). The first kind corresponds to changes in the execution context, and more precisely to any change in the value of an expression over a context-domain. The second kind of events corresponds to things happening in the application itself, like messages sent between component, creation of new components or architectural reconfigurations triggered by the application itself (and not by an adaptation policy).

Both kinds of events can be combined to form *composite events* using a set of operators to detect sequences, alternatives or conjunctions of events. Using these mechanisms, it is possible to define adaptation policies which can react not only to the immediate state of the application or of its context, but also to its evolutions.

Conditions are simple boolean expressions defined in the same language as for synthetic variables, used as guards.

Actions consist in a sequence of concrete reconfiguration actions among the set already presented: structural reconfigurations, parameterization, addition or removal of a generic service using the MOP-like extension. Possible actions are limited to those directly implying the component an adaptation policy is attached to.

Although the adaptation policies are currently coded in plain Java, our goal is to define a DSL (Domain Specific Language [12]) to write these. This would enable verifications on the validity of the policies (to be defined more precisely) and to detect possible conflicts between policies, for example when two policies react differently to the same situation, or when their interaction would create unstable behaviors.

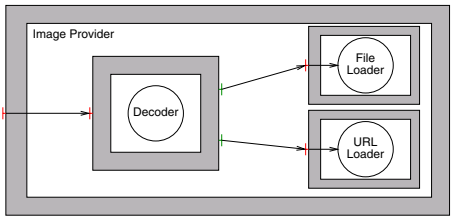

Fig. 2. Initial architecture of the example application.

3 Example Application of Our Approach

This section describes through a small example how the approach we propose can be used to transparently make a component-based application self-adaptive. First, we describe the example application and the problems it can have under certain circumstances. Then, we show how to apply three different (but cumulative) adaptations of this application using our approach, illustrating the three different types of reconfigurations we support. These adaptations overcome the original limitations of the application without breaking the encapsulation of the original components.

The application is an image viewer/browser. At its core is an image decoder component whose role is to interpret the content of an image file (JPEG, PNG...) into a bitmap which can be printed on a screen. It uses another component to load the content of the image files from a source location (local or remote). Figure 2 shows the corresponding part of the initial architecture of the application, coded using Fractal.

3.1 First Adaptation: Conditional Enabling of a Cache

The components in the application have been written to be simple and reusable, implementing one and only one thing. In particular, none of the loaders cache the files it loads. However, adding a cache would improve performance a lot in some circumstances. The first adaptation we will implement is thus the conditional enabling of a transparent caching service. The best place to put a cache is on the decoder (the front end component): this way, we will not only cache the cost of fetching the content of the files but also the cost of decoding the images. Using the meta-programming extension we added to Fractal, it is not very difficult to add a generic caching service to our component: because it works at a meta-level, handling reified messages and responses, the cache doesn't have to be implemented specifically for our example but can be reused from a library of generic services. Figure 3 shows the internal structure of the decoding component once the cache is enabled. Thanks to the dynamicity of Fractal, going from the original decoder to this configuration can be done (and undone) at runtime and completely transparently.

The adaptation policy required to implement this behaviour is very simple, consisting in only one rule:

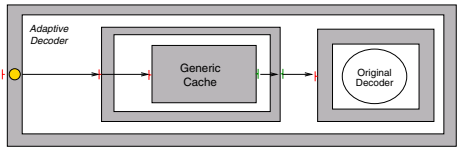

Fig. 3. Conditional cache enabling.

1. ***When*** the average execution time of a request (to load and decode an image) to the image provider goes above a given threshold, ***then*** instantiate a new generic cache component and attach it to the image provider.

This way, when the system detects that images take a long time to get, it will automatically adapt the image provider component by adding a cache service to increase performance when the same images are loaded several times. The average execution time of a request is computed by the adaptation policy attached to the component using the time-stamps of method receptions and method returns, two kinds of events available for writing the *Event* part of an adaptation rule.

3.2 Second Adaptation: Automatic Resizing of the Cache

The second adaptation we describe concerns the size of the cache, which we did not precise in the previous section. This can have a high impact on the performance of the system: on the one hand, if the cache is too small we will not use all of its potential; on the other hand, if it is too big compared to the memory available in the system, we risk to pay an even higher price from the use of swap space. The goal of this second adaptation is thus to adapt the size of the cache to the amount of free memory available on the system, which of course changes dynamically.

One originality of our approach which allows us to do this is that the generic cache we added to the application in the previous step is itself a Fractal component. This means we can use the same techniques as for the application component to adapt the cache service itself. To be able to do this, we assume that the cache component exposes its size as a reconfigurable parameter in the standard Fractal way. We can then use on the cache a second kind of reconfiguration: parameterization. The size of the cache will be determined according to the amount of free RAM available in the system. This amount is tracked by the context-awareness service and is available as `res:/storage/memory.free` (to be read as: the `free` attribute of the resource named `memory` in the `storage` category of the `res` context-domain). The adaptation policy to control the size of the cache has three rules (`free` is used as a shorthand for `res:/storage/memory.free`):

1. ***When*** the value of `free` changes, ***if*** it is less than *low*, ***then*** set the cache size to 0.
2. ***When*** the value of `free` changes, ***if*** it is greater than *high*, ***then*** set the cache size to *max*.

3. ***When*** the value of `free` changes, ***then*** set the cache size to $(\texttt{free} - \mathit{low}) \times \frac{\mathit{max}}{\mathit{high}-\mathit{low}}$.

These rules state that when the amount of free memory is below the *low* watermark, the cache is effectively disabled by setting its size to 0 (Rule 1). Otherwise, the size of the cache grows linearly with the amount of free memory (Rule 3) until it reaches a maximum value of *max* when the amount of free memory is more than *high* (Rule 2). By dynamically attaching such a policy to the cache, its size will always be adapted to the available resources.

3.3 Third Adaptation: Replacement Policy Selection

When a cache is full, it uses a replacement policy to decide which of the elements it contains must be removed. The most classical replacement policy is LRU (Least Recently Used), which chooses to drop the element in the cache which was not accessed for the longest time. This works well for random access patterns but can lead to trashing when the data is accessed sequentially [13]. Glass [13] presents a more sophisticated replacement policy called SEQ which behaves like LRU by default, but switches to MRU (Most Recently Used) behaviour when it detects the beginning of a sequential access. Although Glass implemented his SEQ algorithm in an ad hoc and much more sophisticated way, its general idea fits very well in our model of adaptation policies. We can easily implement a simplified version of SEQ in our application to illustrate the last kind of adaptation we support: structural reconfiguration. We suppose that the component implementing the core cache logic uses another component to implement the replacement policy, using a well-defined interface. An adaptation policy attached to the cache which can detect the start and end of sequential accesses to data can then use the structural reconfiguration facilities offered by Fractal to switch between a component which implement LRU and one which implement MRU[2].

Such an adaptation policy could look like this:

1. ***When*** returning from the invocation of the cached method, ***if*** the last N requests were consecutive, ***then*** unbind the LRU component and replace it with the MRU component.
2. ***When*** returning from the invocation of the cached method, ***if*** the last N requests were *not* consecutive, ***then*** unbind the MRU component and replace it with the LRU component.

3.4 Conclusion & Evaluation

In this section, we have shown how our approach to building self-adaptive applications makes it possible to take a simple component-based application written with only business concerns in mind and make it self-adaptive transparently, adding the required adaptation logic well encapsulated in adaptation policies.

[2] The state transfer between the two replacement policy components is currently handled in an ad hoc way, but we are investigating more generic solutions.

Although these adaptation policies are currently written in pure Java, we have already achieved complete separation between the core, business concern of the application and its adaptation to the limits and evolution of its execution context.

4 Related Work

Apart from Fractal, several component models have been developed to extend the Java language. Some of them like Jiazzy [8] and Java Layers [14] are purely static (compile-time), and hence can not support dynamic reconfiguration. ArchJava [9], on the other hand, is a Java extension which supports dynamic reconfigurations, but is not reflective and thus supports only reconfigurations which have been written explicitly at development time. As for the EJBs, the model does not really support the notion of architecture, and a previous experiment [15] showed us than the model was too rigid and restrictive to support the kind of reconfigurations we want. The CORBA Component Model supports all the features we need, but is much more complicated than the others. Although our approach could be ported to this model, its complexity makes it difficult to use as an experimentation platform.

In [11], Capra describes a middleware architecture which uses reflection and context-awareness to support adaptation to changing context. As in our approach, this architecture encapsulates adaptation decisions in XML-based *user profile* (similar in intent to our adaptation policies), and relies on a hierarchical model of the context to take decisions. However, these decisions can be based only according to the immediate state of the context, whereas our approach allows the definition of composite events to reason on the evolution of the context over time. Also, reconfiguration actions are delegated to user code through call-backs, which makes them arbitrary and impossible to analyze.

The QuO (Quality Object) middleware platform [16] uses an approach similar to ours to adapt distributed applications. It uses a sophisticated context-awareness subsystem based on performance models for the application components to compute the expected QoS. The resulting system is very powerful, but heavily biased towards performance-related adaptations, whereas our approach tries to be more generic.

Bergmans et al. present in [17] a middleware architecture inspired by control theory to deal with QoS issues in the platform itself. In this architecture, QoS contracts are attached to bindings between components. Sensors are then used to observe the performance of components. The system compares this measured QoS to what is allowed by the contract, and if it detects a difference, reconfigures the system using actuators. The general idea is close to ours, but the system only deals with the quality of communications between components (contracts are attached to bindings).

5 Conclusion: Current Status & Future Work

In this paper, we have presented our approach to enable systematic development of self-adaptive component-based applications. The approach follows the

Separation of Concerns principle, where the adaptation logic of an application is developed separately from the rest of it. It relies on the use of an appropriate component model (in our case, Fractal) to develop an application that is *adaptable*. This adaptable application is then supplemented by *adaptation policies* which capture the adaptation logic in an appropriate formalism (Event – Condition – Action rules). The application is executed inside a software platform which interprets these policies at run-time, using a generic *context-awareness* service to detect changes in the environment, and dynamically reconfigures the application when appropriate. We have also shown, using an example application, how this approach can be used in practice to adapt transparently an application.

Currently, we have a working implementation of the extension we designed for Fractal, and we are implementing the context-awareness service. Next, we plan to do more experiments to guide the design of the DSL we want to provide to define adaptation policies. Once we have a formal definition of this language, we will use it to ensure that the reconfigurations do not break the application, and to detect conflicts between policies leading (incompatible actions triggered by the same events, rules interactions creating unstable behavior...).

References

1. Dowling, J., Cahill, V.: The K-Component architecture meta-model for self-adaptive software. In Yonezawa, A., Matsuoka, S., eds.: Proceedings of Reflection 2001, The Third Int. Conference on Metalevel Architectures and Separation of Crosscutting Concerns, Kyoto, Japan. LNCS 2192, Springer-Verlag (2001) 81–88
2. Hürsch, W., Lopes, C.V.: Separation of concerns. Technical Report NU-CCS-95-03, Northeastern University, Boston, Massachusetts (1995)
3. Coupaye, T., Éric Bruneton, Stéfani, J.B.: The fractal composition framework. Technical report, The ObjectWeb Group (2002)
4. 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
5. The First International Workshop on Unanticipated Software Evolution. In: ECOOP 2002, Malaga, Spain (2002) `http://www.joint.org/use2002`.
6. Kiczales, G., des Rivières, J., Bobrow, D.G.: The art of the Meta-Object Protocol. MIT Press (1991)
7. Dittrich, K.R., Gatziu, S., Geppert, A.: The active database management system manifesto: A rulebase of a ADBMS features. In: Proceedings of the 2nd Int. Workshop on Rules in Database Systems. Volume 985., Springer (1995) 3–20
8. McDirmid, S., Flatt, M., Hsieh, W.C.: Jiazzi: New-age components for old-fashioned Java. In Northrop, L., ed.: OOPSLA'01 Conference Proceedings, Tampa Bay, Florida, USA, ACM Press (2001) 211–222
9. Aldrich, J., Chambers, C., Notkin, D.: ArchJava: Connecting software architecture to implementation. In: International Conference on Software Engineering, ICSE 2002, Orlando, Florida, USA (2002)
10. Mulet, P., Malenfant, J., Cointe, P.: Towards a methodology for explicit composition of metaobjects. In: Proceedings of OOPSLA'95. Volume 30 of ACM SIGPLAN Notices., austin, Texas, USA (1995) 316–330

11. Capra, L., Emmerich, W., Mascolo, C.: Reflective middleware solutions for context-aware applications. In Yonezawa, A., Matsuoka, S., eds.: Proceedings of Reflection 2001, The Third International Conference on Metalevel Architectures and Separation of Crosscutting Concerns, Kyoto, Japan. LNCS 2192, Springer-Verlag (2001) 126–133
12. Consel, C., Marlet, R.: Architecturing software using a methodology for language development. In: Proceedings of the 10th Int. Symposium on Programming Languages, Implementations, Logics and Programs PLILP/ALP'98, Pisa, Italy (1998)
13. Glass, G., Cao, P.: Adaptive page replacement based on memory reference behavior. In: Proceedings of ACM SIGMETRICS 1997. (1997) 115–126
14. Cardone, R., Batory, D., Lin, C.: Java layers: Extending java to support component-based programming. Technical Report CS-TR-00-11, Computer Sciences Department, University of Texas (2000)
15. Jarir, Z., David, P.C., Ledoux, T.: Dynamic adaptability of services in enterprise JavaBeans architecture. In: Seventh International Workshop on Component-Oriented Programming (WCOP'02) at ECOOP 2002, Malaga, Spain (2002)
16. Zinky, J., Loyall, J., Shapiro, R.: Runtime performance modeling and measurement of adaptive distributed object applications. In Meersam, R., et al, Z.T., eds.: On the Move to Meaningful Internet Systems 2002: CoopIS, DOA, ODBASE 2002. Volume 2519 of LNCS., Irvine, California, USA, Springer-Verlag (2002) 755–772
17. Bergmans, L., van Halteren, A., Pires, L.F., van Sinderen, M., Aksit, M.: A QoS-control architecture for object middleware. In: IDMS'2000 Conference Proceedings. Volume 1905 of LNCS., Springer Verlag (2001) 117–131