Patterns for Integrating and Exploiting Some Non-Functional Properties in Hierarchical Software Components

Hervé Chang and Philippe Collet
Université de Nice Sophia Antipolis
Laboratoire I3S, CNRS UMR 6070
2000, route des lucioles Les Algorithmes - bât. Euclide B, BP 121
06903 Sophia Antipolis France

Abstract

Providing powerful and fine-grained capabilities for the analysis and management of non-functional properties is a major challenge for component-based software systems. In this paper, we propose integration patterns for non-functional properties of hierarchical software components. These patterns are based on a classification of low-level non-functional properties, which takes into account their nature and lifecycle. They make explicit the implementation of these properties in relation with components and can be used to develop some forms of compositional reasoning. The proposals are exploited in non-functional contract negotiation by enabling a negotiation process to be precisely propagated down the component hierarchy.

1 Introduction

In component-based software engineering, one of the major challenges is still to facilitate the management of non-functional properties. These properties represent various qualities of software components and systems, such as their runtime qualities or system lifecycle. With the need to manage more complex and long-running component-based systems, identifying and handling these properties as precisely as possible during design, (re-)configurations and runtime phases is crucial. Moreover, in order to reason on the composition of non-functional properties, support to determine properties of component assemblies from properties of individual components must also be provided.

Over the past decades, numerous studies have been conducted around the modeling, analysis and management of non-functional properties. At the highest-level of analysis, some approaches provide methodologies to analyze quality attributes [1], or address non-functional properties through quality standards (IEEE 1061, ISO/IEC-9126) and models [2, 4] that provide classifications of high-level properties. They aim at proposing a generic taxonomy and studying relationships between properties, or structuring knowledge by successively defining and decomposing non-functional characteristics. As these properties mostly remain at a high-level, they are not clearly modeled in relation with runtime components and platforms, and their specification and management is thus limited. On the other hand, at the lowest-level of resource management, substantial works exist on providing resource management and monitoring capabilities to applications, at different level of abstractions (API, technologies) [9]. Such works also aim at integrating advanced tools for the diagnosis of performance issues [18]. Besides, to achieve non-functional requirements, numerous works in the domain of distributed systems propose some component-based middleware platforms that provide QoS control and measurement capabilities through reflective and adaptive systems techniques [12, 16]. They particularly focus on critical network-related properties, and provide integrated control mechanisms without explicit representation of non-functional properties. Some other component platforms also enable flexible integration of arbitrary non-functional services using containers. Containers wrap set of components, and are responsible for handling essentially non-functional middleware-related services (transactions, load balancing, security checks, etc.). They are integrated using code transformation such as aspect-weaving, or indirection frameworks like interceptors and meta-object protocols.

Several compositional approaches, which are specific to component-based systems, aim at improving non-functional property analysis in component assemblies. Analysis models and property theories are thus integrated to component technology [13], and they allow one to guarantee, by construction, the predictability of some properties on compo-
nent assemblies. However, they require advanced analysis models and techniques, and are mostly dedicated to specific properties, such as latency [13], reliability [19] or memory usage [11]. Some of these models could be extended to other properties if the properties are properly related to the architecture and modeled in some generic ways that make possible to reason on them. Recently, to make reasoning on non-functional properties possible at the architectural level, the relationship between software components and software architecture has been outlined [21] and exploited by studying how properties relate to component assemblies and individual component properties. In particular, in order to help describing how properties relate to compositions, an interesting classification [8] have synthesized different class of dependency between properties, components and their context.

Our work stands half-way between analysis techniques and management systems. We propose to reify non-functional properties in relation with components, and to provide means to support a basic form of compositional reasoning on these properties. This should then enable software architects to better master the modeling, integration and also the runtime management of non-functional properties into component-based systems. In this paper, we thus present some simple architectural patterns that model non-functional properties. They are based on a classification of some low-level observable non-functional properties, which is established by considering their nature and lifecycle. The proposed patterns reify different kinds of parameters on individual components, as well as physical resources. As for compositional properties, a reasoning support based on meta-level elements is also described.

The rest of the paper is organized as follows. The next section describes the proposed classification and patterns at an abstract level, as well as the support for compositional properties. The mapping of these patterns to the hierarchical component platform Fractal [5] is described in section 3. Section 4 illustrates the use of the proposed patterns for contract negotiation in Fractal, especially to propagate negotiations down into the component hierarchy. Section 5 concludes this paper and discusses future work.

2 Classification and Modeling of Non-Functional Properties

2.1 Rationale

The proposed classification is not intended to be exhaustive. It is rather limited to the range of low-level properties which are measurable and sufficiently orthogonal to functional aspects. Hence, non-functional aspects which concern high-level properties and system lifecycle at development and maintenance phases, are not taken into account here, as well as other temporal aspects, which require more knowledge about the behavior of components. Moreover, to reason compositionally on non-functional properties, the analysis of non-functional properties of a system must also be based on properties of the components that compose it. Consequently, the classification also takes into account the composability of properties at the level of component compositions. The classification is then built by first analyzing what kinds of non-functional properties can be directly derived from individual components, and what kind of features do they express in relation with them. The lifecycle of properties is also analyzed in relation with the one of components. The moments when these properties are defined are distinguished, as well as when they are to be observed.

The proposed patterns directly match the categories of non-functional properties at an abstract level of architecture specification in order to remain independent from underlying component technologies. They are also used to support automated reasoning on compositional properties, once and for all, at the level of patterns themselves.

2.2 Patterns

We first distinguish three categories of non-functional properties that directly match the concept of component attributes.

Some non-functional properties represent the key features or nature of components. For example, they can describe a memory footprint, the compatibility version number of a video codec or a maximum capacity. These properties are comparable to the technical characteristics of electronic or mechanic components, they capture and represent some design and development features of components and have an impact on their whole usage. They result of choices made when designing and developing components, and are generally taken into account at assembly and configuration times to evaluate the suitability of components, for instance when selecting potential components and matching them to requirements. As they describe the nature of components, they cannot be changed and their measurements remain constant all along configuration and runtime phases. The associated pattern models properties of this first category as read-only private attributes of components, which are accessed only by their associated getter operations defined in a provided interface. This is illustrated by the provided interface named IPropertiesOfNature on the left side of the two UML 2 component diagrams\(^1\) (see figure 1a).

Some other properties represent configurable parameters of components themselves. They describe, for exam-

\(^1\) UML 2 components represent independent, interchangeable parts of a system. They realize one or more provided and required interfaces, which determine the behavior of components. Interfaces define sets of operations that components implement. Attributes can also be added to components to represent data fields or properties about them.
ple, the size of a buffer component, the maximum size of a resource pool component. They influence the required and provided services of components, and may be (re-)parameterized to set components to some specific functioning mode, and adapt them to their runtime environment (other components, runtime infrastructure). They can be defined during the development phase by default values, but they are mostly observed and changed at assembly and (re-)configuration times, to properly customize the functioning mode of components. Their pattern models these properties as both read and write component attributes, with associated getters and setters defined in a provided interface (named IConfigurationParameters in figure 1a).

![Diagram](image)

Figure 1: Overview of patterns.

Others properties describe functioning parameters of components. For example, the number of active sessions on a web server, the current state of a video player processing a media stream, and the current number of packets exchanged between a given client and a server can all be considered as media stream, and the current number of packets exchanged.

As components may exhibit properties that belong to each of the categories, the proposed patterns can be applied together. Moreover, as they are only built upon standard elements of the supporting components, these patterns can be applicable to model non-functional properties on both primitive and composite components.

Regarding the measurement of non-functional properties, the proposed patterns provide standardized way to represent properties at the modeling level, but they do not provide predefined mechanisms to measure them, at the implementation level. Such mechanisms are defined at implementation time when the component technology and the runtime environment are determined. However, some non-functional properties that monitor infrastructure may still exploit these patterns, as they make explicit how non-functional properties of components are exposed and how properties may vary along the component lifecycle. Besides, properties implemented through read-only attributes, such as the general notion of reliability, could be statically computed by statistical measures on a system and then set at configuration times in the appropriate attribute. On the other hand, runtime properties, mainly related to resources, can be simply measured through appropriate resource probes and asynchronous communications support for processing monitoring information, like the ones provided in the DREAM framework [14].

2.3 Enabling Reasoning on Compositional Properties

To be able to reason on the realization of a quantifiable compositional property from other ones, information describing the relationships, as well as some appropriate reasoning support must be provided.

Except resource properties, which do not express quantifiable properties, other properties from our classification have been modeled using patterns that derive directly from individual components. These properties are thus compositional by nature, and some simple form of compositional reasoning can be supported. Building on those primitive parts, we define a compositional property as a property providing the following characteristics: (C1) the set of properties that contribute in decomposing that property, (C2) for
functions can be defined with almost arbitrary code, as these functions are specified with assertion-based contracts (see section 4.1), so that checking can be performed during both testing and exploitation stages.

We thus suppose that for each compositional property, compositional information are provided through descriptive meta-data that may be expressed using code annotations or Domain Specific Languages (DSL) facilities. To enable reasoning on them at runtime, compositional properties are reified as meta-objects. A CompositionalPropertyManager (upper left part of figure 3) is built for each component to manage the set of its compositional properties. It uses the provided compositional information to instantiate and register a CompositionalPropertyObject (meta-object on upper right part) that both reifies each compositional property and gives access to all of its compositional information: its value, all other properties necessary to compute its composition function and their contributing components. At runtime, the CompositionalPropertyManager is exploited by elements at the base or at the meta level to retrieve the corresponding CompositionalPropertyObject and get the compositional information about the property it reifies. For example, to get the value of a property (action 1 on figure 3), the implementation of the component function at the base level retrieves the compositional property object from the property manager (2 and 3) and then gets the current value computed from the compositional function (4 and 5). Other compositional information can be retrieved similarly.

3 Mapping to the Fractal Component Platform

Fractal [5] is a modular and extensible component model with different implementations. It supports composite and shared components (components can be formed from other components and contained in several distinct components), reflection (the execution of components can be observed), and reconfiguration capabilities (component instances can be deployed, removed and replaced dynamically). Fractal

\[5\]
components are runtime entities that communicate through provided and required interface bindings. Technical aspects can be integrated into components through controllers, and some basic controller interfaces are provided to manage interface bindings, lifecycle and content of components (respectively described as BC, LC and CC in figure 4).

3.1 Working Example

We use as a working example an instant messaging server that automatically groups users into chat rooms, and streams videos to grouped users according to their interests. This application is already operational and has been developed using both Fractal components and web services. The server, shown on figure 4, is represented by the composite component FractalInstantCommunication, which is formed out of three sub-components: InstantGroup manages the users and their grouping through its provided interface UserMgmt\(^3\). VideoService manages the video streaming service, and BdwMonitor monitors the network bandwidth and measures the overall bandwidth consumption of the server. InstantGroup is composed of UserManager which manages the users, GroupManager which manages groups, MsgMonitor which monitors messages exchanged between users, and InstantFacade which pilots the other components. InstantGroup also exhibits the properties maxUsers and groupedUsersRatio which, respectively, describe the maximum number of concurrent users that the server supports, and the rate of users that have been grouped. VideoService is composed of VideoManager which manages the video streaming, and VideoMonitor which monitors the bandwidth consumption of the video service. The content of BdwMonitor is detailed later.

3.2 Integrating Patterns

The Fractal component model also proposes control interfaces for attributes in order to model orthogonal properties of components. They give access to component attributes before starting a component and without needing to bind and use its functional interfaces. Moreover, these interfaces offer various access modes (read and/or write accesses), which make possible to respect the difference between properties of nature and functioning properties, which cannot be changed, and configuration parameters which can be modified. Hence, as properties of nature, configuration parameters and functioning parameters of components, are modeled through component attributes, they are basically mapped to Fractal attributes and attributes control interfaces (see figures 5a and 5b), with their appropriate read-only or read-and-write operations. As resources are modeled by components, they are reified using full-fledged Fractal components, which do not provide a priori advanced services, except for probing their associated physical resource. Moreover, as resources can be used by several distinct components, component sharing is exploited to reflect resource sharing at the component composition level (see figure 5c). It enables one to use a same instance of a reified resource component, in several distinct enclosing components, at different level of hierarchy, while preserving component encapsulation.

\(^3\)For lisibility sake, this interface is not detailed on figure 4.
is open and to be determined by the developers of probe components. They can span from primitive measurements to more advanced performance indicators processed by various statistical models (interpolations, correlations, etc.). It should be noted that, as Fractal components support nesting, the proposed integration patterns can be applied at any level of hierarchy.

In our example (see figure 4), properties are modeled as follows. The maxUsers and groupedUsersRatio properties of InstantGroup are respectively a configuration parameter that is defined when configuring the server to set the maximum threshold of concurrent users, and a functioning parameter that describes the rate of users that have been grouped. They are modeled with read-and-write Fractal attributes. The property nbUsers of User Manager is a functioning parameter that describes how many users have been registered. It is modeled with a read-only Fractal attribute. The property nbGroupedUsers of GroupManager is a functioning parameter that describes how many users have been grouped, and is also modeled with a read-only attribute. All these attributes are accessed through the attributes controller interface of their corresponding component. As for the network bandwidth property, the associated resource probe is modeled with a Fractal component (BdwMonitor), and the probed data, such as the level of network bandwidth used (getBdWidthLevel()), are modeled through the functional interface BdWidthInfo. Moreover, as BdwMonitor relies on the level of bandwidth used for the messaging and video service to compute the overall bandwidth, it uses MsgMonitor and VideoMonitor which are then shared and hosted in BdwMonitor. For lesibility sake, the sharing of these components is further detailed in section 4.4.

4 Exploitation in Contract Negotiation

In this section, we briefly introduce a contracting system for the Fractal platform, and illustrate how the proposed patterns and the reasoning support for compositional properties are used to develop a propagative contract negotiation policy.

4.1 The Contracting System ConFract

ConFract [7] is a contracting system for Fractal components, which provides several kinds of contracts, both on interfaces and on components themselves. In ConFract, contracts are first class entities during both configuration and run times. They follow the lifecycle of components, and are automatically updated when dynamic reconfigurations occur. Contracts are composed of provisions which are built at assembly time, from specifications provided by designers. Specifications are currently written in an executable assertion language which is inspired by OCL [17] and adapted to the Fractal model. They support classic categories of specifications (pre, post, inv, rely), but their scope can be both on interfaces and components. The ConFract system distinguishes various types of contracts: classic interface contracts, similar to object contracts [15], are built on bindings between a required and a provided interface, and different composition contracts are built on the external and the internal sides of components, to respectively express the usage and the internal assembly of components.

The various contracts are hierarchically managed at the level of each composite component, by dedicated entities which are implemented in Fractal by contract controllers. These controllers also operate contract checkings when appropriate events occur, and, when contracts are violated, they throw an exception describing the context of the violation. By using a metamodel, ConFract also assigns, for each category of specifications, appropriate responsibilities to involved components, which can notably be guarantor, which acts to ensure the provision and must be notified in case of violation of the provision, and beneficiaries which can rely on the provision. More details about ConFract can be found in [7].

4.2 Negotiation Mechanisms

As contracts can specify non-functional properties that may fluctuate, contracts may be frequently violated, and some mechanisms to handle these violations are required. To this end, a general negotiation model has been proposed [6]. It aims at restoring the validity of violated contracts by activating an atomic negotiation for each violated provision of a contract. This negotiation can occur at assembly time if a provision can be statically checked, or dynamically, at run time, if it requires an execution context.

The negotiation model relies on the clearly identified responsibilities (beneficiaries, guarantor) assigned by the ConFract system to participating components. It makes negotiating parties interact following a negotiation protocol which is partly inspired from the extended Contract-Net Protocol (CNP) [20], commonly applied in multi-agent systems for decentralized tasks allocation. In ConFract, the negotiating parties are basically the contract controller and the responsible participating components, which are determined for each violated contract provision. The contract controller acts in the role of the negotiation initiator and conducts the negotiation process, as it manages contract lifecycle and operates contract checking. The protocol basically organizes the interactions between the contract controller and the responsible components following three steps (request of proposals, proposal of modifications and re-checking of the provision against the proposed modification). Finally, the responsibilities of participating com-
ponents are also exploited to develop different negotiation policies which drive the whole negotiation process. Currently, a concession-based policy, in which the negotiation initiator requests concessions from the beneficiary components only, is provided [6].

4.3 A Propagative Negotiation Policy

To enrich the model with more powerful kinds of negotiation, our objective is to design another negotiation policy, which now consists in exploiting the responsibility of the guarantor component, and propagating negotiation processes down the hierarchy of the guarantor component. Some contributing components at lower levels are then consulted so that they can make some efforts to revalidate a violated contract at a higher level. To successfully achieve this, the propagation of the negotiation strongly relies on the compositional information. The contributing properties that decompose a given compositional property are used to drive the propagation from a level of component hierarchy to the sub-level. They are identified using the compositional function. Moreover, for each contributing property, its realizing component is explicitly identified according to the integration patterns. Each contributing component which is asked for efforts, may then propose some changes regarding the property it realizes.

When the checking of a contract provision fails, the overall negotiation process using the propagative negotiation policy involves the contract controller, which manages the violated contract and the guarantor component. It then executes according to the following steps:

1. The contract controller requests proposals from the guarantor component. In response to these requests, the guarantor component can then either make proposals to revalidate the provision at its level or, decide to consult some components in its content (if it is composite);

2. In this latter case, the contract controller of the guarantor takes in charge the negotiation and thus have to identify the set of components that contribute in the property that is specified in the contract provision, and that are to be consulted;

3. These components either implement the property, or belong to the set of components that contribute in decomposing that property. To precisely identify them, this contract controller then either uses the integration patterns to identify the component that implement the property, or interacts with the compositional properties manager\(^4\) and the associated compositional property meta-object to retrieve the compositional information that describe the decomposition of that property, following the protocol described in figure 3 page 4;

4. Once identified, these contributing components are consulted by the contract controller in order to make proposals that may revalidate the violated contract provision. At their turn, they can either propose changes that may revalidate the contract provision, or take in charge the negotiation and propagate it in their content, following the process as in step 2.

To synthesize this, figure 6 shows an overview of the various entities involved and their interactions during the compositional reasoning and the propagative negotiation process. The negotiation focuses on a contract provision built from the specification \( C \text{.ci.P} < 100 \), which expresses a maximum threshold for the property \( P \) realized through the interface \( ci \) of the component \( C \). The contract is managed by the contract controller (CTC) of the component \( D \), which then activates an atomic negotiation in case of violation. The meta-object of the compositional property \( P \) is built from the compositional function \( C \text{.ci.P} := f(A \text{.ai.P1,B.bi.P2}) \). It is exploited by the contract controller of the component \( C \) to retrieve the compositional information, and identify the set of contributing components, \( A \) and \( B \), in order to consult them.

Figure 6: Overview of the propagative negotiation process.

4.4 Illustration

Contracts.

Figure 7 shows an internal composition contract which is built in the content of the component \(<fic>\). This contract is managed by the contract controller (CTC in figure 7) of \(<fic>\), and contains three contract provisions which

---

\(^4\)The compositional properties manager is implemented in Fractal by a dedicated compositional properties controller.
express some internal behavior rules on the implementation of <fic>

The first provision (see figure 7) defines an invariant on the configuration of <ig>, such that the maximum threshold of concurrent users that the server can support (maxUsers) is higher than 250 users. The second one constraints <ig> by defining a minimum threshold of 80% for the groupedUsersRatio (on 10 registered users, at least 8 of them must belong to a group), which must hold all along the execution of every method of <um> (rely construction and operator *). The third provision constraints <bm> by defining a bandwidth consumption threshold of 30 ko/sec, which is required to prevent a high bandwidth use. This constraint must hold all along the execution of every method in the content of <fic>. As for their checking, the first contract provision is checked at configuration time, as it specifies an invariant of the configuration. The other provisions, which specify functioning and resource capacity properties, are checked at runtime. Regarding responsibilities, for each of these three contract provisions, <fic> is at the same time, the guarantor and the beneficiary component, as it has to take in charge its internal assembly and also benefits from it.

Scenario for a configuration parameter.

The first provision may be violated, if for example, the component <ig> supports by default a maximum threshold of 100 users. Let us suppose that the compositional relation (R1) (see figure 8) is provided to describe the fact that the property maxUsers of <ig> decomposes itself identically into the same property maxUsers of <um>. The negotiation process then involves the contract controller (CTC) of <fic> and <fic> itself, as the unique guarantor. It executes as follows. First, the contract controller consults the component <fic> and requests it from some proposals (step 1). As <um> is responsible of its internal assembly, it then takes in charge the negotiation process and consults its subcomponent <ig>, which carries the property maxUsers (step 2). To propagate the negotiation in its content, the contract controller of <ig> interacts with the compositional properties controller (named CPC in figure 8) and the compositional property meta-object associated to the maxUsers property, to identify the components to be consulted. The compositional information that describe the maxUsers property (step 3) are then retrieved, and the component <um> is identified as the unique contributing component. The contract controller of <ig> requests proposals from <um> which, according to its negotiation reasoning, makes proposals that may, for example, consist in reconfiguring its parameter with a higher value of maxUsers (maxUsers=300 for example) (step 4). For each proposal, the contract controller of <ig> uses the compositional function to evaluate whether the proposed changes are sufficient to revalidate the contract provision.

Scenario for a functioning parameter.

The second provision may be violated if the grouped users ratio is lower than 80%. To negotiate this, let us suppose that the compositional relation (R2) (see figure 9), describes the decomposition of the property groupedUsersRatio of <ig> into the property nbGroupedUsers of <gm> and nbUsers of <um>. It expresses the fact that the grouped users ratio is equal to the ratio between the number of users...
in groups and the overall number of users. The negotiation process involves the same negotiating parties as in the previous example, and it is propagated at the level of <ig> using the same propagation scheme (step 1 and 2). However, the components that contribute here in realizing the property groupedUsersRatio are <gm> and <um>, which respectively exhibit the property nbGroupedUsers and nbUsers (step 3). They are thus consulted (step 4), but, as these properties describe functioning properties of <gm> and <um>, they cannot be directly changed. <gm> and <um> are likely to be unable to propose some efforts. The negotiation then terminates with a failure, which leads to an exception. This exception may be caught outside any negotiation process in order to perform more adhoc and global adaptations or reconfigurations of components (changing the <gm> component, etc.).

Scenario for a resource capacity property.

The third provision is challenged if the global bandwidth consumption exceeds 30 ko/sec. As the BdwMonitor component relies on the bandwidth levels of the messaging and video services, which are measured by the probe components <mm> and <vdm>, these two components are shared and their associated slave instances, <mm'> and <vdm'>, are hosted in the content of BdwMonitor. Besides, the compositional relation named (R3) in figure 10 is provided to describe that the property bdWidthLevel of <bm> decomposes into the sum of the property bdWidthLevel of <mm'> and <vdm'>. The negotiation process is then propagated at the level of <bm>. Its contract controller takes in charge the negotiation (step 2) and identifies the components <mm'> and <vdm'> as the contributing components, using the compositional contract controller and the property meta-object (step 3). However, as opposed to previous examples, <mm'> and <vdm'> are not consulted as they merely represent the slave instances of resource probe components, which cannot propose efforts at the application level. The sharing relation is then exploited to retrieve the reference to the master probe components <mm> and <vdm> (step 4), from which the enclosing business components <ig> and <vs> are retrieved (step 5). <vg> and <ig> are then consulted by the contract controller of <bm> (step 6), as they are the components at the application-level that may propose efforts to lower the bandwidth consumption (selecting lower bitrates, changing used codecs, compressing file transfers, etc.).

Figure 9: Propagative scheme for the groupedUsersRatio property.

Figure 10: Propagative scheme for the bdWidthLevel property.

5 Conclusion

In this paper, we proposed integration patterns that model low-level measurable non-functional properties to individual software components. These patterns are based on a classification that focuses on non-functional properties that derive directly from components, and that considers their relation to components and their lifecycle. Properties are then distinguished among attributes of nature, configurable parameters, functioning parameters, resources and resource capacities. Integration patterns are defined at an abstract level of architecture specification using the UML, and we have also described how they map to a hierarchical component platform, namely Fractal. As properties are clearly modeled to components and derived directly only from them, some simple compositional relations, which describe the realization of properties given component compositions, can be expressed. Elements at the component meta-level have also been provided to support reasoning on such compositional properties. We finally showed how integration patterns and the compositional support are then further exploited to negotiate non-functional contracts on Fractal hierarchical components. They are used in a general propagative scheme, which, by following the compositional relationship between properties, propagate the negotiation to
contributing components, so that they may propose efforts to revalidate violated contracts.

Currently, the classification and integration patterns are limited to the range of low-level measurable properties and have to be developed by considering other non-functional properties such as high-level (maintainability, reusability, availability...) or temporal properties (execution time, latency, periodicity...). To do so, high-level properties should be classified and decomposed into successive lower-level properties that could be measured. However, identifying the relevant non-functional properties is essentially domain-specific, and decomposing them requires a very extensive work, as shown in the QoS properties catalog and in the QoS mapping system developed respectively in the UniFrame [4] and COMQUAD projects [22]. As for temporal properties, they should require further information about the execution behavior, and could be measured using scenario simulation approaches [3] or static timing analysis frameworks [10].

The working example of this paper is extracted from a larger web system, which manages automatic grouping of users and application sharing. This system is already operational and has been developed using the Fractal component platform and web service technologies. The integration of the proposed patterns have been validated on this system. The proofs of concept of the compositional meta-level and the proposed patterns have been validated on this system. Further validation elements on the proposed classification and integration patterns, as well as the compositional reasoning support. Other future work will focus on exploiting these patterns in service-oriented architectures built from hierarchical components.

References