Fuzzy Description Logics for Component Selection in Software Design

Tommaso Di Noia[1], Marina Mongiello[1(✉)], and Umberto Straccia[2]

[1] Politecnico di Bari, Via E. Orabona, 4, 70125 Bari, Italy
{tommaso.di.noia,marina.mongiello}@poliba.it
[2] ISTI-CNR, Via G. Moruzzi, 1, 56124 Pisa, Italy
umberto.straccia@isti.cnr.it

Abstract. In the Future Internet era, the way software will be produced and used will more and more depend on the new challenges deriving from the virtually infinite number of software services that can be composed to build new applications. The integration and composition of existing software, components and services is now gaining a crucial role in the software modeling and production and encompasses several aspects ranging from theoretical issues like modeling and analysis, to practical and implementation ones like run-time management and integration. In the wide set of issues concerning software composition, in this position paper we propose a formalization via a Fuzzy Description Logic for modeling architectural aspects of a software system.

The formalism models architectural patterns and non-functional requirements about quality attributes where both the relationships among patterns and the set non-functional requirements are modelled together with their mutual interactions. The declarative approach proposed here would make possible to formally represent and maintain the above mentioned knowledge by keeping the flexibility and fuzziness of modeling thanks to the use of fuzzy concepts as high, low, fair, etc. We also identify the need for a reasoning task able to exploit the fuzzy nature of the adopted logic to retrieve a ranked list of set of patterns covering given user requirements represented in terms of NFRs and families of patterns.

1 Introduction

The way software will be produced in the next Future Internet era —according to given goals and by integration and compositions of existing services and components— calls for new formally grounded and formalized aspects and methods to support the conceptual modeling of system specifications.

Important issues concerning software architectures, design decisions, quality and goals evaluations are closely linked, anyway any formal definition is available to get relevant information and support in software modeling based on this set of features. In defining and modeling software systems a set of related but complex issues must be considered when composing pieces of reusable artifacts

D. Bianculli et al. (Eds.): SEFM 2015 Workshops, LNCS 9509, pp. 228–239, 2015.
DOI: 10.1007/978-3-662-49224-6_19

through design or architectural patterns driven by non functional requirements satisfaction.

In this paper we propose a formalization via a Fuzzy Description Logic for modeling architectural aspects of a software system. The formalism models architectural patterns and non-functional requirements about quality attributes where both the relationships among patterns and the set non-functional requirements are modeled together with their interactions. The framework we propose enables to represent and reason on mutual relationships among non functional requirements by means of fuzzy Description Logics (DL). The fuzzy version of DLs is needed in order to represent both ontological relations and factual ones. In the former case we may model that "*portability* and *adaptability* are directly proportionate" while "*stability* and *adaptability* are inversely proportionate". For the latter we may represent that "the Adapter pattern has a *high portability*". We see that the combination of ontological and factual knowledge allows a reasoning procedure to infer new information about a pattern. With reference to the previous example, we may infer that "the Adapter pattern has a *high adaptability* and a *low stability*". In the framework we propose here we are allowed also to formally define that "a *high adaptability* implies a *medium maintainability*[1]". Once the knowledge about non functional requirements has been modeled via a formal language and encoded in a knowledge base, we need a tool to query and retrieve data related to a specific task. In our case, the task we propose to solve is the following: given a set of non functional requirements $\mathcal{R} = \{r_1, \ldots, r_n\}$, retrieve the minimal subset of patterns that better satisfies them. This means that we prefer patterns with a high value of a specific functional requirement r_i to those with a medium or low one. If there is no pattern with high value for r_i, than we prefer patterns with a medium value to those with a low one. In such patterns, fuzziness is evident: in fact, terms such as *high*, *medium* and *low* can be defined in terms of fuzzy sets [29].

The remaining of this paper is organized as follows. In the next section we motivate our proposal. Section 3 describes the approach we use to model the ontology and defines a theoretical algorithm. Section 4 presents a case study to explain the proposed idea. Conclusions and future works close the paper.

2 Motivation

Since Anton Jansen and Jan Bosch [15] gave a modern definition of Software Architecture, several important issues concerning software architectures, design decisions, quality and goals evaluations have been dominating the scientific literature in this field as comprehensively and systematically provided by Tofan et al. [27]. Designing the software architecture of non-trivial systems belonging to several application domains, namely industrial automation, defense and telecommunication financial services, and so on, is not an easy task, and requires highly

[1] In this case we do not have a high maintainability as the system adapts its configuration to context (or requirements) variations. At the same time the maintainability is not low as it has its own internal logic.

skilled and experienced people. Beyond these, new challenges in the design and in architectural models are derived from self-managing and self-adaptive capabilities that are typical of many modern and emerging software systems, including the industrial internet of things, cyber-physical systems, cloud computing, and mobile computing. The satisfaction of quality requirements and the appropriate options for future changes are among the major goals of software architectures, even more important than functional requirements. Quality goals often compete or even conflict with each other and with functional requirements. In defining and modeling software architecture through patterns, a challenging issue is also concerned with the number of different available decisions depending on the fact that patterns can cooperate, are composable, are complementary or exclusive with respect to a given problem [9,19]. To solve the challenging problem of choosing a set of patterns, some structures have been proposed supported by pattern languages with a given syntax and style [5].

In self-adaptable models but also in classical software architectures, the link between architecture and design-time features modeling and the relationship between non-functional requirements, patterns and design decisions should be made more flexible. The idea is to provide an approach to more faithfully reproduce the existing relationships in order to formalize the extent to which they are guaranteed in the design of software architecture.

3 Problem Statement and Approach

In software design domain, a typical problem to solve is the following:

> "*Given a set of requirements define the software design that (better) models the given requirements*"

In order to solve such a problem, adopted empirical approaches generally depend on the designer's know-how and experience.

According to modern software production and modeling, mainly based on component integration and/or composition and according to the modern definition of software architecture [15], the above problem is that of finding the architectural model as a solution to a decision making problem. The architectural model can hence be defined using design or architectural patterns selected according to Non-functional Requirements (NRFs) satisfaction. The selection of the right NFRs may result crucial in the initial design of a software system. It may happen that the designer is looking for the best design solution given a set of non functional requirements and some problem areas and/or pattern families related to the system. Design patterns give proven solutions to recurrent problems based on typical situations. Therefore, they are a first attempt to formalize the knowledge about NFRs and to give a somewhat structured approach to their compliance in the software design [5,6,11].

With respect to the general problem stated at the beginning of this section, we restrict our interest to the connection among patterns, problem areas/families and NFRs. More in detail we are aiming to finding a solution to the following design problem:

> "*Given a software design to model, a set of NFRs and the problem areas the software refers to, which are the components/patterns that best fit them?*"

The task is non-trivial as: NFRs may be disjoint with each other and cannot be satisfied at the same time; some families may not contain patterns satisfying some of the NFRs. Moreover, the designer may not be aware of all the patterns available given a NFR or given a pattern family.

To the best of our knowledge, the software design theory misses a formal superstructure to integrate and relate the elements of the given sets and implicit or explicit relationships between elements. The approach we propose here, exploits a Fuzzy Description Logic and related reasoning tasks in order both (i) to provide a formal representation of the relations intercurring among design areas, NFRs and design patterns and (ii) to reason with such a representation to help the designer during the selection of the right set of patterns that best match the initial requirements. Full and exaustive background in Description Logics and Fuzzy Description Logic are available in literature ([2] for DLs and [22–24] for fuzzy DLs).

We will illustrate how to encode the information by leveraging on:

- Fuzzy DL statements to have a high level model of the domain we are dealing with and to represent relations among non-functional properties;
- Fuzzy DL reasoning to infer new knowledge about NFRs mutual relations and to retrieve sets of patterns satisfying specific requirements;

We next describe the role played by each of the above indicated technologies in the decision process.

Fuzzy DL statements. In order to encode all the information related to NFRs, patterns and corresponding families, we need a formalization of the domain knowledge. The ontology we use to cope with this task can be seen as composed by two main modules: the one describing, at a high level, the connections between patterns and families and between patterns and NFRs; the other one modeling the relations intercurring among NFRs. The formal definition of the ontology is encoded in Fuzzy DL as:

$$\exists \texttt{isInFamily} \sqsubseteq \texttt{SoftwareDesignPattern}$$
$$\exists \texttt{nFR} \sqsubseteq \texttt{SoftwareDesignPattern}$$
$$\top \sqsubseteq \forall \texttt{isInFamily}.\texttt{Families}$$
$$\top \sqsubseteq \forall \texttt{nFR}.\texttt{NonFunctionalRequirement}$$

The first two statements represent *domain* restrictions while the last two represent range ones. In other words we say that the role `isInFamily` connects instances of the concept `SoftwareDesignPattern` to instances of the concept `Families` while the role `isInFamily` relates instances of `SoftwareDesignPattern` to instances of `NonFunctionalRequirement`.
Please note that the structure of the high level ontology we model makes it possible to easily extend it to deal also with other elements, such as Functional Requirements.

Given the ontology, we can state explicit facts about the description of a pattern in terms of pattern family it belongs to and NFRs it guarantees. These statements form the ABox of our knowledge base. Specifically, let us consider the following Fuzzy DL assertions:

$$\begin{aligned}&\mathsf{proxyPattern{:}SoftwareDesignPattern}\\&\mathsf{resourceManagement{:}Families}\\&\mathsf{reliability{:}NonFunctionalRequirement}\\&\mathsf{loadBalancing{:}NonFunctionalRequirement}\\&\mathsf{reusability{:}NonFunctionalRequirement}\\&\mathsf{(proxyPattern,resourceManagement){:}isInFamily}\end{aligned}$$

That is, we introduced the pattern `proxyPattern`, the family `resource Management` and the non-functional requirements `reliability`, `loadBalancing`, `reusability` as instances/individuals of the classes `SoftwareDesignPattern`, `Families` and `NonFunctionalRequirement` respectively.

Based on these individuals and properties we may wish to state that the *Proxy Pattern* assures a high *Load Balancing* and a high *Reliability*. In order to formally represent such constraints we need to introduce new datatype properties, together with the corresponding fuzzy sets, and axioms related to the non functional requirements we just introduced. In the following we will always refer to them for every datatype property and we will use $\mathbf{R}$ (for rating) to represent the interval [`very bad`, `bad`, `medium`, `good`, `very good`]. The set of axioms we are going to define are needed in order to exploit the full potential of the fuzzy DL reasoning. It is noteworthy that in a production scenario, they can be automatically added to the knowledge base in a straight way. These are:

$$\begin{aligned}&\exists\texttt{nFR.}\{\texttt{reliability}\} \equiv \exists\texttt{reliabilityRate.} \in_{\mathbf{R}}\\&\exists\texttt{nFR.}\{\texttt{loadBalancing}\} \equiv \exists\texttt{loadBalancingRate.} \in_{\mathbf{R}}\\&\exists\texttt{nFR.}\{\texttt{reusability}\} \equiv \exists\texttt{reusabilityRate.} \in_{\mathbf{R}}\end{aligned}$$

In the previous statements we say that whenever we have a pattern with an associated non functional requirement we will always have a corresponding degree and vice versa. Based on the datatype properties just introduced we can state, for instance, that

$$\begin{aligned}&\mathsf{proxyPattern{:}}\exists\mathsf{loadBalancingRate.} =_{\mathsf{good}}\\&\mathsf{proxyPattern{:}}\exists\mathsf{reliabilityRate.} =_{\mathsf{good}}\end{aligned}$$

Besides the modeling of the ABox relations, we use Fuzzy DL also to explicitly model relations intercurring between NFRs. Consider the non-functional requirements *load balancing*, *reliability*, *reusability* previously defined as instances (individuals) of the class `NonFunctionalRequirement`. An example of mutual relation between NFRs is the one between *load balancing* and *reliability*. Indeed, they are directly proportionate, i.e., if the *loadBalancing* increases (decreases) the same happens for the *reliability*. With reference to our ontology, such a relation can be written in Fuzzy DL as

$$\exists \texttt{loadBalancingRate.High} \sqsubseteq \exists \texttt{reliabilityRate.High}$$
$$\exists \texttt{loadBalancingRate.Fair} \sqsubseteq \exists \texttt{reliabilityRate.Fair}$$
$$\exists \texttt{loadBalancingRate.Low} \sqsubseteq \exists \texttt{reliabilityRate.Low}$$

We also know that a system cannot be *reliable* and *reusable* at the same time. That is, the two non functional requirements are inversely proportionate. Hence, if a pattern guarantees *reliability* it cannot guarantees also *reusability*. We may encode such disjoint relations with the following statement:

$$\exists \texttt{reusabilityRate.High} \sqsubseteq \exists \texttt{reliabilityRate.Low}$$

From the two relations explicitly stated before, we may imply that as the *Proxy Pattern* guarantees a high degree of *load balancing*, it cannot guarantee a high degree of *reusability*.

By using automated reasoning over Fuzzy DL knowledge bases we automatically infer all these kind of implicit relations thus providing better results while looking for a design solution. It is noteworthy that the Fuzzy DL we are targeting is allowed to represent also statements like: "a system with a high adaptability has a fair maintainability". Indeed, as the system adapts its own configuration to context o requirements variations it may not be highly maintainable, i.e.,

$$\exists \texttt{adaptabilityRate.High} \sqsubseteq \exists \texttt{maintainabilityRate.Fair}$$

4 Use Case Scenario

We next illustrate how to apply the proposed framework by means of a use case scenario. We model the use case of a *Cloud-Social-Adaptable System*. In the cloud environment, let us think of an application in social domain in which the various applications (apps) share data distributed over different clusters or data centres.

The system is allowed to dynamically and extensively load external applications depending on variations in the context or depending on changes in the behavior of the user. For example, if the user is travelling for a week-end or on holiday, an app arranges all stored material related to the destination of the trip and creates albums, photo collections with captions, stories etc. Other context-dependent conditions set in the application, enable dynamic loading of different apps. Dynamically loaded applications might compromise properties of the entire system, then it is of crucial importance to provide mechanisms working at run-time and able to check and guarantee the preservation of properties of interest. All nodes in the application are started exploiting the Cloud virtualization, i.e. the physical hardware is shared between all services but the software environment is independent, ensuring low coupling between the virtual nodes. The architecture is flexible since every consumer can access a public service with the available resources, with a saving in terms of costs. Moreover, being the various virtual machines unconnected, the fault of one of them does not compromise all others, thus ensuring a good fault-tolerance in the overall architecture. Virtual machines are made up and launched directly from the middleware just as a

consumer requests. The failure of a virtual machine does not affect the others. The availability of content is ensured by a content delivery network. The request for more machines avoids the single point of failure.

The implementation of the scenario previously described requires patterns satisfying characteristics derived both from the running environment and from the main features of the context. We see that such patterns should belong to families that manage *Cloud* features, but also must solve problems about process communication and middleware. They have to ensure adaptability being the system social-adaptable, hence it would be desirable they belong to the *Adaptation and Extension* family. Actually, the patterns we are looking for could also be in the *Application Control* family. Indeed, given the specific environment of the system, it should be necessary to separate the interface from the applications core functionalities. As for the Non Functional Requirements, the architectural model must ensure *adaptability*, being the system a social adaptable and *elasticity* since it will work in a cloud environment and will manage a large amount of data. Also *fault tolerance* is a requirement to be satisfied in order to ensure a dependable system. A low level of *coupling* is also required being the system implemented in a cloud environment. All these requirements can be summarized as:

- belonging families: *Cloud Patterns*, *Application Control*, *Adaptation and Extension*, *Distribution Infrastructure*;
- non functional requirements: high *adaptability*, high *elasticity*, high *fault tolerance*, low *coupling*.

Now suppose we have defined in a knowledge base $\mathcal{K}$ (ABox and TBox) information about a set of patterns, families and NFRs. That is, the ABox contains

$$\mathsf{adaptationAndExtension{:}Families}\ ,\ \mathsf{cloud{:}Families}\ ,\ \mathsf{distributionInfrastructure{:}Families}$$
$$\mathsf{applicationControl{:}Families}$$

$$\mathsf{adaptability{:}NonFunctionalRequirement}\ ,\ \mathsf{stability{:}NonFunctionalRequirement}$$
$$\mathsf{maintainability{:}NonFunctionalRequirement}\ ,\ \mathsf{simplicity{:}NonFunctionalRequirement}$$
$$\mathsf{dependability{:}NonFunctionalRequirement}\ ,\ \mathsf{redundancy{:}NonFunctionalRequirement}$$
$$\mathsf{performance{:}NonFunctionalRequirement}\ ,\ \mathsf{reliability{:}NonFunctionalRequirement}$$
$$\mathsf{elasticity{:}NonFunctionalRequirement}\ ,\ \mathsf{faultTolerance{:}NonFunctionalRequirement}$$
$$\mathsf{loadBalancing{:}NonFunctionalRequirement}\ ,\ \mathsf{security{:}NonFunctionalRequirement}$$
$$\mathsf{flexibility{:}NonFunctionalRequirement}\ ,\ \mathsf{scalability{:}NonFunctionalRequirement}$$
$$\mathsf{coupling{:}NonFunctionalRequirement}\ ,\ \mathsf{robustness{:}NonFunctionalRequirement}$$
$$\mathsf{resilience{:}NonFunctionalRequirement}$$

$$\mathsf{reflection{:}SoftwareDesignPattern}\ ,\ \mathsf{strictConsistency{:}SoftwareDesignPattern}$$
$$\mathsf{hypervisor{:}SoftwareDesignPattern}\ ,\ \mathsf{observer{:}SoftwareDesignPattern}$$
$$\mathsf{broker{:}SoftwareDesignPattern}$$

$$(\mathsf{reflection,adaptationAndExtension}){:}\mathsf{isInFamily}\ ,\ (\mathsf{strictConsistency,cloud}){:}\mathsf{isInFamily}$$
$$(\mathsf{hypervisor,cloud}){:}\mathsf{isInFamily}\ ,\ (\mathsf{observer,applicationControl}){:}\mathsf{isInFamily}$$
$$(\mathsf{broker,distributionInfrastructure}){:}\mathsf{isInFamily}$$

$$\mathsf{reflection}{:}\exists \mathsf{adaptabilityRate}.\ =_{\mathsf{very\ good}}\ ,\ \mathsf{reflection}{:}\exists \mathsf{stabilityRate}.\ =_{\mathsf{bad}}$$
$$\mathsf{reflection}{:}\exists \mathsf{maintainabilityRate}.\ =_{\mathsf{medium}}\ ,\ \mathsf{reflection}{:}\exists \mathsf{simplcityRate}.\ =_{\mathsf{bad}}$$

strictConsistency:∃dependabilityRate. $=_{\text{very good}}$, strictConsistency:∃redundancyRate. $=_{\text{medium}}$
strictConsistency:∃performanceRate. $=_{\text{medium}}$, strictConsistency:∃reliabilityRate. $=_{\text{good}}$

hypervisor:∃elasticityRate. $=_{\text{very good}}$, hypervisor:∃redundancyRate. $=_{\text{medium}}$
hypervisor:∃faultToleranceRate. $=_{\text{good}}$, hypervisor:∃loadBalancingRate. $=_{\text{good}}$
hypervisor:∃securityRate. $=_{\text{good}}$

observer:∃flexibilityRate. $=_{\text{good}}$, observer:∃scalabilityRate. $=_{\text{medium}}$
observer:∃adaptabilityRate. $=_{\text{medium}}$

broker:∃loadBalancingRate. $=_{\text{very good}}$, broker:∃robustnessRate. $=_{\text{medium}}$
broker:∃faultToleranceRate. $=_{\text{bad}}$, broker:∃performanceRate. $=_{\text{medium}}$
broker:∃reliabilityRate. $=_{\text{good}}$, broker:∃resilienceRate. $=_{\text{good}}$

Eventually, the TBox contains

$$\exists \texttt{flexibilityRate.High} \sqsubseteq \exists \texttt{couplingRate.Low}$$

$$\exists \texttt{elasticityRate.High} \sqsubseteq \exists \texttt{adaptabilityRate.High}$$
$$\exists \texttt{elasticityRate.Fair} \sqsubseteq \exists \texttt{adaptabilityRate.Fair}$$
$$\exists \texttt{elasticityRate.Low} \sqsubseteq \exists \texttt{adaptabilityRate.Low}$$

$$\exists \texttt{robustnessRate.High} \sqsubseteq \exists \texttt{faultToleranceRate.High}$$
$$\exists \texttt{robustnessRate.Medium} \sqsubseteq \exists \texttt{faultToleranceRate.Medium}$$
$$\exists \texttt{robustnessRate.Low} \sqsubseteq \exists \texttt{faultToleranceRate.Low}$$

$$\exists \texttt{faultToleranceRate.High} \sqsubseteq \exists \texttt{reliabilityRate.High}$$
$$\exists \texttt{faultToleranceRate.Medium} \sqsubseteq \exists \texttt{reliabilityRate.Medium}$$
$$\exists \texttt{faultToleranceRate.Low} \sqsubseteq \exists \texttt{reliabilityRate.Low}$$

$$\exists \texttt{reliabilityRate.High} \sqsubseteq \exists \texttt{dependabilityRate.High}$$
$$\exists \texttt{reliabilityRate.Medium} \sqsubseteq \exists \texttt{dependabilityRate.Medium}$$
$$\exists \texttt{reliabilityRate.Low} \sqsubseteq \exists \texttt{dependabilityRate.Low}$$

$$\exists \texttt{loadBalancingRate.High} \sqsubseteq \exists \texttt{elasticityRate.High}$$
$$\exists \texttt{loadBalancingRate.Medium} \sqsubseteq \exists \texttt{elasticityRate.Medium}$$
$$\exists \texttt{loadBalancingRate.Low} \sqsubseteq \exists \texttt{elasticityRate.Low}$$

In order to get the best set of patterns satisfying the requirements needed to solve the task, we need a reasoning method for retrieving more suitable patterns according to given requirements.

The set of retrived pattern should be incrementally built considering subsets of the original requirements:

Specifically, let us start with the subset

– belonging families: *Adaptation and Extension*;
– non functional requirements: *adaptability.*

The formula modeling the previous requirements is thus:

$$C' = \exists \mathsf{isInFamily}.\{\mathsf{adaptationAndExtension}\} \sqcap \exists \texttt{adaptabilityRate.High}.$$

Therefore, the retrieved ranked list of top-3 patterns satisfying C' would be: $\langle$reflection, strictConsistency, hypervisor$\rangle$ When we defined C' we assumed the patterns we were looking for belonged only to the *Adaptation and Extension* family.

Actually, we may look also for patterns in other families which satisfy the *adaptability* NFR. We then extend the previous subset with

- belonging families: *Cloud Patterns*, *Application Control*, *Adaptation and Extension*, *Distribution Infrastructure*;
- non functional requirements: *adaptability*.

In order to model such a more relaxed requirement we modify C' in C_1 as

$$C_1 = (\exists \mathsf{isInFamily}.\{\mathsf{adaptationAndExtension}\} \sqcup \exists \mathsf{isInFamily}.\{\mathsf{cloud}\} \sqcup \exists \mathsf{isInFamily}.\{\mathsf{applicationControl}\} \sqcup \exists \mathsf{isInFamily}.\{\mathsf{distributionInfrastructure}\}) \sqcap \exists \mathtt{adaptabilityRate}.\mathtt{High}.$$

With respect to the newly stated requirements, the retrieved ranked list of patterns is: ⟨hypervisor, reflection, observer⟩ with `hypervisor` and `reflection` in the same ranking position, and `observer` that can be equivalently substituted by broker.
We may also define C_2, C_3 and C_4

$$C_2 = (\exists \mathsf{isInFamily}.\{\mathsf{adaptationAndExtension}\} \sqcup \exists \mathsf{isInFamily}.\{\mathsf{cloud}\} \sqcup \exists \mathsf{isInFamily}.\{\mathsf{applicationControl}\} \sqcup \exists \mathsf{isInFamily}.\{\mathsf{distributionInfrastructure}\}) \sqcap \exists \mathtt{elasticityRate}.\mathtt{High}.$$

$$C_3 = (\exists \mathsf{isInFamily}.\{\mathsf{adaptationAndExtension}\} \sqcup \exists \mathsf{isInFamily}.\{\mathsf{cloud}\} \sqcup \exists \mathsf{isInFamily}.\{\mathsf{applicationControl}\} \sqcup \exists \mathsf{isInFamily}.\{\mathsf{distributionInfrastructure}\}) \sqcap \exists \mathtt{faultToleranceRate}.\mathtt{High}.$$

$$C_4 = (\exists \mathsf{isInFamily}.\{\mathsf{adaptationAndExtension}\} \sqcup \exists \mathsf{isInFamily}.\{\mathsf{cloud}\} \sqcup \exists \mathsf{isInFamily}.\{\mathsf{applicationControl}\} \sqcup \exists \mathsf{isInFamily}.\{\mathsf{distributionInfrastructure}\}) \sqcap \exists \mathtt{couplingRate}.\mathtt{Low}.$$

Hence, the best solution is to adopt both `broker` and `hypervisor`, then we have the pair `reflection`, `broker` and eventually `observer`, `broker`.

5 Related Work

In this section we briefly review the literature on knowledge representation approaches architectural concerns such as nonfunctional requirements, design pattern and service composition.

QoS aspects of Service composition are studied in [7,18]. While Architectural concerns about services and future Internet applications are considered in [3,14,16]. Temporal behavior of design patterns are modeled in [20] using formal methods. A Balanced Pattern Specification Language (BPSL) is modeled in [25] based on a subset of First Order Logic and Temporal Logic of Action to specify both structural and behavioral aspects of patterns [26]. Relationships among design pattern as architectural tactics and architecture are studied in [13]. Pattern languages connect patterns from a variety of different sources into a single pattern network [17]; they are supported by trees or directed graphs and

enable typical search algorithms of depth-first or breadth-first. Anyway the modeling of relationships and sequences of patterns and the taxonomy of the several issues concerned with their use needs a more structured approach with respect to data structures such as trees or directed graphs, due to the complex semantics they convey. A state of the art on the treatment of non-functional requirements is in [8]. Definitions of non-functional requirements are surveyed in [10,12]. In [4] a model of non-functional requirements is given.

Mylopoulos et al. propose two complementary approaches for using non-functional information: process-oriented and product oriented, [21]. State of the art of Model Driven Development approaches with respect to non-functional requirements is in [1], where also a framework for integrating NFRs in the core of Model Driven Development process is given [28].

6 Conclusion and Future Work

In this paper we proposed to use a Fuzzy Description Logic (DL) to model the knowledge related to NFRs, patterns and related families in order to ease and support software components integration and composition in architectural decision making problems.

Although the overall framework has been presented to deal with NFRs, thanks to its declarative nature, it can be easily extended to other characteristics such as Functional Requirements.

We are currently working on how to take into account also information related to temporal interaction among software components thus extending the Fuzzy DL we propose with temporal operators. We are also performing extensive experiments on a structured benchmark to test the framework functionalities and performance on simple and on composable schemes also in more advanced architectural environments.

References

1. Ameller, D., Franch, X., Cabot, J.: Dealing with non-functional requirements in model-driven development. In: 2010 18th IEEE International Requirements Engineering Conference (RE), pp. 189–198. IEEE (2010)
2. Baader, F., Calvanese, D., McGuinness, D.L., Nardi, D., Patel-Schneider, P.F. (eds.): The Description Logic Handbook: Theory, Implementation, and Applications. Cambridge University Press, Cambridge (2003)
3. Baresi, L., Caporuscio, M., Ghezzi, C., Guinea, S.: Model-driven management of services. In: 2010 IEEE 8th European Conference on Web Services (ECOWS), pp. 147–154. IEEE (2010)
4. Botella, P., Burgues, X., Franch, X., Huerta, M., Salazar, G.: Modeling non-functional requirements. In: Proceedings of Jornadas de Ingenieria de Requisitos Aplicada JIRA (2001)
5. Buschmann, F., Henney, K., Schmidt, D.C.: Pattern-Oriented Software Architecture: A Pattern Language for Distributed Computing, vol. 4 (2007)

6. Buschmann, F., Meunier, R., Rohnert, H., Sommerlad, P., Stal, M.: Pattern-Oriented Software Architecture: A System of Patterns. John Wiley & Sons Inc, New York, NY, USA (1996)
7. Caporuscio, M., Mirandola, R., Trubiani, C.: Qos-based feedback for service compositions. In: Proceedings of the 11th International ACM SIGSOFT Conference on Quality of Software Architectures, pp. 37–42. ACM (2015)
8. Chung, L., do Prado Leite, J.C.S.: On non-functional requirements in software engineering. In: Borgida, A.T., Chaudhri, V.K., Giorgini, P., Yu, E.S. (eds.) Conceptual Modeling: Foundations and Applications. LNCS, vol. 5600, pp. 363–379. Springer, Heidelberg (2009)
9. Egyed, A., Grunbacher, P.: Identifying requirements conflicts and cooperation: how quality attributes and automated traceability can help. IEEE Softw. **21**(6), 50–58 (2004)
10. Franch, X.: Systematic formulation of non-functional characteristics of software. In: 1998 Third International Conference on Requirements Engineering, pp. 174–181. IEEE (1998)
11. Gamma, E., Helm, R., Johnson, R., Vlissides, J.: Design Patterns: Elements of Reusable Object-Oriented Software. Pearson Education, Massachusetts (1994)
12. Glinz, M.: On non-functional requirements. In: 15th IEEE International Requirements Engineering Conference, RE 2007, pp. 21–26. IEEE (2007)
13. Harrison, N.B., Avgeriou, P.: How do architecture patterns and tactics interact? a model and annotation. J. Syst. Softw. **83**(10), 1735–1758 (2010)
14. Issarny, V., Georgantas, N., Hachem, S., Zarras, A., Vassiliadist, P., Autili, M., Gerosa, M.A., Hamida, A.B.: Service-oriented middleware for the future internet: state of the art and research directions. J. Internet Serv. Appl. **2**(1), 23–45 (2011)
15. Jansen, A., Bosch, J.: Software architecture as a set of architectural design decisions. In: 5th Working IEEE/IFIP Conference on Software Architecture, WICSA 2005, pp. 109–120. IEEE (2005)
16. Johnson, K., Calinescu, R.: Efficient re-resolution of smt specifications for evolving software architectures. In: Proceedings of the 10th International ACM Sigsoft Conference on Quality of Software Architectures, pp. 93–102. ACM (2014)
17. Li, F.-L., Horkoff, J., Mylopoulos, J., Liu, L., Borgida, A.: Non-functional requirements revisited. In: iStar, pp. 109–114. Citeseer (2013)
18. Maiden, N., Lockerbie, J., Zachos, K., Bertolino, A., De Angelis, G., Lonetti, F.: A requirements-led approach for specifying qos-aware service choreographies: an experience report. In: Weerd, I., Salinesi, C. (eds.) REFSQ 2014. LNCS, vol. 8396, pp. 239–253. Springer, Heidelberg (2014)
19. Mairiza, D., Zowghi, D., Nurmuliani, N.: Managing conflicts among non-functional requirements (2009)
20. Mikkonen, T.: Formalizing design patterns. In: Proceedings of the 20th International Conference on Software Engineering, pp. 115–124. IEEE Computer Society (1998)
21. Mylopoulos, J., Chung, L., Nixon, B.: Representing and using nonfunctional requirements: a process-oriented approach. IEEE Trans. Softw. Eng. **18**(6), 483–497 (1992)
22. Straccia, U.: Reasoning within fuzzy description logics. J. Artif. Intell. Res. **14**, 137–166 (2001)
23. Straccia, U.: A fuzzy description logic for the semantic web. In: Sanchez, E., (ed.) Capturing Intelligence: Fuzzy Logic and the Semantic Web, Chapter 4, pp. 73–90. Elsevier (2006)

24. Straccia, U.: Foundations of Fuzzy Logic and Semantic Web Languages. CRC Studies in Informatics Series. Chapman & Hall, Boca Raton (2013)
25. Taibi, T., Ngo, D.C.L.: Formal specification of design patterns - a balanced approach. J. Object Tech. **2**(4), 127–140 (2003)
26. Tichy, W.F.: A catalogue of general-purpose software design patterns. In: Proceedings on Technology of Object-Oriented Languages and Systems, TOOLS 23, pp. 330–339. IEEE (1997)
27. Tofan, D., Galster, M., Avgeriou, P., Schuitema, W.: Past and future of software architectural decisions-a systematic mapping study. Inf. Softw. Tech. **56**(8), 850–872 (2014)
28. Xu, L., Ziv, H., Richardson, D., Liu, Z.: Towards modeling non-functional requirements in software architecture. In: Early Aspects (2005)
29. Zadeh, L.A.: Fuzzy sets. Inf. Control **8**(3), 338–353 (1965)