Experiences from Developing a Component Technology Agnostic Adaptation Framework

Eli Gjørven[1], Frank Eliassen[1,2], and Romain Rouvoy[2]

[1] Simula Research Laboratory,
P.O.Box 134, 1325 Lysaker, Norway
`eligj@simula.no`
[2] University of Oslo, Dept. of Informatics,
P.O.Box 1080 Blindern, 0314 Oslo, Norway
`frank@ifi.uio.no`, `rouvoy@ifi.uio.no`

Abstract. Systems are increasingly expected to adapt themselves to changing requirements and environmental situations with minimum user interactions. A challenge for self-adaptation is the increasing heterogeneity of applications and services, integrating multiple systems implemented in different platform and language technologies. In order to cope with this heterogeneity, self-adaptive systems need to support the integration of various technologies, allowing the target adaptive system to be built from subsystems realized with different implementation technologies. In this paper, we argue that state-of-the adaptation frameworks do not lend themselves to ease technology integration and exploitation of advanced features and opportunities offered by different implementation technologies. We present the QuA adaptation framework and its support for technology integration and exploitation. Unlike other adaptation frameworks the adaptation framework of QuA is able to exploit a wide range of adaptation mechanisms and technologies, without modification to the adaptation framework itself. As a demonstration of this property of QuA, we describe the integration of an advanced component model technology, the Fractal component model, with the QuA framework. Our experience from this exercise shows that the QuA adaptation framework indeed allows integration of advanced implementation technologies with moderate effort.

1 Introduction

Increasingly dynamic computing environments require software developers to support a wider range of technologies with applications that need to handle continually evolving situations and environments. Well designed *component models* enforce *separation of concerns*, thus relieving application developers from having to address concerns, such as extensibility, distribution, and reconfiguration of the application, and letting them focus on business and application logic. In order to ease the tasks of system developers and administrators, separation of concerns can be supported by a *generic adaptation framework* for handling self-adaptation of applications and services [1,2,3]. Self-adaptation includes the ability to self-configure automatically and seamlessly according to higher-level policies. By the same approach, the application developer can model a set of components and their non-functional properties, and leave it to an underlying middleware to reason about changes in context and how these changes should impact and possibly reconfigure the application components to provide the optimal end-user satisfaction with the service. This way, adaptive behavior is developed separately from the application business logic.

However, a challenge for self-adaptation is the increasing heterogeneity of applications and services, integrating multiple systems implemented in different platform and language technologies [4,5,6]. In order to cope with this heterogeneity, self-adaptive systems need to support *technology integration*, which is the process of building a system from subsystems technologies. Successful technology integration includes overcoming three challenges: *i)* ensure that the *integrated subsystems are able to interoperate safely*, *ii) integrate into the adaptation framework the different technologies* used in the system to be adapted, and *iii)* whenever possible *exploit the specific features and opportunities* offered by the different implementation technologies used. The latter is generally preferable as it will reduce duplication of efforts when using advanced implementation technologies, such as state-of-the-art component platforms. Recently, much effort has been spent on interoperability, in particular in the area of *Service-Oriented Architectures* (SOA) [7] and web-services composition [8,9]. However, SOA focus on solving the first problem by standardizing the interaction between services, thus hiding the service implementation platforms. Consequently, SOA does not facilitate the exploitation of adaptation-related features that service implementation platforms may provide. Thus, adaptation techniques are limited to the specification and orchestration of workflows through dedicated languages and engines.

In this paper, we focus on the second and third problems, namely *technology integration* and *exploitation*, in the context of self-adaptive systems. In order to fully support technological heterogeneity, self-adaptive systems must support integration and exploitation also of adaptation-related technologies, while ensuring that the resulting system as a whole performs as expected. In order to be applicable to applications and adaptation mechanisms implemented with different technologies, the adaptation framework of a self-adaptive system needs to be *technology agnostic*. It must be able to adapt the behavior of applications and services without depending on knowledge of particular adaptation mechanisms, and application implementation technologies. In contrast, current adaptation frameworks are bound to particular adaptation mechanism technologies, such as component models, middleware, or communication infrastructures [2,3]. Integrating new technologies into these adaptation frameworks may require major changes to be made both to the integrated systems and the framework itself. Furthermore, the resulting system may not be able to exploit the specific capabilities of the integrated technology. Such a tight coupling between the adaptation framework and the adaptation mechanisms does not facilitate an easy technology integration.

This paper describes the QuA adaptation framework and its support for technology integration and exploitation. As a demonstration of the latter, we describe our experience with integrating an advanced component model technology, the FRACTAL component model [10], with the QuA framework. Unlike other adaptation frameworks, the adaptation framework of QuA is able to exploit a wide range of adaptation mechanisms and technologies, without modification to the adaptation framework itself. In order to establish a clear separation between the adaptation framework and the adaptation mechanisms, we apply the *Dependency Inversion Principle* [11] to the QuA architecture. Under this principle, higher level policies do not depend on the modules implementing the policies, but rather on abstractions. Specifically, by expressing adaptation policies as *utility functions* [12], we enable the specification of adaptation policies that are independent from the technologies used to implement the adaptation actions enforcing the policies. Furthermore, rather than defining yet another component model, the QuA adaptation framework defines a concise, technology-agnostic, meta-model that abstracts over the various legacy component models, which can be plugged in to the adaptation framework.

In the remaining of the paper, we first study the requirements that self-adaptive systems must satisfy in order to facilitate easy technology integration, and we introduce the design principles that support these requirements (cf. Section 2). These principles are demonstrated through the QuA adaptation framework design (cf. Section 3), and the integration of the FRACTAL component model (cf. Section 4). We discuss the experiences made from this integration (cf. Section 5) before concluding and presenting our future work (cf. Section 6).

2 Technology Integration and Adaptation Frameworks

This section analyzes the challenges of designing an adaptation framework supporting technology integration and further motivates the need for clearly separating the adaptation concerns. Then, the main design principles adopted for achieving a better separation of adaptation concerns in the QuA framework are introduced.

2.1 Limitation of Technology Integrations in Self-adaptive Systems

Conceptually, a self-adaptive system consists of three parts: the *adaptation framework*, the *adaptation mechanisms*, and the *target adaptive system*.

The *adaptation framework* (also known as control loop) is responsible for controlling the ongoing adaptation processes. The adaptation framework constantly observes and analyzes the behavior of the target adaptive system, and instantiates, plans, and executes adaptations when necessary. The adaptation framework is based on adaptation policies, used to decide which adaptation to carry out in each situation. The adaptation framework depends on *adaptation mechanisms*, which perform adaptation related actions, such as collecting and processing information about the *target adaptive system* and its environment, evaluating alternative adaptation actions, and performing the selected ones. Examples of such mechanisms are context monitoring, component lifecycle handling, and reconfiguration mechanisms.

The *target adaptive system* represents the target of adaptation. The adaptive system spectrum covers application software, middleware infrastructure (*e.g.*, communication, transaction, persistence), lower level operating system modules (*e.g.*, scheduler, driver), or device resources (*e.g.*, screen resolution, network interface).

The current direction in self-adaptive software research is to isolate the adaptation concerns from the application logic using generic adaptation frameworks [2,3]. However, state-of-the-art adaptation frameworks and corresponding adaptation policy specification languages are tailored to specific component models and platforms. The adaptation policy languages, such as SAFRAN [2] or PLASTIK [3], can be used to define both coarse-grained adaptations, such as replacing one component with another, and more technical and fine-grained adaptations, down to the level of setting the value of a component parameter. These adaptation frameworks impose a tight coupling between the adaptation policies—stating what adaptations should be carried out and when—and the adaptation mechanisms—implementing the corresponding adaptation actions. Typically, the adaptation policy refers directly to the adaptation actions themselves.

Actually, the integration of a new technology can have the following impacts:

→ integration of adaptive systems requires porting the target adaptive application or service to the technology platform of the adaptation framework, and to integrate associated adaptation mechanisms into the framework;
→ integration of new adaptation mechanisms requires updating the adaptation framework with knowledge about the new mechanisms;
→ updating the adaptation framework requires careful evaluation of the effects that the updates will have on other mechanisms and adaptive systems controlled by the adaptive systems.

Thus, a possible, and unfortunate, consequence of the above dependencies may be that adding a new component to a target adaptive system requires updating the higher level adaptation policies. In order to overcome the above challenge, design principles for building technology-agnostic adaptation frameworks are needed. Technology agnostic adaptation frameworks preserve the technological heterogeneity of the target systems, while exploiting adaptation-related features provided by their implementation platforms. We argue that to achieve the above, separation of adaptation concerns should be enforced when designing and implementing the adaptation behavior. In particular, by handling the three parts as separate concerns, we are able to reduce the dependencies between them, and thereby facilitate the integration of new solutions in each concern with less impact on the others.

2.2 Providing a Clear Separation of Adaptation Concerns

In the area of agile programming, the *Dependency Inversion Principle* (DIP) has been introduced as a fundamental design principle, which contributes to improving software maintainability and extendability [11]. The DIP can be applied to systems where higher level modules, containing the important policy decisions and business models of an application, controls lower level modules, containing the implementation of the higher level policies. Thus, according to this principle:

a) high level modules should not depend on low level modules. Both should depend upon abstractions;
b) abstractions should not depend upon details. Details should depend upon abstractions.

We apply the DIP to the case of an adaptation middleware consisting of a higher level module, the adaptation framework containing the adaptation policies and lower level modules, containing the adaptation mechanisms.

In [11], the author points out two consequences of applying the DIP. The first, and most obvious, consequence is that no implementation class should depend on another implementation class, but rather on abstractions. The second consequence is that the abstractions should be owned by the higher level policies, rather than the lower level implementations. From this, we formulate the following requirements for the design of the adaptation framework:

1. the *adaptation framework and the adaptation mechanism should depend on adaptation mechanism abstractions* (according to DIP *a)*),
2. the *adaptation mechanisms and the adaptation target should depend on adaptation target abstractions* (according to DIP *a)*),
3. the *adaptation mechanism abstraction should be owned by the adaptation framework* rather than the mechanisms (according to DIP *b)*),
4. the *adaptation target abstraction should be owned by the adaptation mechanism*, rather than the adapted system (according to DIP *b)*).

Figure 1 illustrates the design of an adaptation framework that satisfies the DIP principle. The modelling convention used here, and in the rest of the paper, is that closed arrows represent an implementation relationship from the implementing class to the interface, while open arrows represent associations and dependencies.

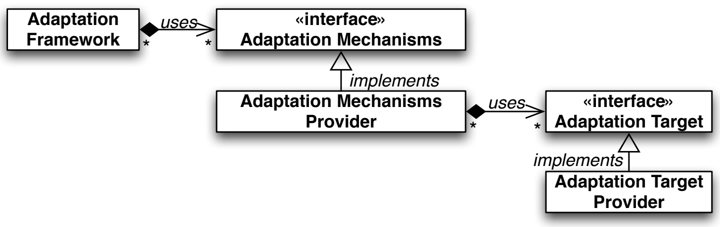
Adaptation Framework
uses
«interface» Adaptation Mechanisms
implements
Adaptation Mechanisms Provider
uses
«interface» Adaptation Target
implements
Adaptation Target Provider

Fig. 1. Applying dependency injection principle to adaptation frameworks.

Many adaptation frameworks satisfy the requirements 1, 2, and 4 [2,3,13]. They typically define adaptation-related interfaces that must be implemented by target systems in order to conform to the adaptation mechanisms. However, as discussed in Section 2.1, these frameworks use adaptation policies that tightly couple the adaptation framework and the mechanisms, making the adaptation mechanism abstraction not truly owned by the adaptation framework as specified by requirement 3. Below, we discuss how to design adaptation policies, making possible to fully satisfy the DIP.

2.3 Using Technology-independent Adaptation Policies

The adaptation framework depends on an adaptation policy, which is applied to decide which mechanism to use in a certain situation. Many adaptation frameworks are based on variants of *rule-based adaptation policies* [2,3], where policies are specified using condition-action expressions. Rule-based approaches can be simple and practical, at least as long as the rule-set is small. However, adaptation rules do not separate well between the adaptation framework and adaptation mechanism. Rules map adaptation conditions directly to detailed knowledge about the target adaptive system, and the available adaptation mechanisms. When integrating new adaptation mechanisms into the rule-set, at best, new rules have to be added to the rule-set. In order to keep the rule-set consistent, then the entire rule-set has to be checked for completeness (all conditions map to an action) and conflicts (conditions mapping to multiple actions that are contradictory). At worst, the policy language is not expressive enough for the new mechanism. For example, the policy language designed for supporting the capabilities of a given component model, may not be directly applicable to another component model. The essential problem is that the rule-based policy languages are owned by the mechanisms, producing dependencies that are difficult to handle when integrating new technologies.

Utility-based adaptation policies have been elaborated as an alternative to rule-based policies in self-managing systems [12]. Utility-based policies are expressed as functions assigning to each configuration alternative—including adaptation mechanisms necessary to implement the alternative—a scalar value indicating the desirability of this alternative. A utility-based adaptation framework discovers a set of configuration alternatives, computes their utility, and selects the one with the highest utility. This way, utility functions introduce a level of indirection between the adapted system and the mechanisms implementing a configuration, and its desirability. The utility value is calculated from metadata describing the functional and qualitative properties of a configuration, rather than the technical implementation knowledge. Thus, utility functions provide a higher-level, mechanism and technology independent adaptation policy language. The reader can refer to [14] for a detailed discussion about the characteristics of utility functions.

2.4 Reflecting the Target Adaptive System Properties

In order to be able to compute utility values, metadata about the functional and qualitative properties of configuration alternatives and adaptation mechanisms must be available to the adaptation framework. Thus, the adaptation framework depends on a technologically independent meta-model that is able to express information about the required properties.

In order for the adaptation framework to be independent of the existence of particular reflective capabilities provided by target technologies, the metadata must be provided by a separate module. A variant of traditional reflection, called *mirror-based reflection* [15], can be used to define reflective APIs suitable for technologically independent adaptation frameworks. In mirror-based reflection, the reflective capabilities are provided by separate objects called *mirrors*, instead of by the reflected objects themselves, as is common in traditional reflection. The reader can refer to [15] for a detailed discussion about the characteristics of mirror-based reflection.

3 Designing the QuA Adaptation Framework

This section introduces the design of the QuA adaptation framework, which proposes to improve the state-of-the-art adaptation middleware approaches by offering a modular support for reflecting, reasoning, and deploying services.

3.1 An Overview of the QuA Middleware

The QuA middleware supports middleware-managed adaptation, which means that the adapted system is specified by its behavior, and then *planned*, *instantiated*, and *maintained* by the middleware in such a way that its functional and qualitative requirements are satisfied throughout its lifespan. In order to be able to represent the adapted system from specification to termination, the unit of adaptation in QuA is a *service*, which we define as:

> A service describes a set of capabilities that are defined by *i)* a group of *operations* and their *input and output data*, and *ii)* a *contract* (explicit or implicit) describing the work done, as delivered output data, when invoking these operations with valid input data. The *service lifespan* encloses its specification of behavior, association with implementation artifacts, service instantiation, execution, and termination.

Thus, a service may be associated with implementation artifacts implementing its behavior, or running objects performing its behavior. Service implementation artifacts always require a particular *service platform*, which can be used to instantiate a service by interpreting the implementation artifacts. Finally, service implementations may depend on other services in order to implement the promised functionality.

A QuA client is typically a client application, using QuA to instantiate services, or service a development tool, using QuA to deploy service implementations and meta-data. QuA defines a programming API that can be used to invoke the QuA middleware services from tools or applications, and providing the following operations:

- *Publication of service implementations*: service implementations may include different types of implementation resources, such as implementation classes (Java classes or library modules), component descriptors (ADL or XML documents), interface definitions etc., depending on the type of technology used to implement the service.
- *Advertising service implementation meta-data*: Meta-data describing the static and dynamic properties of service implementations can be advertised to the middleware.
- *Instantiation of services*: service instantiation means evaluating, selecting, and instantiating service implementations, and perform initial service configuration. The resulting service will be maintained by the QuA middleware throughout its lifespan through adaptation.
- *Reflection on services*: the QuA middleware defines a reflective API, called the *Service Meta Object Protocol* (SMOP), used to inspect and manipulates services.

In contrast to other adaptation frameworks, which mixes the adaptation policies with the adaptation mechanisms, QuA identifies a clear separation between the three adaptation concerns described in Section 2.

Conceptually, we order the three adaptation concerns horizontally, as depicted in Figure 2. By applying the *Dependency Injection Principle* (DIP), we achieve an horizontal separation of concerns by establishing an ordering of module pairs where the higher level module always owns the interfaces shared with next lower-level modules.

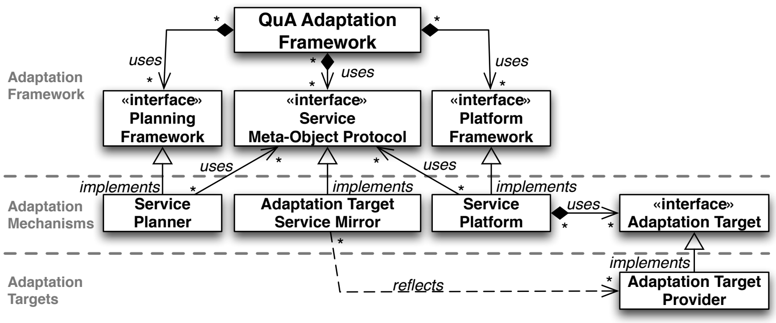
QuA Adaptation Framework
uses
uses
uses
Adaptation Framework
«interface» Planning Framework
«interface» Service Meta-Object Protocol
«interface» Platform Framework
uses
uses
implements
implements
implements
Adaptation Mechanisms
Service Planner
Adaptation Target Service Mirror
Service Platform
uses
«interface» Adaptation Target
implements
Adaptation Targets
reflects
Adaptation Target Provider

Fig. 2. Design of the QuA adaptation framework.

The adaptation framework module define three abstractions; The Planning Framework is responsible for selecting service implementations, while the Platform Framework is responsible for managing service implementations during their execution. The Service Meta-Object Protocol (SMOP) can be used to inspect and manipulate services throughout their lifespan.

The planning and platform frameworks abstractions are implemented by concrete planning and implementation mechanisms. The planning framework is implemented by Service Planners that use metadata provided by the SMOP to find alternative service implementations, analyze their expected behavior, and select an alternative that match the service requirements. The platform framework is implemented by Service Platforms that enclose technology specific code and mechanisms supporting service instantiation and adaptation, including binding and rebinding of service dependencies. Such adaptation mechanisms typically define adaptation-related interfaces implemented by the target systems. Service Platforms are responsible for maintaining the causal connection between Service Mirrors implementing the SMOP, and the Adaptation Targets. In the adaptation target layer, we find the Adaptation Target Providers, which are the base level objects implementing the adaptation targets.

3.2 Reasoning Support: the Utility-based Planning Framework

The planning framework applies *utility-based adaptation policies* [12] as a way to keep the adaptation framework independent of integrated technologies and mechanisms. Each service may be associated with a *utility function*, which is applied by the planning algorithm to metadata describing the qualitative properties of each alternative service implementations. Metadata about the qualitative properties of a service implementation can be expressed by *quality predictors*, which are functions of the run-time environment, and the quality provided by other services that the service depend on, if any. Such predictor functions are written by the implementation developer, and made available through the SMOP. By computing utility functions and quality predictors, the utility of a particular implementation can be calculated based on the desirability of alternative behaviors, rather than knowledge about the alternative implementations and mechanisms.

The planning process can be implemented by numerous algorithms. By applying the DIP also to the planning framework, the adaptation framework is protected from changes in the mechanisms used by the planning framework.

3.3 Technology Support: the Platform Framework

When an implementation has been selected by the planning framework, the platform framework is responsible for applying the correct mechanisms for instantiating the service. A service platform is able to interpret implementation artifacts of certain types, instantiate services from those artifacts, and provide a run-time environment for the instantiated services. For example, a Java Service Platform provides access to a *Java Virtual Machine*, and is able to instantiate Java objects hosted by that machine, from Java classes. The platform also defines the types and natures of service collaborations defined by the technology, such as component composition through component connectors, or specialized communication patterns, such as event-driven communication and data streaming.

Upon service instantiation, a service platform receives from the adaptation framework, metadata describing the required service, implementation artifacts that have been selected by the planning framework during initial planning, and services that the implementation depends on. Adaptation-aware platforms monitor their managed services, and when they find it necessary, trigger the adaptation framework for a re-planning. The result from the re-planning is a new set of metadata and implementation artifacts that can be used by the platform to perform an adaptation.

In order to hide the details of service instantiation and configuration from the adaptation framework, we apply the DIP to the platform framework. The adaptation framework invokes a service platform to instantiate a service with a package encapsulating the implementation artifacts, called *blueprint*, as a parameter. As the type of implementation artifacts used by a certain technology is highly technology specific, blueprints are black boxes to the adaptation framework. The blueprint may contain technology specific information related to different types of adaptation mechanisms, such as component replacement, component parametrization, insertion of interceptor or monitors, etc. The QUA adaptation framework does not define the format of a blueprint, nor does it ever inspect or manipulate its content. Blueprints are created by technology expert developers, and deployed to the QUA middleware using technology specific tools.

The platform depends on technology specific mechanisms in order to instantiate and adapt the service. Examples of such services are component factories, parsers for component descriptors, binders and configurators, resource managers, etc. These mechanisms may either be implemented as a part of the platform, or they may be deployed to the middleware as services that the platform depends on. Based on the simple abstractions described above, multiple adaptation techniques can be integrated and exploited through the platform framework concept [2,3,4,5,6,13].

3.4 Reflection Support: the Service Meta-Object Protocol

The QuA middleware defines a service meta-object protocol that can be used to reflect on services in all phases of their life-cycle. The SMOP is based on a services meta-model, which is used to describe exactly the aspects of a service related to planning, instantiation, and execution of services managed by QuA—*i.e.*, its behavior (required or provided type, and utility function), implementation (including blueprint, required service platform, and implementation dependencies), and instances if any. Figure 3 describes the QuA service meta-model.

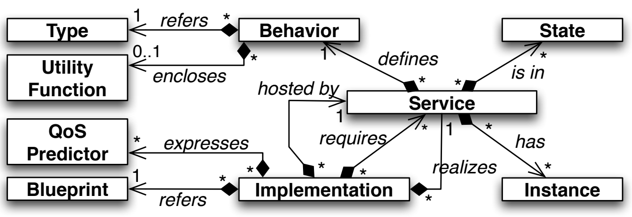

Fig. 3. The QuA service meta-model.

In order to conform to the DIP, the SMOP must provide the adaptation framework with a technologically independent reflective API. The QuA reflective API is based on mirror reflection, where the meta-level facilities are implemented separately from the reflected system as described in Section 2.4. Thus, mirror-based reflection does not require any changes to be made to the reflected system, and it allows the coexistence of technology specific reflective APIs required by service platforms and their mechanisms. In [16], we describe a comprehensive application scenario demonstrating the application of the service meta model, including examples of quality prediction and utility functions.

4 Implementing the QuA-Fractal Adaptation Middleware

In [14], we have shown that the framework is applicable to simple programming models, such as the Java programming languages, by designing lightweight component models based on this language. In order to confirm the ability of the QuA adaptation framework to integrate and exploit concrete adaptation technologies, we need to apply the framework to an adaptation technology that provide a rich set of features. To this end, we consider the FRACTAL component model [10] as an interesting candidate technology. The FRACTAL component model is a lightweight and hierarchic component model targeting the construction of efficient and highly reconfigurable middleware systems. FRACTAL has been used in several projects to implement advanced adaptive and self-adaptive behavior [17,2,18]. Thus, if we are able to successfully integrate and exploit the rich set of available mechanisms and tools provided by FRACTAL ecosystem, without coding FRACTAL-specific knowledge into the generic adaptation framework, it is a strong indication that the QUA adaptation framework has the expected capabilities with regards to supporting integration.

The work presented in this paper is based on integrating the powerful, expressive, and flexible component reconfiguration mechanisms provided by FRACTAL. In particular, the *FractalADL Factory* is a component factory that instantiates FRACTAL components and composites from architecture descriptions written in the FRACTAL *Architecture Description Language* (FRACTALADL) [19]. The FSCRIPT engine interprets configuration scripts written in the FRACTAL-based configuration language FSCRIPT [2]. The FSCRIPT language includes primitives for standard FRACTAL component management, and can be extended in order to support more advanced configurations.

4.1 The FRACTAL Component Model

The reconfiguration capabilities are defined by controllers that defines the level of introspection and control of a component (life-cycle, attributes, bindings, interfaces, etc.).

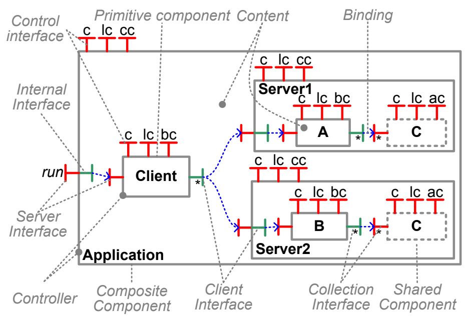

Fig. 4. Architecture of a FRACTAL component.

Figure 4 illustrates the different entities in a typical FRACTAL component architecture. Thick black boxes denote the *controller part* of a component, while the interior of these boxes correspond to the *content part* of a component. Arrows correspond to *bindings*, and tau-like structures protruding from black boxes are internal or external interfaces. Internal interfaces are only accessible from the content part of a component. External interfaces appearing at the top of a component represent reflective control interfaces, such as the *Life-cycle Controller* (lc), the *Binding Controller* (bc) or the *Content Controller* (cc) interfaces. The two dashed boxes (C) represent a *shared* component.

4.2 The QuA-Fractal Middleware

The QuA-Fractal middleware has been implemented as a *service platform*, Fractal Platform, that includes an implementation of the Julia run-time, the FractalADL factory, and the FScript engine, as illustrated in Figure 5.

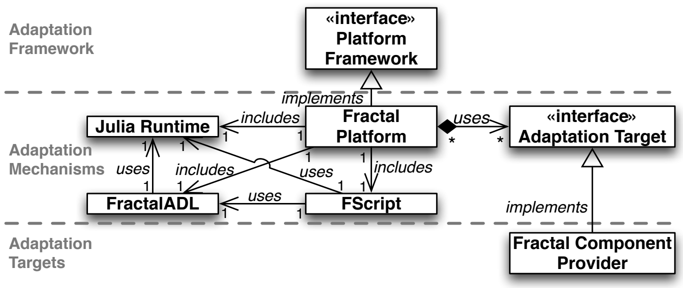

Fig. 5. Architecture of the Fractal platform.

The Fractal platform instantiates services from Fractal blueprints, containing FractalADL descriptors, implementation classes, and FScript configuration scripts. It extracts ADL descriptors and implementation classes from the blueprint, and invokes the ADL factory to instantiate components from the descriptors. The ADL factory depends on the Julia run-time to create the component instances from the implementation classes. Finally, FScripts are extracted from the blueprint, and the FScript engine is invoked to perform configuration based on the FScripts. FractalADL and FScript use standard Fractal controllers to perform component management tasks, such as binding, life-cycle management, and parameter configuration.

In order to be able to exploit different combinations of Fractal components, we have to enable the QuA planning framework to plan alternative Fractal components independently. For example, in the case of composite components, we want to be able to plan certain sub-components independently, in order to find the combination of components that best satisfy the service requirements. The recursive meta-model provided by QuA enables such a nested planning through the definition of implementation dependencies. Instead of publishing an ADL descriptor describing the complete composition, we extract ADL descriptions describing sub-components into separate Fractal blueprints. Thus, in the case where several implementations of a sub-component are available, the planner will select the one giving the highest utility.

4.3 The *Comanche* Application

Below, we illustrate our prototype application *Comanche*: a legacy web server developed by the Fractal community. *Comanche* is a lightweight web server implemented with the Fractal component model[3]. This implementation provides the core features

[3] *Comanche* tutorial: `http://fractal.ow2.org/tutorial`.

of a web server as a proof of concept of the relevance of FRACTAL for building middleware systems.

In its initial version, *Comanche* is made of several components that identify the various concerns of a web server, as depicted in Figure 6.

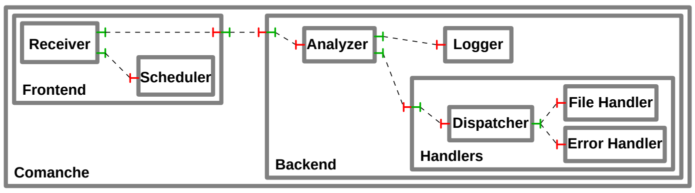

Fig. 6. Architecture of the *Comanche* web server.

In particular, *Comanche* contains a component Scheduler that schedules the treatment of incoming requests (see Figure 6). The implementation of this component controls the allocation of dedicated activities for analyzing incoming requests. The initial implementation of the scheduler creates a thread per incoming request without controlling the number of activities created. In the SAFRAN project [2], an alternative to the scheduler proposes to use a pool of threads to control the number of threads used by the web server. However, the thread pool scheduler provides lower response times than constantly creating new threads.

```
ServiceMirror comancheMirror = QuA.createServiceMirror( Comanche );
comancheMirror.setUtilityFunction( ComancheUtility );
comancheMirror.setImplBlueprint( ComancheBlueprint );
comancheMirror.setImplQoSPredictor( ComanchePredictor );
comancheMirror.setImplDependencies( "s", Scheduler );
```

Listing 1.1. Service mirror reflecting the *Comanche* web server.

In order to be able to apply the QUA-FRACTAL middleware to the configuration of the *Comanche* web server, we had to deploy the *Comanche* application, including ADL descriptors, FScripts, and implementation classes, to the QUA middleware as a blueprint, and to advertise the necessary meta-data, including utility function and quality predictors, to the middleware (see the QuA-specific code for advertising metadata, represented as a service mirror in Listing 1.1).

```
<definition name="Frontend" extends="FrontendType">
  <component name="rr" definition="Receiver"/>
  <component name="s"  definition="Scheduler"/>
  <!-- Definitions of bindings -->
</definition>
```

Listing 1.2. FRACTALADL descriptor of the *Comanche* front-end.

The ADL for the *Comanche* front-end is depicted in listing 1.2. The utility of the *Comanche* server is expressed by a function that returns high utility values for low response times, and low utility values for high response times. Listing 1.3 contains the component replacement script *replace-scheduler*, used to replace one scheduler component with another. The script uses a number of primitive operators, such as *stop*, *bind*, and *remove*, in order to implement the routine that has to be followed in order to safely replace one component with another. These operators are mapped by FSCRIPT to invocations of standard FRACTAL controller interfaces.

```
action replace-scheduler(comanche, scheduler) {
  stop($comanche);
  var frontend = $comanche/child::fe;
  unbind($frontend/child::rr/interface::s[client(.)]);
  remove($frontend, $frontend/child::s);
  add($frontend, $scheduler);
  bind($frontend/child::rr/interface::s, $scheduler/interface::s);
  start($comanche);
  return $comanche;
}
```

Listing 1.3. FSCRIPT statements replacing component in *Comanche*.

5 Evaluating the QUA-FRACTAL Implementation

In order to evaluate the adaptation framework presented in Section 3, we have to consider to what degree the combined QUA-FRACTAL middleware was able to solve the challenges mentioned in Section 1, namely: *ii)*to *integrate into the adaptation framework different technologies* used in the system to be adapted, and *iii)* to whenever possible *exploit the specific features and opportunities* offered by the different implementation technologies used. With regard to *ii)*, we have managed to integrated the FRACTAL run-time and FRACTAL components into the QUA adaptation framework. The integration required an acceptable amount of work, given the availability of developers that have moderate knowledge about the QUA middleware, and some knowledge about the FRACTAL middleware. With regard to *iii)*, we have managed to exploit two FRACTAL specific adaptation mechanisms, namely the FRACTALADL factory and the FSCRIPT language.

In order to reflect the amount code in the resulting middleware that is technology agnostic, technology specific, and application specific, Table 1 presents the number of classes and the byte-code size of the different parts of the resulting middleware and application. As indicated by the table, the FRACTAL mechanisms contribute with by far the largest amount of files and byte-code. The QuA middleware consists of a rather small middleware core, which byte-code size is less than 10% of the size of QUA-FRACTAL platform. Furthermore, the number of QUA-specific files required in order to implement the QUA-FRACTAL platform was only 8. This number includes both the definition of the QUA-FRACTAL platform interface, the QUA-FRACTAL implementation classes, the QUA-FRACTAL blueprint used to encapsulate FRACTAL implementation artifacts, and an helper platform used to instantiate the QUA-FRACTAL platform itself, as a service.

The relatively small size of the QUA middleware is the result of keeping the responsibility of the QuA middleware small and concise, and independent of technology specific details and knowledge by the application of the DIP and utility functions. Due to these principles, we are able to control an advanced and comprehensive adaptation middleware technology from this small and generic adaptation framework.

Table 1. Distribution of code in QUA-FRACTAL.

Concern	Number of class files	Byte-code size (Kb)	Distribution (%)
QUA middleware	53	276	7
QUA-FRACTAL platform	**8**	**76**	**2**
JULIA run-time	300	1,782	45
FRACTALADL factory	171	816	21
FSCRIPT engine	151	828	21
Utility classes	33	168	4
QUA-FRACTAL total	716	3,946	100
Comanche application	17	76	

6 Conclusions

Due to the growing heterogeneity of technologies used to implement nowadays distributed systems, existing adaptation middleware faces more and more difficulties to perform technology agnostic adaptations. This phenomenon is particularly true in the component-based software engineering community where most of the state-of-the-art approaches to adaptation suffer from their tight coupling to a particular component model [2,3]. This strong dependency restricts the integration of new technologies (*e.g.*, component models or middleware frameworks) to the fixed set of abstractions supported by the adaptation middleware, thus avoiding the integration of technology-specific adaptation capabilities.

The contribution of this paper is to present the implementation of a modular adaptation middleware, called QUA, whose design supports the integration of various technologies. This design combines the definition of a *Meta-Object Protocol* [15] and the application of the *Dependency Inversion Principle* [11]. The former is used to reflect the meta-data associated to technology artifacts, while a *utility-based planning framework* and a *platform framework* apply the latter to reason about the reflected metadata and perform adaptations, respectively. We validate this design by reporting the integration of the FRACTAL component model into the QUA middleware, and we illustrate the resulting adaptation middleware on the adaptation of a component-based application: the *Comanche* web server. By facilitating the integration of technologies, this approach clearly separates the adaptation concern from the application and the technology.

As a matter of perspective, we plan to extend the set of supported technologies and experiment the consistent adaptation of cross technology applications.

Acknowledgements

The authors thank the reviewers of the CBSE conference for valuable comments. This work was partly funded by the European Commission through the project MUSIC (EU IST 035166).

References

1. Oreizy, P., Gorlick, M.M., Taylor, R.N., Heimbigner, D., Johnson, G., Medvidovic, N., Quilici, A., Rosenblum, D.S., Wolf, A.L.: An architecture-based approach to self-adaptive software. IEEE Intelligent Systems **14**(3) (1999) 54–62
2. David, P.C., Ledoux, T.: An Aspect-Oriented Approach for Developing Self-Adaptive Fractal Components. In: 5th International Symposium on Software Composition (SC'06). Volume 4089 of LNCS., Springer (2006)
3. Batista, T.V., Joolia, A., Coulson, G.: Managing Dynamic Reconfiguration in Component-based Systems. In: 2nd European Workshop on Software Architectures (EWSA'05). Volume 3527 of LNCS., Springer (2005) 1–17
4. Sun microsystems: Java Platform, Enterprise Edition (Java EE) `http://java.sun.com/javaee`.
5. OSGi Alliance: OSGi Service Platform Release 4 `http://www.osgi.org`.
6. Microsoft .Net: Microsoft .NET Framework 3.5 `http://www.microsoft.com/net`.
7. Erl, T.: Service-Oriented Architecture: Concepts, Technology, and Design. Prentice Hall (2005)
8. Erradi, A., Maheshwari, P., Tosic, V.: Policy-Driven Middleware for Self-adaptation of Web Services Compositions. In: 7th International Middleware Conference. Volume 4290 of LNCS., Springer (2006) 62–80
9. Kuropka, D., Weske, M.: Implementing a Semantic Service Provision Platform Concepts and Experiences. Journal Wirtschaftsinformatik – Special Issue on Service Oriented Architectures and Web Services **1** (2008) 16–24
10. Bruneton, E., Coupaye, T., Leclercq, M., Quéma, V., Stefani, J.B.: The FRACTAL component model and its support in Java. Software Practice and Experience – Special Issue on Experiences with Auto-adaptive and Reconfigurable Systems **36**(11/12) (2006) 1257–1284
11. Martin, R.C.: Agile Software Development, Principles, Patterns, and Practices. Prentice Hall (2002)
12. Kephart, J.O., Das, R.: Achieving Self-Management via Utility Functions. IEEE Internet Computing **11**(1) (2007) 40–48
13. Georgiadis, I., Magee, J., Kramer, J.: Self-Organising Software Architectures for Distributed Systems. In: 1st International Workshop on Self-Healing Systems (WOSS'02), ACM (2002) 33–38
14. Alia, M., Eide, V.S.W., Paspallis, N., Eliassen, F., Hallsteinsen, S.O., Papadopoulos, G.A.: A Utility-Based Adaptivity Model for Mobile Applications. In: 21st International Conference on Advanced Information Networking and Applications (AINA'07), IEEE (2007) 556–563
15. Bracha, G., Ungar, D.: Mirrors: Design Principles for Meta-level Facilities of Object-Oriented Programming Languages. In: 19th Annual Conference on Object-oriented Programming, Systems, Languages, and Applications (OOPSLA'04), ACM (2004) 331–344
16. Gjørven, E., Eliassen, F., Lund, K., Eide, V.S.W., Staehli, R.: Self-Adaptive Systems: A Middleware Managed Approach. In: 2nd IEEE International Workshop on Self-Managed Networks, Systems and Services (SelfMan). Volume 3996 of LNCS., Springer (2006) 15–27
17. Bouchenak, S., Palma, N.D., Hagimont, D., Taton, C.: Autonomic Management of Clustered Applications. In: International Conference on Cluster Computing (Cluster'06), IEEE (2006)
18. Roy, P.V., Ghodsi, A., Haridi, S., Stefani, J.B., Coupaye, T., Reinefeld, A., Winter, E., Yap, R.: Self-management of large-scale distributed systems by combining peer-to-peer networks and components. Technical Report 18, CoreGRID - Network of Excellence (2005)
19. Leclercq, M., Özcan, A.E., Quéma, V., Stefani, J.B.: Supporting Heterogeneous Architecture Descriptions in an Extensible Toolset. In: 29th International Conference on Software Engineering (ICSE'07), IEEE (2007) 209–219