University of West Bohemia in Pilsen
Department of Computer Science and Engineering
Univerzitni 8
30614 Pilsen
Czech Republic

Component Compatibility in Terms of Extra-functional Properties

The State of the Art and the Concept of Ph.D. Thesis

Kamil Ježek

Technical Report No. DCSE/TR-2010-06
May, 2010

Distribution: public

Technical Report No. DCSE/TR-2010-06
May 2010

Component Compatibility in Terms of Extra-functional Properties

Kamil Ježek

Abstract

Architectures based on composing target application functionality from pre-existing components have been successfully used in many projects, yet there are several aspects in which they fail to reach the desirable level of maturity. Since different vendors may provide components with the same functionality, extra-functional properties must be taken into account to select the component which suits the final system. This report addresses the problem of inadequate means to define extra-functional. We first provide a survey of the state of the art. We second propose a formalism based on existing approaches which addresse this inadequacy and targets at context dependency of properties. Furthermore we discuse possible directions of future work. The further aim is to evaluate extra-functional properties attached to chains of components into resulting properties representing abilities of the whole system.

This work was supported by the Grant Agency of the Czech Republic under grant number 201/08/0266 "Methods and models for consistency verification of advanced component-based applications".

Copies of this report are available on
http://www.kiv.zcu.cz/publications/
or by surface mail on request sent to the following address:

University of West Bohemia in Pilsen
Department of Computer Science and Engineering
Univerzitni 8
30614 Pilsen
Czech Republic

Contents

Chapter 1

Introduction

Current software products increase in their size. Together with effort to let computers manage still more complex kinds of information which were previously computed by humans or even were not processed for their complexity, the software expands to very complicated systems.

The current software is often created from scratch where each functionality is hand coded by developers. It is obvious that a lot of applications use the same parts of logic or work-flow. For instance, almost every web page providing user access has a log-in form, but each vendor of the pages uses they own application logic to control the login process. For that reason, partial solutions have been developed to avoid a need to repeatedly code the same or very similar functionality. The developers usually use existing libraries covering common functions, though the core of the software is still developed by hand. This consequently arises a need for a "glue" code that must be written to connect the library code with the application code. The repetition of code evidently leads to an inefficient process of the development.

The current computers are used for solving tasks from different areas of human lives. A software developed for a concrete purpose must meet concrete user needs and for that reason the developer of the system must have a good knowledge of the users domain. Since the software may cover a wide domain, the developers must learn a considerable amount of information.

Despite of software complexity, the vendors want to decrease the time-to-market as well as the price of the product. The only solution for it is to solve (i) an inefficiency of current development process and (ii) decrease amount of extra-information developers must learn. The current research aims at developing means which allow to encapsulate pieces of functionality. These functionalities are stored in so called components. The components compose the final software product and the composition is done without any additional glue code. The

underlying benefit of it is that the components may be prepared by developers who are experts in a particular domain. The developer of the final systems uses components only with knowledge about the interfaces of the components. He or she does not have to learn each detail of the targeted domain. Since no additional code is written the development time is rapidly decreased. In addition, the components may be repeatedly used in different projects which consequently decreases the price. Components seem to solve both mentioned disadvantages of current software process.

Component-based software engineering (CBSE) is promising technology with an advantage to solve problems of current development of software. The usage of components leads to a new concept of a components market (technically implemented as a component repository). The market contains pre-existing components covering a wide area of functionalities. A developer of a final system takes these components form the repository to compose the final system. When a concrete functionality is not covered by any components, the new component is developed and published on the market via the repository.

Although the CBSE idea promises considerable improvement of software development process, there are still some barriers preventing to reach its advantages.

1.1 Problem Definition

We see the main problem related to components as the trust the developer has on the component. Once the developer does not write the code and uses the component instead, he or she needs to verify whether the component is suitable for the system that is under development. In essence, the developer must know if the component communicating with other parts of the system (also composed from components) will be compatible.

Let us state that the developer of component systems needs to check component's:

1. Functional characteristics
2. Extra-functional (also referred as non-functional) characteristics

It is generally assumed that functional characteristics are easier to manage. Depending on a component model, the function of a component is expressed through an interface, a connector or any other explicit definition. The developer may easily read the definition and assume the functionality.

Extra-functional properties expresses all attributes of component which are not functional ones. It contains quality attribute such as performance, memory con-

sumption, response time, etc. or market and maintenance characteristics such as marketability, price, time to failure, time to repair, etc. They are usually difficult to manage and, so far, there is no widely used mechanism allowing to easily work with them.

1.2 Goal of the Work

This work addresses inadequate means of definition of extra-functional properties attached to software components. The main goal of this report is to provide a mechanism which allows to enrich components by extra-functional properties. The mechanism should be easy-to-use to allow its practical usage. The underlying goal of the work is to analyse the state-of-the-art and show pros and cons of other approaches. The proposed mechanism aims at avoiding lapses detected in other approaches.

The proposed mechanism for defining extra-functional properties leads to a creation of a repository of these properties where the properties are first taken from the repository and they are then attached to components. This allows to enrich components of additional meta-information allowing better compatibility checks. Another goal of the report is to develop a comparator which compares components each other and decides whether the components are compatible in terms of extra-functional properties.

This work aims at expressing extra-functional properties on the interfaces of components. It provides system developers with the possibility to bind only compatible components in two situations: (i) the system is composed into an assembly in a testing environment or (ii) the system is starting in the runtime framework.

We will explicitly target at context dependency and composition of extra-functional properties. Firstly, components run in different environments and for that reason their extra-functional properties must be related to a concrete context of usage. Secondly, the properties attached to one component may be influenced by properties of other components. For that reason we aim at developing a mechanism that composes properties in a chain of connected components and results in properties that express the whole components assembly.

To support practical usage of the mechanism and verify the correctest of theoretically defined model, it will be implemented as a toolbox including a tool working with the properties repository, a tool allowing to attach properties to components and finally a tool comparing two components.

This report is organised as follows: Chapter 2 overvies fundaments of component-based software engineering, Chapter 3 brings a survey of the state-of-the-art

related to extra-functional properties, Chapter 4 first describes registry of extra-functional properties we have developed to manage context dependency of the properties. It second evaluates pros and cons of the current mechanisms and then suggests an improvement for our future work.

Chapter 2

Background: Component Architectures

2.1 Motivation

There are several motivations which prove advantages of component-based approach. This section summarizes benefits of component-based programming. A deeper explanation of terms such as a component, a component model and a component framework is not given yet, but they will be detailed in next sections. In [8] few benefits of component-based programming are highlighted:

- **Independent extension** – Legacy software is difficult to extend. A new functionality must be inserted directly to the source code and a whole application must be rebuild, because legacy software is often developed as one monolithic system. In opposite, when components are used, a new functionality may be added by adding a new component or an existing functionality may be updated by an updated component. In addition, an extension mechanism is defined by a component model which effectively decrease a possibility of side effects (e.g miss-used communication protocol in two stand-alone systems).

- **Component markets** – Component models themselves define standards for components. Together with component frameworks, it defines mechanism of componet's deployment, running and usage. That is, no explicit definition how to install, run, uninstall etc. repeated for each components (which is typical for stand-alone programs) is needed. The definition of these standards lead to unified components that may be distributed via a common market.

- **Reduce time-to-market** – A component developed for a specific function-

ality contains only a code for the functionality itself. A component framework provides other runtime means commonly needed by components. For that reason, each component may simply use them and do not have to e.g allocate system resources. It increases the development speed.

- **Improve predictability** – When a problem with a functionality of a component appears it certainly means that the component itself contains a defect. All general design rules and patterns of the whole system are defined by the component model and it is thus enforced to each component. For that reason, it is unlikely to do a mistake in the design of a whole system.

2.2 Component

The definitions of components vary and so far there is not only one general definition of what a component is [13]. The empirical definition says that a component is a unit of a composition. It is obvious without any deeper prove that we build components to use them to compose a variety of final applications.

This empirical definition is not however sufficient. There is a lot of variants how the components may look like, how they are created and how they compose a final product. The very term "component" is either used for different areas of software which have a little common characteristics together. On one hand a bean defined in Enterprise Java Bean (EJB) [16] or a bundle created for OSGi [33] evidently encapsulates functionality that may be repeatedly used in different systems. For that reason, it fits to the empirical definition of the components. On the other hand, vendors of software time-to-time claim that their product contains a set of components, but finally they sell a monolithic application. The distinction to components is, in this case, only a logical or a commercial one.

Before working with components, it is good to state a preciser definition of components. One definition is in Szyperski's book [38]. It reads:

> A software component is a unit of composition with contractually specified interfaces and explicit context dependencies only. A software component can be deployed independently and is subject to composition by third parties.

Another definition of the components is in [8]. It sums up the definition in three points:

> The component is:

1. an opaque implementation of functionality
2. subject to third-party composition
3. conformant with a component model

It is evident that this definition is partly equivalent with the Szyperski's definition.

The rest of this report deals with components in terms of these two definitions. It essentially means that a component is assumed as a unit, typically prepared by a third-party, which has a defined interface and is deployed into a system. The components model will define how the components look like.

2.2.1 Components and Object-Oriented Programming

Szyperski also describes a relation of components with object-oriented programming. Despite of component as a unit of deployment, an object is a unit of instantiation. Each object has often, after instantiation, assigned a state together with an identifier. An object is instantiated and later destroyed in any time of an application run. The state of the object may be observed while an instance of the object exists.

In opposite, the component is started (or is activated) when the whole application is started and runs until the application runs. The component should not have any observable state and for that reason it has no sense to have more copies (instances) of one component in the application. Another benefit of a stateless component is its re-entrance. Since there is no state while a component runs, each call of component's methods is independent and it is thus re-entrant. On the contrary of Szyperski's rules, component frameworks such as EJB, Spring [36] or OSGi allow to run more copies of one component.

Object-oriented languages such as Java or C# may be used to define a component as an object or a set of objects. For instance, Spring defines one component as one Java class while a component in OSGi is a set of classed packed in one JAR file. On the other hand a component may be defined in non-object languages such as C.

2.2.2 Black-box and White-box

Black-box is generally a part of a program which provides a functionality and users know only inputs and outputs. The users call the functions with inputs and except outputs. The inner implementation of the functionality remains hidden.

If the user of the program may look in the source code, it is called white-box. White-box is generally more problematic to replace an old program by a new one [38]. Once a user may study the source code of the program he or she tends to adjust client programs to use any "hidden" benefits of the code. It means that the client may e.g change a sequence of calls, modify somehow input and even output values to exhaust e.g a maximal performance of the program.

It is obvious that white-box representation of a program may pose problems when the program is replaced by another version. The new version may work perfectly well, but some clients may rely on the inner representation of the older version. The new version may e.g change a type of return values or use different algorithms which change in some performance coefficients.

For that reasons the black-box program better suites for future replacement. When claimed functionality is not changed, there is a considerable change that all clients will work with the new version without any problem, unless they rely on an inner representation of older version.

When we speak about components, the black or white-box nature remains valid. All components designed as black-boxes are more convenient for future replacement. Since the components primary goal is to be replaceable, it even more highlights the need for black-box components.

2.2.3 Component Interconnections

As was already mentioned a component is independent unit which is deployed to a system. Obviously, the component deployed to the system needs to communicate with other components. It arises a question how to define a communication mechanism, or in other worlds, how to connect components for the time they run together in the system.

Different systems use different approaches to connect components. Offten, interfaces, events, shared memory or connectors are used. Sometimes a connector is used to modify or adjust data flowing from one component to the other one. Interface is a point accessing component's functions or services. An interface usually aggregates a set of methods (often called services) which may be called independently by other components. A component may provide more than one interface.

Most component models use the provided and the required role of interfaces. The meaning is that the provided interface expresses functions the component offers. The required interface expresses a functionality the component needs.

The other purpose of interfaces is that they basically serve as a contract for components. In other worlds it informs what the component may guarantee

when the assumptions are fulfilled.

The interfaces consequently hold the contract which must be fulfilled when a component is to be replaced by another one. Most current component systems hold only information about component's functions. Section 2.5 of this work will show that describing only a function is insufficient to express all guaranties the caller must full fill.

In a real components market, the components evolve and new versions arise. These changes has often an influence on interfaces and may even break a backward compatibility. To distinguish on which version a component depends, the interfaces are often enriched by versions. The versioning is nothing new and it is widely used in current software processes. Each software vendor usually uses a versioning system concerning a combination of major, minor and micro number of the version, however each vendor uses different mechanism assigning a number for the concrete version. Despite of it, the components system needs a common system of versions which vendors of components are comply with. The common versioning system is needed to guarantee comparable versions of components shipped by different vendors.

Another question raised with interfaces is how many functions should an interface publish and how many interfaces should a component offers. An ideal component is fully re-usable with only a set of useful functions. This empirical rule is, however, often in contradiction. When a component provide a big amount of functions it is barely re-usable. On the other side, a widely re-usable component may offer only a very limited set of functions or even only a one function. Szyperski summarises:

> *Maximizing reuse minimizes use.*

Every component should offer the right set of functions with minimal dependencies on other components. So far, it is up to developers to empirically estimate size of a component.

2.3 Component Model

If we removed the third rule of the components definition form Section 2.2 saying: *"a component is conformant with a component model"*, we could claim that any two stand-alone programs are components. They are opaque implementation of functionality, independently deployable and often use means to communicate each other. For that reason the conformance with a component model is the most important addition to the world of components.

The component model gives a uniformity to components and their composition.

Its use is to define how a component should look like, how components communicate each other, which resources they use, etc. The component model ensures the components are compatible in terms of deployment, the communication, etc. It determines the rules components must hold to be able to cooperate and it minimalists misunderstood assumptions.

Another use of the component model is to ensure sufficient quality. The component model may define typical software requirements e.g. it avoids deadlock, manages race-conditions, synchronization etc. The component model may also support requirements such as performance, memory consumption, etc. generally covered by extra-functional properties.

A success of CBSE depends on the component market. When developers produce a component it is published by a vendor to the marker where it may be bought by an architect of a final application. It is generally excepted that the component works equivalently in an original developer environment as well as in an environment of a consumer. This rule is practically defined by the component model which ensures that once a component is conformant with the component model, it must work the same way in each environment supporting the same component model. A degree of such assurance depends on a component model. The level of ussurance will be quite low unless current industrial models widely support additional advanced features like extra-functional properties.

The work [8] claims that the component model should impose:

- **Component types** expressed by interfaces the component implements. When the component implements more interfaces it is of the type of all implemented interfaces. In other words the component is polymorphic with respects to all implemented interface.
- **Interaction schemes.** The component model should specify how components are located, which protocol to communicate is used and may also define which quality of services are achieved.
- **Resource binding.** Each deployed component is bound to some resources. A resource is provided by a framework the component is deployed in, or by other components. The component model describes which components are available and how and when the components bind to them. Consequently, the component model drives the life cycle of components and manages resources assignment.

2.4 Component Framework

A component framework is basically an implementation of a component model. It supports all mechanisms such as deployment, synchronization, life-cycle, communication of components which are defined in the component model.

Component framework works like an operating system. It also manages processes (components), life-cycle, receive resource requests and decides their assignment. The framework also allows components to communicate each other which an operating system also does. Operating systems typically run all the time while the processes are variously started and stooped. Although a lot of component frameworks also run all the time components are started and lately stooped, it is not necessary needed. A component framework may be also an implementation of functionality which components explicitly invokes. In contrary to the last sentence, the work [8] says: *"The trend in component technologies seems to be towards framework as independent implementation, making the operating system analogy quite apt."*. This claim is also supported by practically used frameworks such as Java EJB, Spring, OSGi, which are consistent with the operating systems analogy.

2.5 Extra-functional Properties

Section 2.3 shows that a component model ensures the components compatibility in different environments. Although current component models guarantee a component will work in a customer environment in terms of correct using of communication channels, resources binding, component's life-cycle etc., this is still not sufficient to guarantee the full functionality. Even when a particular function of a component is reached, the functionality of the component in the target environment may not be guaranted unless extra-functional properties are taken into account.

The very term functionality as is assumed from the user point of view we may essentially denote by two disjunctive parts:

1. a particular function the component has been developed for
2. a set of extra-functional properties the component is comply with

The component reaches the desired functionality if and only if both of these rules are fulfilled. It is clear a component has been developed to provide a concrete function. For that reason, the correctly working function is necessary to fulfil users needs. This requirement is necessary but not sufficient at all. The problem

of current industrial models and frameworks is a lack of the support of extra-functional properties. A components vendor may not guarantee a functionality of published components as long as he or she is not able to guarantee extra-functional characteristics together with the function. It leads to a considerable weakness which limits component-based development.

It is need to point out that the very term *extra-functional properties* lacks a standardized definition. This poses a problem in situations where the border between functional and extra-functional is fuzzy.

The definitions available, so far, are vague and varied. For example, [1] defines extra-functional properties as *"the degree to which [a component] meets requirements or customer/user needs or expectations"* . The most commonly used synonym is "quality characteristic, factor and/or attribute" [1, 2, 6] while for example [3] uses the terms "performance" and "attributes". Another term, extra-functional characteristics, is used by Franch, X. in [17].

The work [8] does not define precisely extra-functional properties, but defines three main groups of extra-functional properties instead. They are:

- **Behavior** concerns outcome of operations. Each call of a method and the outcome of the method vary depends on a call of other methods. For instance, the Eifell language allows to define pre-conditions and post-condition to capture conditions which must hold to guarantee a result of the computation. The other example is a usage of *assert* commands in languages such as Java. It typically guards whether input parameters of methods contain valid values. In any case, the behaviour characteristics concern sequential ordering of methods call.

- **Synchronization** concerns all aspects connected with multi-threaded computation. Although modern programming languages contain means to deal with synchronisation – they allow programmers to define semaphores, monitors, lock shared resources, etc. – the formal verification checking whether a component is thread-safe and synchronised has been barely exploited.

- **Quality of service** typically concerns attributes limited by hardware or any other technical means. Quality of service includes attributes such as maximum response time, delay, average response, memory usage, processor speed demands, precision. They are mainly relevant in resolving whether the whole component system will work with available platform properties.

This distribution does not explicitly name non-functional requirements (abbreviated as NFR). NFR covers all characteristics expected from a user point of view. It includes both, user requirements and technical means concerning quality.

Users typically mix functional and extra-functional requirements when they define their requirements on the system. It first is necessary to sort out which requirements belongs to each group. The group of extra-functional requirements must be then divided to other groups covering quality of software, behaviour, synchronization, etc., because mechanisms working with each group usually differ.

The current research aims at expressing at least a subset of extra-functional requirements by extra-functional properties (EFPs). We show the distribution of

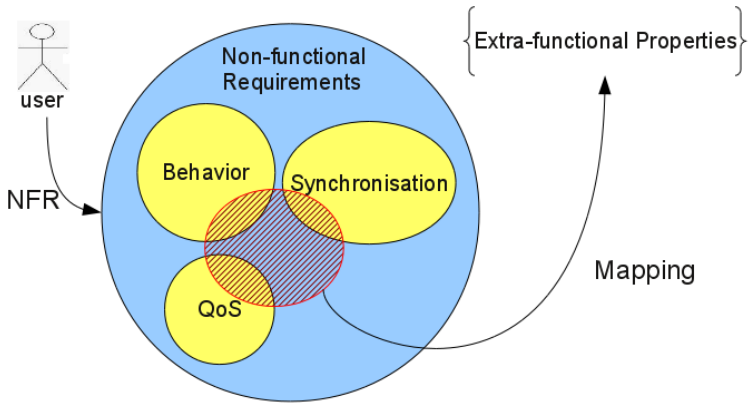

Figure 2.1: Extra-functional Properties Overview

NFR set ($NFRset$) into other disjunctive sets in Figure 2.1. When the sets are defined, an extraction of desired extra-functional properties may be performed. The definition of EFPs allows a formal expression of non-functional aspects (together with a function of components). It allows to check whether a component matches NFR. When component compatibility checks are improved by EFPs, it leads to more precise decisions whether one component is suitable as a replacement of the other one.

Note that we does not try to precisely target a complete distribution of the $NFRset$ in Figure 2.1. The dimension of the NFR set has not been determined yet and for that reason, the distribution of the whole set is unknown. From our point of view, the following essentialy holds: $NFRset \supset behaviour + synchronization + QoS$ and $NFRset - (behavior + synchronization + QoS) \neq \{\}$. It means that *behavior*, *synchronisation* ans *QoS* do not cover the whole $NFRset$. For instance, a user may require an easy-to-use graphical interface which is evidently a NFR, but it does not fit into any mentioned group.

Figure 2.1 also shows that we may probably map – at least in the current stage of the research) – only a subset of NFR.

Concerning the multitude of existing terms, we try to propose one general defi-

nition:

Definition 1 *An* extra-functional property *of a software item (including a component) is its characteristic which (1) concerns the item's clients or even end-users, and (2) is not an invokable functionality.*

In this work we follow the trend expressing components connected through interfaces. Whereas interfaces inform others about their functions, it is ideally suitable to contain informations about EFPs. Provided an required side of a component expresses a functionality covering a function and extra-function pair. For that reason, we assume each provided and required method enriched by a set of their EFPs to complement this pair.

Chapter 3

Extra-functional Properties Approaches

Previous chapter described why it is good to think of extra-functional properties when designing component systems. EFPs are, however, not easy to treat. There are a lot of problems which have been addressed by current research but non of them reached its maturity. EFPs are difficult for practical usage. Some typical challenges related with EFPs are:

1. A transformation of user requirements, expressed in a natural language, to a formal language [21];
2. A distinction of all NFRs into disjunctive groups;
3. A language or any other formalisation of EFPs allowing their general automatic processing [29, 4];
4. Systematic way computing how EFPs are influenced by other EFPs. [40, 15];
5. A relevance of EFPs to a concrete area of usage [22].

The first two points, though they are important, will not be addressed in this work. The following sections introduce mainly approaches that aim at formalisation of EFPs.

The last two points are very important in component-based programming. Each component provides a set of functions and it is desired these functions provide the same results for the same inputs all the time independently of a context of usages. It is not the case with EFPs. Some qualitative aspects of the systems often vary for different usage. For that reason some EFPs may be highly relevant

in one context while they are barely relevant in another one. In addition, some EFPs may be influenced by other EFPs and for that reason they should not be treated in isolation. Our work presented in Chapter 4 will explain our current attempt to solve these two points.

3.1 Extra-functional Languages

This section introduces current research approaches to a formal definition of extra-functional properties. They are mainly expressed as a formal language. Different approaches focus on different degree of granularity.

There is a group of works preparing EFPs only for a concrete area of usage. Their approaches do not have to solve the context dependency and are not targeted for a general usage. On the other hand, there are other approaches preparing general formalisations of EFPs. They are typically more suitable for components and their context independent usage, though they do not mature to practical usage yet.

3.1.1 Specialised Languages

This section focuses on few approaches which defines EFPs formalisation for concrete areas of usage and thus are not general. They generally better succeed in a practical application, but are worse in combination with components for component general usage purposes.

HQML

A Hierarchical QoS Markup Language (HQML) [29] is designed as a XML-based language targeted to the World Wide Web. Although the internet used to serve for exchange textural information only, nowadays it also provides a variety of services remotely invoked. HQML aims at describing quality of service (QoS) for the services accessible via the internet. HQML uses a XML language for its simplicity and popularity.

HQML uses three-layered structure:

- **User Level** – defines quantitative criteria in a textural representation (e.g. “high”, “low”, “average”), an attention (“clarity”, “smoothnest”) and a price from a user point of view. This level is used during runtime when the best suitable service is matched.

- **Application Level** – this layer serve as a specification of all king of application QoS (e.g frame rate, resolution, size). It also allows a connection of a distributed application expressed in an oriented acyclic graph. The main use of this level is for a middle-ware entities of the system independently on underlying resources such as hardware, OS, etc.
- **System Resource Level** – defines different resources requirements. When a concrete resource is available it allows to associate it.

The XML representation is translated to an in-memory representation where it is processed by *QoS-proxies* which provide a generic middle-ware representation of QoS-services (negotiation, adoption, ...). The transformation of XML data into memory ensures the HQML Executor that consequently cooperate with QoS-proxies in QoS negotiation. HQML Executor works in following steps:

1. The HQML Executor interprets users requirements (from User Level) and contacts QoS-proxies to discover current application resource availabilities. The request is sent to a server.
2. Web/HQML server search in HQML Profiles (which is Application Level) to find a profile matching users requirements. It returns information about suitable services or returns an error if the matching does not exist. The result is returned back to the user.
3. In a case more than one profile matches the user is asked for a selection. When the suitable profile is selected, the selected service is invoked together with allocating demanded resources (described by Resource Level).

Together with the HQML language, they propose a tool *QoSTalk* covering the presented solution.

HQML seems to serve as a comprehensive language which targets different level of an application. Although the paper [29] addresses a mechanism of evaluation of defined properties mediated by QoS-proxies, it does not explain how the QoS-proxies work to define a precise evaluation mechanism. It is desirable to know whether the HQML mechanism directly compares values provided on each level, or matches the best suitable services. The former one is easy to cope with while the last one is challenging.

SLang

SLang [24] stands for a Language for Service level agreement (SLA). The language is targeted at systems concerning web services providing data among systems, component-based middle-ware and containers accessing system resources

and data storages. SLang aims at capturing different scales of extra-functional properties for different tiers of an application and different scales of the properties among applications.

Slang first captures inter-organisation EFPs with respects to a storage, network, middle-ware and an application level. It second captures EFPs between a services provider and a client.

It defines a *horizontal layer* which basically copes with a layered structure of classic applications. It separately defines EFPs for each layer: layer of web services, middle-ware components and a container. The rationale behind it is that each layer may use the same EFPs, but they are far from being the same in each layer. For instance, a web service may offer a throughput as well as a database may do, but scales of values for both layers differ. The other, a *vertical layer*, concerns EFPs of the same layers for different systems, e.g. the properties of two web services of two communicating application or two components on the middle-ware layer. It expresses EFPs a server must meet to satisfy clients.

The main goal of SLang is (i) to express qualitative and quantitative features of a service with the high degree of accuracy, (ii) make easily comparison of offers. They define a set of main concepts to reach both goals:

- **Parameterisation** – each SLA is parametrised by values that quantitatively describes a service.
- **Compositionality** – since services may be cascaded or aggregated, SLAs of the services must be also composable in order to express an offer of the composed service.
- **Validation** – a syntax and validity of SLA must be feasible.
- **Monitoring** – SLA should be able to provide automated monitors showing which service levels are met.
- **Enforcement** – an execution must be enforced when service levels are agreed.

Although these points are mentioned in [24], it is not stated how SLang reaches them. The work describes mainly the language itself rather than evaluation mechanisms.

The two layers (horizontal and vertical one) of SLang contain seven different kind of SLA – four for the vertical layer and three for the horizontal layer.

The vertical layer uses the SLA of kinds:

- **Application** – between application or web services and components

- **Hosting** – between a container and components
- **Persistence** – between a container and a storage
- **Communication** – between a container and network providers

The horizontal layer uses the SLA of kinds

- **Service** – between components and web services
- **Container** – between containers
- **Networking** – between network providers

SLang targets different scales of values in term of (i) values used in different layers of an application and (ii) values used among applications in which each domain covers different scales of values. The domain dependency of values is important to cope with. Although SLang allows to bind properties to a concrete feature in which properties are valid, the work [24] does not state the mapping of values each other. Since concrete feature contains its values the matching would be performed with incompatible values and for that reason they should be re-mapped or somehow consolidated. Despite of it, this challenge is not deeper developed in the work.

TADL

An Architecture Description Language for Trustworthy Component-Based Systems (abbreviated as TADL) [26] is a specialised language describing the whole architecture of a system. TADL is a language specialised for trustworthy system and explicitly concerns extra-functional properties as a part of an architecture of systems. It specifically targets structural, functional and extra-functional properties to define a system's architecture.

In addition to structural and functional characteristics of the system, TADL defines *safety* and *security* representing extra-functional properties. The detailed specification of an architecture is denoted by explicit specification of services, data parameters, contracts and architectures at the interface level.

The services are provided via interfaces which is typical for other approaches but it, in addition, explicitly defines data parameters expressing a data coming through services. The benefit of an explicit definition of data is a possibility of guarding a validity of values. It is used for increasing the security of the system and also allows the system to react to specific values.

Services may include constraints that are invariants defined as first-order predicate logic. Specifically, the safety contract is reached by a different kind of services:

1. *Regulating service*: enables real-time schedulability. The response of a component is regulated by *time constraints*. Time constraints guard the time consumed by the execution of the service and do not allow to exceed a set value.

2. *Restricting service*: all data coming through services are restricted by *data constraints*, which decide a request that should be sent.

3. Filtering service: a response is filtered according to the security rules. A request is maintained by a component's service or a response is provided by a component's service only if the user has the right access privileges.

TADL deals with extra-functional properties at a design time of an application. The considerable drawback is that the set of supported extra-functional properties is very limited. It defines only two types of EFPs: security and safety. In addition, it is a question whether security belongs to functional or extra-functional characteristics.

3.1.2 General Languages

This section provides an overview of approaches that aim at providing a general mechanisms dealing with extra-functional properties. They are typically languages expressing EFPs as stand-alone notations that may be used by various systems.

NoFun

The NoFun [17] language is a representative of structured extra-functional property definition approach, stemming from the component field but applicable to general software systems. The authors of NoFun have identified three concepts of extra-functionality: *Non-functional attribute, Non-functional behaviour* and *Non-functional requirement*. The meaning is as follows.

Non-functional Attributes These are attributes of any kind which can be used to describe or measure a software system. Every attribute belongs to a data type which determines the set of valid operations and values. The available data

types are standard types such as Boolean, Integer, Real, String, plus structured types Enumeration and Mapping.

The values stored in an enumerated type can be ordered and some additional operations are available (<, >. <=, >=, *max* and *min*).

Attributes may be basic or derived. Derived attributes are derived from basic ones. Basic attributes belong to the data type which defines them. Derived attributes are computed by the equation 3.1:

$$C_i \Rightarrow P = E_i \tag{3.1}$$

The value P is a derived attribute which is equal to an expression E_i if the boolean condition C_i is *true*. E_i yields a value in a P's domain.

This example shows the computation (matching the equation 3.1) of the derived attribute *reliability* depending on two simple attributes: *error recovery* and *fully portable* (The example has been copied from [17]):

```
not error_recovery and not fully_portable => reliability = none
error_recovery and not fully portable => reliability = low
...
```

Every attribute may be bound to the whole component or only to an individual operation. An attribute may be also the derived one in the meaning that this attribute is composed of basic attributes bound to *all* operations of the component.

Non-functional Behaviour NoFun separates the definition of extra-functional attributes from their application on a particular component. This is allowed by the behaviour specification in which particular attributes are bound to a component.

This way separates the definition of extra-functional attributes from the demand of the concrete component described by the behaviour specification. In addition, it allows reusing definitions of non-functional attributes for other components.

This example shows a definition of the behaviour specification (copied from [17]):

```
behaviour module for IMPL_LIBRARY
behaviour
 time(list_all_members) = n_members
 time(check_out) = log(n_books)
end
```

Note that the exact meaning of the lines of the example shown above is not clearly explained in [17]. *List_all_members* and *check_out* are operations and *n_members* and *n_books* are any measurable units. *N_members* holds the number of members and *n_book* holds the number of books.

Non-functional Requirements These are used in the situation in which components are assembled. If the component has the behaviour specification and needs any other component to work together, the component must specify which behaviour it demands from the behaviour specification of the other component. These demands are specified by non-functional requirements. In essence, non-functional requirements say which EFPs are demanded on the required side of a component.

NoFun provides ideas of which information should an EFP contains. It is a good base for developing other more sophisticated solutions. Although NoFun provides a description of assignment of EFPs to components as well as expressing a demands among other components, it is barely explained in [17]. They uses operators and functions in the notation that are not briefly defined and hence a semantic and the complete set of allowed operators and functions remain unclear.

QML

A language called QML [18] is specialised for all systems that comply with an object-oriented approach. It attaches QoS specifications to interfaces and it is designed to conform with objects, interface and inheritance features of object-oriented programming.

QML aims at fulfilling these goals:

- A specification of QoS is separated from the code of an existing system
- It allows to specify provided and required QoS properties
- It provides mechanisms to determine whether the client needs for QoS are fulfilled
- It supports a refinement of QoS, because object-oriented approach uses inheritance and inherited objects may need to work with modified QoS. QML allows to inherit and modify QoS properties of inherited objects.

The main building blocks of QML consist of

Contracts and Contracts Types A contract contains a list of constraints. Each constraint is associated with a dimension selected from a set of *enum*, *numeric*, *set*. A constraint is a tuple consisting of a name, an operator and a value (e.g. $memory < 100$). A name is typically the name of a dimension.

Aspects Aspects are used to characterise measured values over a time period. The predefined aspects are: *percentile*, *mean*, *variance*, *frequency*.

Definitions of Contracts and Congrats Types It binds a name to the value of a contract or a contract type.

Profiles A profile holds the QoS properties for services. The profile is specified for an interface and the interface may assign more profiles for different implementations. A profile is used for expressing provided and required QoS the properties of the interface.

Definitions of Profiles It is used for assigning a profile to a service and gives the profile a name.

Conformance QML defines conformance for profiles, congrats and constraints. A general rule is that a stronger rule conforms to a weaker rule. A target is to find a service witch suites a client rather than exact match. To achieve this goal QML uses an ordering of set, increasing or decreasing ordering of numbers etc.

QML binds profiles to interfaces statically. In addition QML allows to define QoS-aware object which may use statically defined QoS, but may also define QoS dynamical. The dynamic creation of QoS is achieved by QRR (QML-based QoS Fabric) which creates QoS properties at runtime while QML does it the same way statically.

QML is a comprehensive language covering creation of EFPs and attaching them to objects that are typically components. They provide a run-time mechanism of constructing EFPs. Although they target different run-time environment by defining different profiles for each environment, the profiles must be manually re-attached. The QRR seems to be able to dynamically attach EFPs for an individual environment, but they do no describe a mechanism configuring objects automatically for different environment.

Ontology

In the field of Service Oriented Architectures where quality of service (QoS) and Service Level Agreement (SLA) are an important issue the community aims at providing different kinds of ontologies that captures EFPs. Ontologies allow to express EFPs with respects to theirs semantics and relations each other.

For instance, [39] extends the Web Service Modeling Ontology (WSMO)[1] to better support EFPs and propose a service comparison method using quality characteristics.

[1] http://www.wsmo.org/

Another work, [19] developed a reasoning framework in which a user query is evaluated by a selected engine. The user first inputs a query concerning QoS and a scheduler selects the most suitable engine. The engine then evaluates the query into a result. The result may possible be an empty set, the best offer, or an ordered list of offers by an optimality criterion. The scheduler works with a knowledge base which caches results to improve performance of the reasoner.

The constraint programming in combination with logic programming is used in [20]. They use WSMO to express QoS. The WSM language (WSML) axioms are used for defining EFPs. Each EFPs has attached a number expressing its importance – a weight of the property. A user may express EFPs in both terms: logical programming rules and constraint programming. Both of them are separately evaluated. The results are sorted and the most ranked service is selected.

Our work follows similar goals using more traditional means.

CQML

An approach proposed by Aagedal is the CQML [4] language. He has described a complete syntax of an EFPs language and introduced a UML profile for quality attributes. The CQML approach is a language usable for general description of EFPs. The language defines basic data types: Number, Enum or Set. There is no complex type (record) provided. CQML also provides derived properties, but they are meant only to extend an existing simple property or to compose a derived property from other ones without any further definition how this composition is treated.

CQML defines few basic constructs concerning EFPs:

QoS Characteristics is a basic building block. One QoS characteristics represents one EFP. It contains a unique name and a data type of the property. Depending on the data type, it may contain additional information such as a restrictive interval for values, ordering of enums, measuring unit, etc. Additionally, it may define invariants for values of the property. Values are passed through as input parameters. However, it is not stated how one can define an input parameter of a property when the property is defined independently of a targeted system. Consequently, an input value may not exist in the time the QoS characteristic is being developed.

QoS Statement assigns constraints to QoS characteristics. A constraint is expressed using logical rules. There may be also added other modifiers e.g. best-effort, compulsory, threshold to complement the constraint. Each statement is

enhanced by the name and encapsulates a set of constraints for a set of QoS. A set of QoS with their constraints is then refereed by this name.

QoS Profile is used for aggregating a set of QoS statements into one record with a unique name. A component links a profile to attach the QoS that the component works with. QoS profile defines with QoS statements are used or provided by the component. The benefit of this solution is that the profile may be re-used by other components and the underlying definitions of EFPs may not be repeated. On the other hand a need for the same EFPs and their constraints by more components seems to be rare, and a separate profile must be defined for each components even if one EFP or its constraint differs.

CQML assigns a *profile* to a component. The profile contains a set of *qualities* with a set of QoS properties. The *quality* allows to encapsulate context dependent values, but assuming we have c contexts and n QoS properties it may produce up to 2^n quality records and 2^{2^n} different profiles. In addition, each profile must be created for c contexts. This may lead to a hardly manageable number of records.

CQML+

Components are designed to use or serve to other components. For that reason, the typical relation in a component world is the relation to other components. However resources available in different environments indeed influence components running in and thus the relation to the environment may not be avoided.

The CQML mentioned before has been extended by other authors. An extending language proposed by Röttger and Zschaler is called CQML+ [34]. They aim at an explicit definition of resources needed by components. They consider not only demands between components but also demands between a component and a system (framework or hardware) called as resources. Their work allows an explicit expression of relations to the deployment environment.

As an addition to CQML, they propose a meta-model including an abstract *Resource* class. The class may be instantiated by concrete resources such as memory, cpu, network etc. This allows a user to define an infinite set of different resources, but CQML+ lacks of describing a mechanism of evaluation these resources. This is evidently a drawback, because different resources must be treated differently and one generalised mechanism would be interesting.

Another extension to CQML, they introduced, is a definition of a tuple that allows to associate more resources in a one. The semantics is that all resources in the tuple must be available concurrently.

To conclude, CQML+ extends syntax of original CQML rather than providing a more generalised mechanism of expressing the resources between components

and environment.

Deployment Contracts

Deployment Contracts [23] presented by V. Ukis are focused on detecting possible conflicts among components or a component and its execution environment. It targets the same issue as CQML+ does.

Deployment Contracts (DCs) defines a comprehensive set of meta-data describing (i) environmental dependencies of components and (ii) components threading models. The description of (i) consists of specification which resources a component requires and how it accesses them (e.g. read-write exclusive access or read-only shared access to a file). The description of (ii) includes various aspects of a component regarding threading issues and concurrency (e.g. whether a component spawns a thread, or whether a component assumes to be executed in a single thread). These meta-data have the form of parametrised attributes that can be attached to a component, a components method, a methods parameter or a return value. In the prototype of DC, meta-data are implemented as .NET annotations.

Components' DC is checked against the specification of the execution environment in component deployment phase in order to prevent possible run-time conflicts.

DCs is defined only in terms of annotations for .NET but no formal definition of DCs is specified. They also provides an implementation of about 100 different deployment contracts specified in [25] with an algorithm of evaluating them in [25]. The algorithm, however, branches to evaluate every case of implemented DC rather than generalising in a simple sequence of steps. The algorithm itself focuses on the conflict prevention rather than selecting the most suitable component candidate as in our case. DC might be considered as EFPs of a certain kind and for that reason we aim at using DC attributes in our work, but we create a general formalism for them which is consistent with EFPs.

3.2 Frameworks Support for Extra-functional Properties

Although the previous section introduced a lot of approaches expressing EFPs, the main weakness of the introduced languages is theirs lack of relevance to the components. The expressiveness of the languages provides the comprehensive ability to define EFPs separately of the (component) system, however, it does not addresses how component systems should treat these EFPs

This section shows component models that works with extra-functional properties in a certain degree. So far none of the frameworks acquire any presented EFP language and instead define EFPs using their own means.

3.2.1 Palladio

Palladio [10] targets whole development process in component-based development. It includes roles of a component developer, a system architect, a system deplorer and a domain expert. A system in Palladio is modelled by a set of models where each model covers the different role.

The different role is distributed in terms of EFPs: (i) a component developer implements a component and attaches a parametric properties of behaviour, (ii) a software architect estimates components EFPs from a component specification, (iii) a system developer models the resource environment to allocate different resources for components in different environment, (iv) a domain expert provides a usage model describing critical as well as typical parameters of the system.

The detailed scenario of the development process looks like: A component developer annotate each provided service of a component (a method of one of provided interfaces) with an additional specification called Resource Demanding Service Effect Specification (RDSES). RDSES is in practise a modified UML activity diagram. Its use is to describe a simplified control flow of the service, it can express the service's dependencies on input arguments and resource demands on abstract resource types stored in the global resource repository. RDSES describes the flow only for parts called by or calling other components. This concept Palladio names as gray-box. Simple components may be composed into hierarchical components by a software architect.

In further phases of system development the resource types in RDSES are parametrised by a resource model (the role of an system deployer), which binds the abstract resource types to concrete service's resource demands in a target resource container.

A domain expert role is to define system usage, a workload or a behaviour of the system. A usage model is used for describing service's usage scenarios and anticipated workload. In the end, all models composed together can be used for component's and system performance prediction.

Palladio focuses only on performance-related EFPs for whose specification it provides a rich palette of models. Specifically, EFPs' values defined as random variables and taking usage profiles into account are strong concepts. On the other hand, the necessity to create a number of detailed models imposes a significant burden on system and component developers. Moreover, resource platform specification in the form of a resource model has to be created for each system from

scratch since the resource repository contains only resource types, not particular instances with performance characteristics.

3.2.2 Robocop

The ROBOCOP model [28, 11] uses multi-layered components which contain specifications, models, and executable code within the component distribution package. The approach allows performance analysis by combining static analysis and simulation on the executable system model provided by the development framework and the execution framework.

Development framework defines aspects of the development trading and downloading of components. The developed components are generic in the sense they must be tailored to fit in a concrete environment. Execution framework defines the middle-ware layer of single devices.

A component in Robocop consists of a set of models including a resource model, a simulation model, an executable model. Extra-functional properties are contained in the *Resource model*. ROBOCOP components can specify only processor or memory utilization on operations, with best, mean and worst cases distinguished. This is a limited extent typical for the domain, however, combined with the performance model of hardware blocks it allows the above mentioned analyses.

To sum up, the ROBOCOP component model provides a comprehensive support in the field of specialised embedded devices demanding mostly system resources.

3.2.3 ProCom

An approach to integrate EFPs in component models using structured attributes is presented in [35] and implemented in the ProCom component model. ProCom's attributes comprise multiple values, each of which is further composed of *data*, *meta-data* and *validity conditions* parts. The data part contains the actual value of a measured EFP of the type specified in the attribute definition in the Attribute Type Registry. The meta-data part is used for distinguishing a particular attribute value and for its description (e.g. the source of a value, a degree of importance). Validity conditions specify in which contexts an attribute value is valid in terms of platform, usage profile or inter-attribute dependencies. The attributes are stored in a general repository that aims at avoiding duplicity of attributes and providing a unified storage.

The proposed structure of attributes can lead to complex EFP descriptions that are hard to manage without extensive tool support. The authors try to address these problems by introducing a language for defining which values are valid

based on the current configuration (so-called *configuration filters*). However, this makes the whole system even more complicated.

Furthermore, while ProCom attributes are meant to be used during the whole system life cycle, which motivated introducing multi-valued attributes, we are interested in describing EFPs of the final black-box components. The most interesting idea in ProCom is the usage of registries storing EFPs. The main reason for introducing registries is to gather attribute types.

3.2.4 Enterprise Java Bean

This part discusses the type of support for extra-functional specifications that can be expected from an enterprise component framework. It is intuitively clear that the needs in this area are different from those in embedded and real-time domains. The emphasis in this class of systems is on "horizontal" aspects such as security and (transparent) distribution.

In particular, the Enterprise JavaBeans [37] component model is one of the strongest industrial frameworks, used in the application and data layers of enterprise applications. Despite its focus on the functionality of these applications, the model works with several properties that can be classified as extra-functional:

- **Locality** – a global property of a component is whether its operations can be accessed remotely or only by clients local in the same container.
- **State** – a session bean (which clients use to invoke functionality in a synchronous manner) can be either stateless or stateful, with consequences for the client's view of the operations behaviour and for bean pooling in the container.
- **Transaction demarcation** – for a bean's operation, it defines the level of transaction support expected, ranging from *never* to *required* and *mandatory* in which the client must provide a transaction context for the operation. This holds for a *container-managed* demarcation, the other option is that the bean handles transaction contexts internally.
- **Security** – involves the definition of client roles and their access to bean's operation; plus a bean can be designated to run under a different identity than that of the original request.

The technology uses a combination of XML-based specification of the EFPs (in the bean's deployment descriptor) and annotation-based specification in the bean's source code. There is no formal model that would underpin the property

specifications, and the values can be seen as being of boolean or enum types (when abstracted of the form in which they are specified).

Enforcement of the properties is done partly by design of the EJB application, partly on the part of the container (both as implementation artefacts it generates and run-time checks it uses).

3.3 Summary of EFPs Languages and Frameworks

Figure 3.1 summarises our survey of the state-of-the-art. A desired approach should (i) allow general definition of EFPs, (ii) deal with context and domain dependency of components (iii) be easy to use, and (iv) allow to express a dependency both on other components and on the environment. The table shows how current works fulfil our needs and which requirements are missing.

Framework	General	Context Independent	Easy-to-Use	DC
NoFun	√		√	
CQML	√	√		
CQML+	√	√		√
Ukis's DC			√	√
TADL			√	
HQML			√	
SLang			√	
Palladio		√	?	√
Robocop			?	√
ProCom	√	√		
EJB			√	

Figure 3.1: Important attributes of existing approaches

3.4 Modeling of Extra-functional Properties

An important aspect of the development is a support of modeling. Current applications are often modelled before they are developed. It allows a better understanding of the system, its parts and their connections. Inspired by the classical modeling means, a lot of work aims at modeling of EFPs.

Whereas UML [32] has been acquired as a widely used modeling notation, other works introduced the modeling based on UML. UML provides the rich palette of

UML diagrams, but a class diagram is most often used. It leads others to prepare diagrams based on UML's class diagram. the UML concept of *stereotypes* allows to extend the basic elements of UML to support EFPs.

3.4.1 UML Profile for CQML

Aagedal introduced [4], together with CQML, a UML profile covering the expressiveness of CQML. He defines a set of stereotypes that correspond to CQML keywords (including *QoSStatement*, *QoSCharacteristics*, *QoSProfile*, *QoSQuality*). The introduction of these stereotypes allows to model EFPs the same way as they are written in the CQML's language. Hence, the CQML's profile is coupled with CQML.

3.4.2 UML Profile for NoFun

Another work has been presented by Guadalupe Salazar-Zárate and Pere Botella [12]. They introduced UML profile coupled with NoFun. The stereotypes they defined cover NoFun concepts.

They first defines a stereotype *NF-attribute* for non-functionality. In other words, a *NF-attribute* is an EFP. *NF-attributes* model simple properties as well as derived properties. When a derived property is modelled, the stereotype *import* is used for importing an aggregation of other properties to this derived one. Another stereotype, *OCL-expression*, defines rules deriving the derived properly. They second define stereotypes *NF-Requirements* and *NF-behavior* with an obvious meaning in terms of NoFun concepts. Furthermore a set of EFPs expressed in a NF-behavior is attached to a class labelled by the stereotype *ImplementationClass* and the connection is labelled by the stereotype *has behavior*.

3.4.3 Marte UML Profile

UML Profile for MARTE (The Modeling and Analysis of Real-Time and Embedded systems) [31] has been introduced by the QMG group and has already became a standard. It has been develop to replace an older profile – the UML profile for Schedulability, Performance and Time – also issued by the OMG group.

The profile serves for model-based development of real-time and embedded systems. It consists of a lot of extensions of UML covering real-time and embedded (RTE) applications. A considerable amount of the extensions are targeted at non-functional aspects of RTE. Non-functional aspects are classified to qualitative and quantitative ones. The aspects may be available at different levels of

abstraction. Finally, these aspects provide modeling or analysis support or they may provide both.

Hence, MARTE is organised as a hierarchy of profiles and subprofiles. The fundamental profiles comprise :

- **Non-functional properties** – it provides constructs for declaring, qualifying, and applying non-functional aspects. Each EFP is modelled as a UML data type. For the definition of EFP values, the Value Specific Language (VSL) has been proposed. VSL also defines potential functional relations of EFPs.
- **Time** – it allows the definition of time and it also deals with a time representation in applications.
- **Resources** – it deals with resources the applications demand from the system.
- **Allocation Modeling** – it is used for allocating of functionality to responsible entities

These fundamental profiles are then used for model-based design and model-based analysis.

Model-based design proceeds mostly in a declarative way. It means that the users of MARTE annotate their models with RTE properties. They use the *High-level Application Modeling* sub-profile. Note that component-based systems are explicitly supported by the *Generic Component Model* profile.

Model-based analysis is allowed mainly by *Quantitative Analysis Modeling* or by its two refinements (*SAM* or *PAM*) used for schedulability and performance analysis respectively. The annotation mechanism uses the UML stereotypes where the UML elements modeling an application correspond to the analysing domain.

In contrary to previously mentioned UML profiles, the MARTE profile is more general. The previous UML profiles express the notation of underlying languages while MARTE is not coupled with a concrete language. Still, MARTE is not general and it is targeted particularly to the domain of real-time systems.

3.4.4 OMG's Quality of Service Profile

Another work, proposed also by the OMG group, aims at providing a general model for QoS. UML Profile for Modeling Quality of Service and Fault Tolerance [30] provides the ability to model EFPs by the means of UML. This profile introduces new stereotypes that covers elements of extra-functionality and theirs relations.

The profile consists of several main building blocks.

QoSCharacteristics represents quantifiable characteristic of a service. It is basically an extra-functional property. The QosCharacteristics may first have a set of parameters (*QoSParameter*). Each characteristic second has a dimension (*QoSDimension*) that stores: a data type, ordering of the values, measuring unit and so called statistical qualifier. The statistical qualifiers are: min, max, range, mean, variance, standard deviation, percentile, frequency, moment, and distribution. A characteristic may furthermore be assigned to a category (*QoSCategory*). The categories are used for dividing properties to groups. Each QoS characteristic is inherited from a context (*QoSContext*) informing in which context the QoS characteristic is valid.

Each QoSDimension has assigned a value (*QoSValue*) through a dimension slot (*QoSDimensionSlot*). In addition, each value has assigned a constraint that evaluate its validity.

QoSConstraint defines constraints of the *QoSCharacteristics*. It limits values allowed for application requirements. There are other classes inherited from *QoSConstraint*: *QoSRequired*, *QoSOffered* and *QoSConcrat*. Theirs usage is respectively for what application services require, what they offer and finally the agreement between all constraints.

QoSLevel is used in situations where an application provides a variety of extra-functional properties. For instance, an application may provide different algorithms or configuration that lead to different properties. For that reason a set of *QoSConstraint* is bound to a set of *QoSLevel* allowing to switch from one level to another one. This mechanism is furthermore supported by *QoSTransition* that holds transitions between layers.

3.4.5 Component Quality Model

In a case we would be able to describe EFPs using any presented language or model EFPs using any presented profile, the question remaining open is which properties to define. Since the general consensus still does not exist, the works [6, 7] categorise component quality characteristics which can be used as EFPs.

The authors follow the standard terminology defined by ISO/IEC 9126 [2], but they made a few modifications to better fit component-based development. The resulting Component Quality Model (CQM) is composed of:

1. Functionality: the ability to provide the required service
2. Reliability: the ability to maintain the specific level of performance

3. Usability: the ability to be understood, learned, used, configured and executed
4. Efficiency: the ability to provide appropriate performance, relative to amount of resources
5. Maintainability: the ability to be modified
6. Portability: the ability to be transformed across environment
7. Marketability: the marketing characteristics

The characteristics mentioned above are then split into more detailed subcharacteristics. Additionally, the characteristics are distinguished as either runtime or life-cycle ones.

According to [2], an attribute is a measurable (physical or abstract) property and as such every attribute needs to define a metric. The metric defines both the measurement method and the scale. The CQM uses following metrics:

1. *Presence* to indicate whether an attribute is present, if so, the *string* value contains information how the attribute is implemented.
2. *IValues* to indicate exact values. It is described by an *integer* variable and a *string* indicating the unit.
3. *Ratio* shows percentages measured from 0 to 100 by an integer variable.

In addition to CQM characteristics, the authors have defined *additional information* linked to a particular component. These additional information are: Technical information (Component version, Programming language, Patterns, Lines of code and Technical support) and Organisation information (CMMI level and Organisation's reputation). Technical information is important for developers while organisation information is important for customers. The authors suppose those information to be provided by the component vendor, usually as string values.

3.5 Evaluation of Extra-functional Properties

Although previous sections have presented a rich support of the definition and modeling of EFPs, it still lacks evaluation of EFPs. The previous works mostly treat EFPs as standalone definitions. The purpose of this section is to overview other works that take evaluation of EFPs into account.

3.5.1 QoS Negotiation

Generally, a process when an appropriate component or service is selected at run-time based on its QoS (EFPs), is called QoS (EFPs) negotiation.

Mesfin Mulugeta and Alexander Schill introduce in their work [27] a QoS negotiation framework that defines EFPs using CQML+. The main architectural block of the framework is *Negotiator*. In order to select a service, the Negotiator needs a reference to: (i) QoS specifications of all cooperating components, (ii) users QoS requirements and preferences, (iii) available resources, (iv) network and container properties, and (v) policy constraints.

To achieve these needs, the framework contains other architectural blocks. QoS for all components are stored in profiles implemented as CQML+'s *QoSProfile*. Network channels expressing a communication between components are explicitly modelled by *Connectors*. It provides Negotiator with information about QoS of communication links that may also have an impact to the services selection. Furthermore, the framework allows to model *Resources* where any change in a resource may trigger re-negotiation. User requirements are expressed in *user profile*. The user profile is constructed by the run-time system after obtaining the user's requests for the service.

When QoSProfiles and a user profile are established, *Negotiator* may find an appropriate service. The task is to find an appropriate service and select the best one in case there is more suitable services. *Negotiator* relies on Constraint Satisfaction Optimization Problem (CSOP). A CSOP comprise a variables, constraints and objective functions. The task in CSOP is to assign a value to each variable to satisfy all constraints. All suitable results are first mapped to the ordered set of numbers expressing the weight of the result. The most suitable service is then selected among the mapped number.

When a selection of an appropriate service is finished, *Contract* is established. *Concract* holds mainly information about the selected client profile, the server-side profile and the user profile.

The presented framework introduced a complete mechanism of QoS negotiation. Unfortunately, the mechanism suits only service-oriented architecture. Although authors speak about components in their work, they probably thought only about server-side components. The profiles assigned to services contains concrete (context-dependent) values that may work only when a provider of the services has a full control of the run-time environment. When the components are hidden on the server, the context-dependent values may be guaranteed. The most common case, independently deployable components, does not allow such guaranties.

3.5.2 QoS Dependency

The work [40] introduced a preliminary approach to model EFPs by functions that are evaluated at run-time. The mechanism allows to compute concrete values of EFPs from input parameters. For instance, they proposed an example based on the CQML language in which the input parametr determines resulting value:

```
profile response_times for ImageStreamEncoder {
  qos_dependency
    response_time (encodedImages.getNextImage) =
    response_time (unencodedImages.getNextImage) + 5;
}
```

In addition, they noticed a weakness of this solution that is the concrete value "5". As a solution they suggested an improvement that uses intervals instead of concrete numbers. The improved example looks like:

```
profile response_times for ImageStreamEncoder {
  qos_dependency
    response_time (encodedImages.getNextImage) =
    response_time (unencodedImages.getNextImage) + [5,10];
}
```

The second example shows an important aspect. When we define EFPs, we often prefer approximated values rather than concrete numbers. For instance, when a system requires a service with a response time equal to 5ms, it would probably accept also a response time equal to 6ms. This rationale leads to an idea of defining intervals, or – in other worlds – limiting values.

Another work presented in [15] introduced functions covering dependency of EFPs. The functions express the influence of the provided properties by the properties on the required side. Together with these functions they provided a meta-model called QoSCL that allows to specify: the intrinsic qualities of a service, the required or provided quality levels of a service, the extra-functional dependency of a provided services quality on a set of qualities defined on required services.

The QoSCL allows to express three kinds of relations: numerical constrains, mathematical functions, or empirical rules. From our point of view, we see the main benefit in mathematical functions. These functions determine the resulting provided property depending on required properties, for instance: $memory\_consumption_{prov} = \sum_{i=O}^{N} memory\_consumption_{req_i} + 10$. A considerable weakness of such solution is that each function has to be defined for each component. We would like to add template or generic predefined functions that would users take and fill in the body relevant to each component.

3.5.3 Treatment of EFPs from Design to Run-time

The work [5] overviews EFPs from the earliest phase when components are being developed to run-time when EFPs of the components are evaluated. At design-time they first proposed to use CQML+ to define EFPs. When the defined EFPs will be evaluated at run-time, they then introduced the transformation of CQML+ notation into the XML notation. Furthermore the XML notation is used by a container to evaluate EFPs. The work [5], however, does not detail this evaluation and provides only an example with response time instead. Unfortunately the provided example does not show how it could be generalised to a general evaluation mechanism.

Chapter 4

Concept of the Thesis

The overall goal of this work is to propose an improvement of component-based programming. It targets extra-functional properties and the goal is to contribute to current research in the field of improved definitions and evaluations of extra-functional properties especially for components.

The survey of the state-of-the-art has shown that a lot of approaches defined how an extra-functional property should look like. A structure of an EFP has been typically presented by means of specialised languages such as QML, CQML, HQML. Other approaches instead addressed the relevance and the evaluation of EFPs in terms of the whole component application. We may name component models such as Palladio, Robocop, ProCom, etc. that take EFPs into account as part of the model.

In our work, the rationale behind taking EFPs into account is to improve compatibly checks of components. On one hand the component system behaviour may be evaluated upon comprehensive models that is the case in Palladio. On the other hand component verifications may rely only on a type-based conformance [9, 14]. The type-based conformance covers conformance of interfaces.

Main obstacle of the evaluation of the behaviour based on models is the computational complexity – the state space explosion problem. In addition, we see the need of creation of a lot of parametrised models too difficult for common users. In opposite, the type-based evaluation has relatively low resource needs [9]. We also think that it is more user friendly to work only with information provided by component interfaces. In addition, the mechanism of providing and evaluating only interfaces supports black-box nature of components that we want to follow.

On the other hand a problem of languages we have studied is that they often treat EFPs independently of each other. Despite of it, EFPs are typically influenced by EFPs connected through other components.

The mentioned reasons lead us to follow the type-based approach. The type-

based approach matches two components as compatible ones if interfaces are of the same types or sub-types. In our view, the compatibility of interfaces includes (i) the compatibility of method signatures: service names, input and output parameters and (ii) the compatibility of extra-functional properties attached to the interfaces. The compatibility of interfaces in terms of method signatures has been addressed in [9] or [14]. The goal of this work is to supply methods for the EFPs on interfaces which has so far be neglected. In addition, we aim at providing a mechanism of composing EFPs in chains of components. The mechanism should be capable of expressing the influence of EFPs each other at a sufficient level and consume lower resources in a comparison to behaviour based models.

The approach leads to a proposed algorithm comparing a compatibility of component interfaces is useful in two situations. Firstly, it helps to test component assemblies when the system developer composes components in a test environment.. Secondly, it may guard a component assembly when the system runs – typically when the system is started or components are changed at runtime.

4.1 Extra-functional Properties

We aim at enriching interfaces by extra-functional properties and for that reason we have analysed existing extra-functional languages. The research of existing languages has shown a rich base of notations allowing to attach EFPs to interfaces. It has consequently shown that there is no reason to prepare yet another language. As a result we have created a structure of our EFPs combining existing languages.

We have first identified that an extra-functional property should consist of: a name and a data type. We have second added a comparing function and a block of additional meta-information. The name and the date types of EFPs have been inspired by existing languages (mainly CQML). We furthermore distinguish between simple and derived properties used also in NoFun. We have finally defined a special king of EFPs – called deployment contract. They express properties that a component has upon the environment the component run in.

More formally, we define:

$$e_{def}^{simple} = (n, \gamma, t, META) \tag{4.1}$$

$$e_{def}^{derived} = (n, E, \gamma, t, META) \tag{4.2}$$

$$e^{deployment\_contract} \equiv e^{simple} \vee e^{deployment\_contract} \equiv e^{derived} \tag{4.3}$$

where the meaning of the formula is:

n is the name of a property

$t \in T = T_c \cup T_s$ is the type of a property

T_s is a set of simple types. $T_s = \{real, integer, boolean, enum, set, ratio, string\}$

$T_c = \{(t_1, \cdots, t_N) | N > 1, t \in T\}$ is a set of complex types containing a non primitive value. It aggregates other (simple or complex) types. The essence is similar to *struct* in the C language or *record* in Pascal

$\gamma : x \times y \to z; z \in \{-1, 0, 1, \text{“}n/d\text{”}\}$ is a function which compares two instances x, y of the property type t, stating which of the two values is better. We work with several predefined *gamma* functions such as Increasing (more is better), Decreasing (less is better), and assume the possibility to define new ones. The meaning of the return values is:

Value	Meaning
-1	x is worse than y
0	x is equal to y
+1	x is better than y
“n/d”	not-defined.

The function may not be explicitly defined and then the following implicit rules hold: (i) *real, integer, ratio* use mappings -1: $x < y$, 0: $x = y$, +1: $x > y$, (ii) *string* uses mappings 0: x literally equal to y else “n/d”, (iii) *boolean* uses mappings 0: $x = y$ else “n/d”, (iv) *set, enum and complex* use previous rules for each element and the result is “n/d” unless each evaluation holds the same value. When an explicit rule does not exist and comparison can not be determined by the implicit rule, the value “n/d” is resulted.

$E = \{e_1, \cdots, e_N\}$ are properties composing a derived property

$META$ is a record containing any additional information refered by the domain. Its elements are described by an extensible model which currently contains the items $unit, names$, where

unit : String – is a measuring unit of the property

names is an ordered enumeration containing every name for the values of this property allowed to be used in local registries

Note that all META values are optional and to be used only when they are needed and meaningful in the domain.

4.2 Context of Usage

The approaches we have studied, so far, rarely address context dependency of EFPs. It is obvious that components are deployed in different contexts of usage. Although the function of components must stay unchanged in all contexts, EFPs often vary in different contexts. The nature of components disallow to attach properties with exact numbers to the interfaces. Whereas a component will run in different environments, direct numbers mostly have no sense. For instance, memory consumption of 20MB would be considered small on a desktop computer while it would be considered high on a portable electronic device. Other properties such as performance will probably change in different environments[1]. For that reason we propose to introduce abstract names that encapsulate real numbers valid in contexts.

Furthermore, users often do not rely on exact values of EFPs. For instance, one does not have to know how much memory the component needs, he only needs to know an interval in which the memory consumption is limited. For instance, it is difficult to state that a component will always need 20MB of memory. Moreover such information is often needlessly exact. Users may want to know that the memory consumption will not exceed the interval from 10MB to 20MB. For that reason, we suggest to divide continuous intervals of values into disjunctive sets expressing typical scales of values.

Finally, when the mechanism comparing EFPs is desired, all vendors of the components must use unified properties. The overall idea is that different vendors provide components enriched by EFPs. Once a developer obtain a set of components he or she needs to compare and bind them together with respects to their EFPs compatibility. The properties may be comparable only when they have the same meaning and the vendors use the same properties to express the same aspects of extra-functionality. For that reason we suggest a common repository of EFPs from which the vendors take unified EFPs.

4.3 Registry

The mentioned reasons lead us to propose common repositories of EFPs that:

1. store unified properties – the vendors then use comparable properties;
2. encapsulate context-dependent values for each context;
3. divide continuous values of intervals into disjunctive sets.

[1] There are, however, some exceptions concerning values that do not change among contexts. It includes e.g. physical values such the gravity constant

The common repository we name Registry and its core idea is shown in Figure 4.1. The picture demonstrates two dimensions: one concerns **context** dependencies

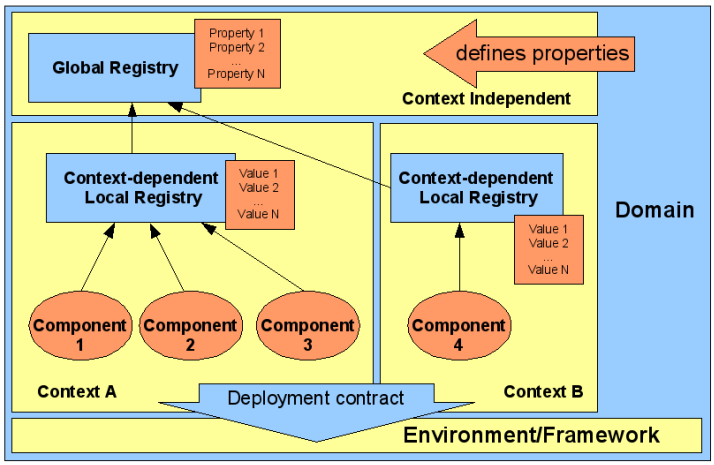

Figure 4.1: A relation of registries and components in contexts and a domain

and the other one concerns a **domain** of usage. By a context we mean a different computational environment (e.g. context of mobile phones, desktop machines, servers) and a domain is an area which a final system is developed for (e.g. systems for libraries, hospitals, schools, automotive industry).

All registries, components and computational environment are bordered by a domain (an area of usage). Each context is also bound to the domain.

Global registry (GR) is a store with definitions of EFP from Section 4.1. GR only defines the properties themselves but does not contain their values. The validity of GR is for all contexts specified by the domain. It contains the definition of all EFPs meaningful for a domain. The main reason for having GR is to provide all vendors with the same meaning of the properties.

Local registry (LR) is concerned with a contextual meaning of EFPs. Each context has one local registry which stores values valid for the context. The values are linked to the definitions provided by the GR. Another usage of LR is that it splits continuous values of properties to several disjunctive sets that are easier to manage.

The mechanism used by LR creates symbolic names for values resulting in named intervals. The EFPs themselves are usable in other contexts with different values, because the value names can remain the same while the underlying values are changed.

Note: The deployment contract shows a dependency of components on execution environment or framework (e.g. a resource as a file in operating system, access

to hardware, an execution of other processes/binaries). The system of registries does not distinguish between extra-functional and deployment contract properties. They are defined equivalently and they are distinguished when they are used on components.

4.3.1 Formalisation

In this section the formal model of regitries is presented.

Global registry is a simple list of definitions of properties.

$$GR = (loc, \{e_i\}) \tag{4.4}$$

where:

loc is the registry's URI location which associates it with the domain

$\{e_i\}$ is a set of (simple or derived) extra-functional properties

The loc value is defined implicitly by the registry deployment location and does not have to be provided explicitly.

Assume there exists a global registry GR. Then **local registry** for a context contains records defining values valid in the context. In effect, this assigns a semantics to the properties.

$$LR = (loc, loc_{parent}, loc_{gr}, S, D) \tag{4.5}$$

where:

loc is the URI of registry associating it with the context

loc_{parent} is the URI of parent local registry. It allows to create a tree hierarchy of local registries. The meaning is that registry inherits values from parents. Similarly to object-oriented approach, the inherited values may be overridden and modified or new items may be added.

loc_{gr} is a link to the global registry

$S = \{s_i\}$ is a set defining context dependent values for simple properties

$s_i = (name, value\_name, range)$ is a tuple of a property name, a value name and the value's range

$name$: String is a name of a property from GR

value_name : String is an assigned name of the value which must be selected from the list of available names given in the $META :: names$ part of the definition of the property in GR

range is an interval, a set or a value $\in T$ which defines a restriction on available values

$D = \{d_i : \{r_{i1}, \cdots, r_{iK}\}\}$ is a set of derived property definitions, where each derived property d_i is governed by r_{ij} rules

$d_i = (name)$; a derived property name from GR

$r_{ij} : F \Rightarrow x; x = value\_name$ or $x = value \in T^{enum}$ is a resulting name or an enum value which is valid when the logical expression F is evaluated to *true*

The local registry contains both the assignment of values to the names (the set S) and the rules expressing the derivation of derived properties (the set D).

The elements of the set S simply associate values to simple properties through names. The elements of the set D associate also names but do so using logical rules which express the definition of a derived property.

4.4 Property Comparison

When two components are compared whether they are compatible – whether one may replace the other one – the EFPs attached to interfaces are compared.

Two components C_1 and C_2 can be marked as compatible when (i) extra-functional properties on the provided side guarantee at least the same level of quality, (ii) extra-functional or deployment contract properties on the required side declare the need for the same or a lower level of quality, (iii) properties with the same names match. An algorithm for comparing components (usually two versions of the same component or a component in different environments) works in two steps. Firstly, it binds provided and required properties comparing their names and then it checks whether no property is missing on the provided side and no property has been added on the required side.

Secondly, a matching function is applied on all equivalent properties. Using a sequence $E_C(C_1, C_2) = ((x_i, y_i)_k), x_i \in efp(C_1), y_i \in efp(C_2)$ of equally named properties from the two components, the function $m : C \times C \rightarrow (z_k)$, $z_k \in \{-1, 0, 1, n/d\}$ matches the components by evaluating the function $\gamma(x, y)$ which is defined by formulas 4.1 and 4.2 above.

The mechanism of the comparison function is:

$$m(C_1, C_2) : z_k = \gamma_k(E_C(C_1, C_2)_k) \tag{4.6}$$

The algorithm is identical for both provided and required (deployment contract or extra-functional) properties. The two components are compatible only if each $z_k^{prov} \in \{0, 1\}$ for provided properties and each $z_k^{req} \in \{-1, 0\}$ for required ones.

The same function $m(C_1, C_2)$ also matches a provided side of one component to a required side of another component and vice versa when interoperating components are to be bound.

The DCs comparison requires the runtime environment to be enriched by properties on which the comparison is performed. Attaching of properties to environment works essentially the same way as attaching to components. The environment provides some properties which are compared on components required deployment contracts.

This mechanism may be repeatedly evaluated on all interfaces of all components and it evaluates whether a chain of components contains only components with compatible interfaces (in terms of extra-functional properties).

4.5 Case Study

This section first shows how the presented mechanism may be used in practise. This section then evaluates the mechanism and discuses current drawbacks that will be addressed in the future.

4.5.1 Example

At this point, we would like to demonstrate the usage of the approach on an example. Let us assume we have a component system from Figure 4.2. The com-

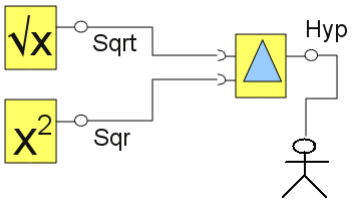

Figure 4.2: A triangle component in an assembly

ponent we will be investigating has a symbol with a triangle – the component is under development and is called the *triangle* component. The component is used for computing some operations with triangles. Let us assume that the interface labelled as *Hyp* provides a computation for the well-known triangular formula:

$c = \sqrt{a^2 + b^2}$. A developer may want to attach extra-functional properties to the component.

The developer uses properties defined in registry. For instance, there is the definition of the property *response_time* in global registry:

```
(GR)
response_time : decreasing integer {unit:‘‘ms’’, names: {low, average, high}}
```

The example shows a pseudo-code corresponding to the presented formalisation: the extra-functional property is defined by the name, the data type, the comparing function (decreasing) and the measuring unit in milliseconds. The continuous interval of values is divided into disjunctive sets low, average, high.

The distribution of the (low, average, high) intervals, which is valid for a concrete context of usage, is expressed in local registry (also a pseudo-code):

```
(LR)
# link to GR
URI: http://services.kiv.zcu.cz/triangle/extrafunc/v1/

response_time : high = (500; +INFINITY)
response_time : average = (100; 500]
response_time : low = (0; 100]
```

Obviously, the component computing the formula $c = \sqrt{a^2 + b^2}$ must call other interfaces. The interface *Sqr* must be called twice to evaluate "x^2"and the interface *Sqrt* once to evaluate "$\sqrt{x}$". Let us assume that the operation "+" is performed inside the *triangle* component. The developer of the component may evaluate that the *response_time* on the interface *Hyp* will be:

$$response\_time^{Hyp} = 2 \cdot response\_time^{Sqr} + response\_time^{Sqrt} \tag{4.7}$$

Response time on the interface *Hyp* may be also slowed by an inner computation of the *triangle* component and the equation may look:

$$response\_time^{Hyp} = 1.1 \cdot (2 \cdot response\_time^{Sqr} + response\_time^{Sqrt}) \tag{4.8}$$

Where the constant 1.1 expresses the delay of the inner computation.

In any case, the developer of the component uses internally equations 4.7 or 4.8 to estimate extra-functional demands and offers of the component. The developer may e.g. compute the equation 4.7 resulting to: $2 \cdot low + low \leq average$ ($average$ is for worse-case). It then results in the decision: *if the component assumes "low" response_time on both required interfaces Sqr and Sqrt it may then guarantee "average" (worse-case) response time on the provided interface Hyp.*

The result estimated by the components developer may be then attached to the component descriptor:

```
# link to registry
ExtraFunc-Catalog: URI: http://services.kiv.zcu.cz/triangle/extrafunc/v1/

Provided-Services : cz.zcu.kiv.services.Hyp;
  extrafunc=(response_time = average)

Required-Services : cz.zcu.kiv.services.Sqr; extrafunc=(response_time = low)
Required-Services : cz.zcu.kiv.services.Sqrt; extrafunc=(response_time = low)
```

This descriptor is distributed together with the component. Once the component is to be used in a system, the comparing mechanism from Section 4.4 evaluates whether the component may be used.

In this example, the distribution of intervals is *low*, *average*, *high* with decreasing ordering. It means that *low* is better then *average* and *average* is better then *high*. For that reason a matching mechanism would accept to bind the *triangle* component only with different *Sqr* or *Sqrt* interfaces that provide also *low* response_time. Other intervals are worse and could not be accepted.

A different configuration that guarantee *high* output response_time for *average* input response_time could look like:

```
Provided-Services : cz.zcu.kiv.services.Hyp;
  extrafunc=(response_time = high)

Required-Services : cz.zcu.kiv.services.Sqr; extrafunc=(response_time = avarage)
Required-Services : cz.zcu.kiv.services.Sqrt; extrafunc=(response_time = avarage)
```

For this configuration, the matching algorithm would allow the *triangle* component to be bound with *Sqr* and *Sqrt* interfaces providing *low* or *average* response_time.

4.5.2 Evaluation

Section 4.5.1 has shown a practical usage of the mechanism we have developed, though the mechanism may be inflexible in some situations. The main drawback is that the mechanism treats components in isolation. It means that the developer of the component may state the extra-functional condition in which the component will guarantee extra-functional properties on the result. However a developer of a final system often needs to know extra-functional properties of the whole system rather than relying on each component. For instance, the comparing mechanism could reject a component because of an insufficient level of EFPs, however, the component would work in the system and the EFPs level of the whole system would be reached.

Let us assume a modified example from the previous section shown in Figure 4.3. The system contains a new mathematical component called *Math*. The component has an interface *Triang* that contains mathematical operations for

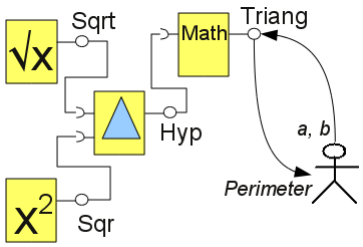

Figure 4.3: A math component in an assembly

triangles. The *Triang* interface is the only point where the user accesses the system. A user of the system first calls the interface giving the edges a and b of a right triangle in a time t_1. The user then receives a perimeter of the triangle in a time t_2. Obviously, the missing edge c of the right triangle is evaluated from the connected *triangle* component.

Evidently, the response time of the system is:

$$response\_time^{triang} = t_2 - t_1 \tag{4.9}$$

While the *Math* component is connected with the *triangle* component, the response time of the system is:

$$response\_time^{triang} \approx response\_time^{Hyp}. \tag{4.10}$$

Let us assume we have registries and component descriptors from Section 4.5.1. Let us also assume we have only components implementing interfaces *Sqr* and *Sqrt* with response time equal to *high*:

```
Provided-Services : cz.zcu.kiv.services.Sqr; extrafunc=(response_time = high)
Provided-Services : cz.zcu.kiv.services.Sqrt; extrafunc=(response_time = high)
```

Whereas the computation of the *Math* component is not time-critical, the developer of the final system may decide that the response time may be *high*.

If we used only a mechanism from Section 4.5.1, it would be impossible to compose the system. The *triangle* component could not be bound to available *Sqr* and *Sqrt* interfaces, because they provide the *high* response time while the *triangle* component require the *low* or *average* response time.

We may use the equation 4.7 from Section 4.5.1 and put the *high* interval as a parameter:

$$response\_time^{Hyp} = 2 \cdot high + high \leq high \tag{4.11}$$

The result of the equation informs that the usage of *Sqr* and *Sqrt* interfaces with *high* response time results in the *high* response time on the *Hyp* interface. This

result may be put to the equation 4.10 and the results is:

$$response\_time^{triang} \approx response\_time^{Hyp} \leq high$$
$$\Rightarrow response\_time^{triang} \leq high \tag{4.12}$$

The last equation 4.12 proves that the whole system will work and reach the user expectations while the preliminary approach from Section 4.5.1 would reject to compose the system.

The example leads us to following conclusions:

(i) the components in a component assembly may be divided into three group

1. Source components – have only provided interfaces. They provide EFPs that do not depend on other EFPs.
2. Interconnected components – have both required and provided interfaces. They use other components to process a computation and returns a result. EFPs of these components should be propagated from the required to the provided side of the components.
3. Sink components – have provided interfaces that are accessed by terminal clients. EFPs should be also propagated from the required to the provided side. The results on the provided side inform about the EFPs level of the whole assembly.

(ii) the evaluation mechanism should take into account the following:

1. Source component EFPs may have directly attached values or intervals from registries
2. Sink components have EFPs on provided side that express users expectations. Their result values are important for the developer of the assemblies in evaluating the level of EFPs of the whole system.
3. Interconnected components should compute EFPs by mathematical formulas instead of relying on direct values or intervals.

4.6 Future Work

According to finished work mentioned in Section 4 and the evaluation in Section 4.5.2 our future work aims at preparing a mechanism that:

1. Uses named intervals to attach approximated values to the source components.

2. Allows to define mathematical formulas to compute the transformation of EFPs from the required side of components to the provided side of components.

3. Allows users to check the level of EFPs of the system on the sink components

The mathematical formulas are inspired in works [15] and [40]. However developers of components may be in difficulty when considering how such functions should look like and which EFPs impact other EFPs. For that reason, we would like to extend registry to store templates or pre-defined headers for these functions. The developers would first obtain these templates or headers from the repository. They would then fill the body of the function to correspond to a concrete implementation of the component.

The usage of the intervals and the mathematical formulas will lead to approximated results of EFPs values. Behaviour based models such as Palladio allows probably more accurate results, though our solution aims at less resource consuming mechanism that is also easy-to-use by common developers. Still, the values that are behind approximated intervals serve as useful hints to the developers to assemble trustworthy component systems.

The completed mechanism will be verified by an implementation. We would like to prepare a toolbox consisting of three tools shown in Figure 4.4 These tools cover three phases of component development: (i) First is the *Registry Provider* tool implementing registry. The registry is filled by a domain expert and allows other tools to access EFPs. (ii) Second is the *EFP Attachment* tool that loads data from registry and allows to attach EFPs and the mathematical functions to components. This tool is used by the developer of components that attach EFPs from registry to components. (iii) Third is the *EFP Evaluator* tool that is used by the system developer that composes components into an assembly and let the tool to evaluate resulting EFPs.

We are currently working on the first tool – the repository implementation. Second two tools will use the registry provider tool to read EFPs from registry.

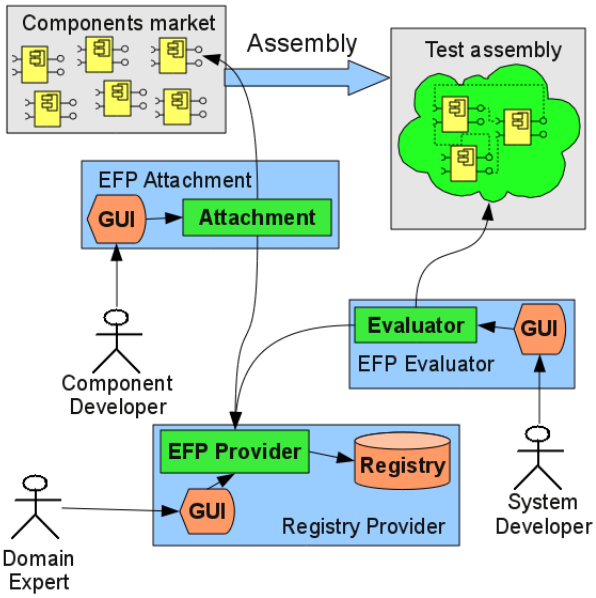

Figure 4.4: The toolbox

Chapter 5

Conclusion

This report has overviewed fundamentals of component based software engineering. It has consequently highlighted extra-functional properties as an important aspect to deal with. The main part of this work has been analyses of the state of the art related to extra-functional properties and components.

We have explained our current work that explicitly targets context dependency of extra-functional properties on components. We have developed a system of registries that serves as a common repository of extra-functional properties.

The introduced case study evaluates our mechanism and shows our future work. The improvements suggested will deal with a composition of extra-functional properties when components are connected in chains in which the properties are influenced by each other. In addition, we would like to implement the approach as a toolbox.

Bibliography

[1] IEEE 610.3-1989 standard glossary of modeling and simulation terminology, 1989.

[2] International standard ISO/IEC 9126. Information technology - Software product evaluation - Quality characteristics and guidelines for their use, 1991.

[3] IEEE 830-1998 recommended practice for software requirements specifications, 1998.

[4] J. Ø. Aagedal. *Quality of Service Support in Development of Distributed Systems*. PhD thesis, University of Oslo, 2001.

[5] R. Aigner, M. Pohlack, S. Rttger, and S. Zschaler. Towards pervasive treatment of non-functional properties at design and run-time. 2003.

[6] A. Alvaro, E. S. de Almeida, and S. L. Meira. A software component quality model: A preliminary evaluation. In *EUROMICRO '06: Proceedings of the 32nd EUROMICRO Conference on Software Engineering and Advanced Applications*, pages 28–37, Washington, DC, USA, 2006. IEEE Computer Society.

[7] R. Alvaro, E. S. D. Almeida, S. Romero, and L. Meira. Quality attributes for a component quality model. In *In the 10th International Workshop on Component-Oriented Programming (WCOP) in Conjunction with the 19th European Conference on Object Oriented Programming (ECOOP*, 2005.

[8] F. Bachman, L. Bass, C. Buhman, S. Comella-Dorda, F. Long, J. Rober, R. Seacord, and K. Wallnau. Volume ii: Technical concepts of component-based software engineering, 2nd edition. Technical report, SEI Joint Program Office, 2000.

[9] J. Bauml and P. Brada. Automated versioning in osgi: A mechanism for component software consistency guarantee. In *EUROMICRO-SEAA*, pages 428–435, 2009.

[10] S. Becker, H. Koziolek, and R. Reussner. The palladio component model for model-driven performance prediction. *Journal of Systems and Software*, 82(1):3 – 22, 2009. Special Issue: Software Performance - Modeling and Analysis.

[11] E. Bondarev, M. R. Chaudron, and P. H. de With. Compositional performance analysis of component-based systems on heterogeneous multiprocessor platforms. In *Proceedings of Euromicro conference on Software Engineering and Advanced Applications*, pages 81–91. IEEE Computer Society, 2006.

[12] P. Botella, X. Burgues, X. Franch, M. Huerta, and G. Salazaruml. Modeling non-functional requirements. In *Proceedings of Jornadas de Ingenieria de Requisitos Aplicada JIRA 2001*, 2001.

[13] P. Brada. The CoSi component model: Reviving the black-box nature of components. In *Proceedings of the 11th International Symposium on Component Based Software Engineering*, number 5282 in LNCS, Karlsruhe, Germany, October 2008. Springer Verlag.

[14] P. Brada and L. Valenta. Practical verification of component substitutability using subtype relation. In *Proceedings of the 32nd Euromicro SEAA conference*, pages 38–45. IEEE Computer Society, 2006.

[15] O. Defour, J. marc Jzquel, and N. Plouzeau. Extra-functional contract support in components. In *In Proceedings of 7th International Symposium on Component-Based Software Engineering (CBSE 7*, 2004.

[16] EJB. *Enterprise JavaBeans, Version 3.0. EJB Core Contracts and Requirements.* Sun Microsystems, May 2006. JSR220 Final Release.

[17] X. Franch. Systematic formulation of non-functional characteristics of software. In *Proceedings of International Conference on Requirements Engineering (ICRE)*, pages 174–181, 1998.

[18] S. Frlund, S. F. Lund, J. Koistinen, and J. Koistinen. Quality of service specification in distributed object systems design, 1998.

[19] J. M. García, D. Ruiz, A. Ruiz-Cortés, O. Martín-Díaz, and M. Resinas. An hybrid, qos-aware discovery of semantic web services using constraint programming. In *ICSOC '07: Proceedings of the 5th international conference on Service-Oriented Computing*, pages 69–80, Berlin, Heidelberg, 2007. Springer-Verlag.

[20] J. M. Garca, I. Toma, D. Ruiz, and A. Ruiz-corts. A service ranker based on logic rules evaluation and constraint programming.

[21] I. Hussain, L. Kosseim, and O. Ormandjieva. Using linguistic knowledge to classify non-functional requirements in srs documents. In *NLDB '08: Proceedings of the 13th international conference on Natural Language and Information Systems*, pages 287–298, Berlin, Heidelberg, 2008. Springer-Verlag.

[22] K. Jezek, P. Brada, and P. Stepan. Towards context independent extra-functional properties descriptor for components. In *Proceedings of the 7th International Workshop on Formal Engineering approches to Software Components and Architectures (FESCA 2010)*, 2010.

[23] K. kiu Lau and V. Ukis. Defining and checking deployment contracts for software components. *In Proceedings of the 9th International Symposium on Component-Based Software Engineering, volume 4063 of LNCS*, pages 1–16, 2006.

[24] D. D. Lamanna, J. Skene, and W. Emmerich. Slang: A language for defining service level agreements. *Future Trends of Distributed Computing Systems, IEEE International Workshop*, 0:100, 2003.

[25] K.-K. Lau and V. Ukis. Deployment contracts for software components. (CSPP-36), February 2006.

[26] M. Mohammad and V. S. Alagar. TADL - an architecture description language for trustworthy component-based systems. In *ECSA '08: Proceedings of the 2nd European conference on Software Architecture*, pages 290–297. Springer, 2008.

[27] M. Mulugeta and A. Schill. A framework for qos contract negotiation in component-based applications. pages 238–251, 2008.

[28] J. Muskens, M. R. Chaudron, and J. J. Lukkien. *Component-Based Software Development for Embedded Systems*, chapter A Component Framework for Consumer Electronics Middleware, pages 164–184. Springer Verlag, 2005.

[29] G. Nahrstedt and Y. Wichadakul. An xmlbased quality of service enabling language for the web, 2001.

[30] OMG. UML profile for modeling quality of service and fault tolerance characteristics and mechanism specification. Technical report, OMG - Object Management Group, 2008.

[31] OMG. *UML Profile for MARTE: Modeling and Analysis of Real-Time Embedded Systems*. OMG, 2009. available at: http://www.omg.org/spec/MARTE/1.0/PDF (2010).

[32] OMG. UML unified modeling language. techreport, n.d. ver 2.

[33] OSGi. *OSGi*. OSGi Aliance, n.d. Available at http://www.osgi.org/.

[34] S. Röttger and S. Zschaler. CQML+: Enhancements to CQML. In J.-M. Bruel, editor, *Proc. 1st Int'l Workshop on Quality of Service in Component-Based Software Engineering, Toulouse, France*, pages 43–56. Cépaduès-Éditions, jun 2003.

[35] S. Sentilles, P. Stepan, J. Carlson, and I. Crnkovic. Integration of extra-functional properties in component models. *12th International Symposium on Component Based Software Engineering (CBSE 2009), LNCS 5582*, June 2009.

[36] Spring. *Spring Framework*. Spring Comunity, ver. 3 edition, n.d. Available at http://www.springsource.org/.

[37] Sun Microsystems. *Enterprise JavaBeans, Version 3.0. EJB Core Contracts and Requirements*, May 2006. JSR220 Final Release.

[38] C. Szyperski, (with Dominik Gruntz, and S. Murer). *Component Software - Beyond Object-Oriented Programming: Second Edition*. Addison-Wesley / ACM Press, 2002.

[39] X. Wang, T. Vitvar, M. Kerrigan, and I. Toma. A qos-aware selection model for semantic web services. *Service-Oriented Computing ICSOC 2006*, pages 390–401, 2006.

[40] S. Zschaler and M. Meyerhfer. Explicit modelling of qos-dependencies. pages 57–66, 2003.