University of West Bohemia

Faculty of Applied Sciences

Extra-Functional Properties Support For a Variety of Component Models

Ing. Kamil Ježek

Doctoral Thesis
submitted in partial fulfilment of the requirements
for the degree of Doctor of Philosophy
in the specialization Computer Science and
Engineering

Supervisor: Ing. Přemysl Brada MSc., Ph.D.
Department of Computer Science and Engineering

Pilsen, 2012

Západočeská univerzita v Plzni

Fakulta aplikovaných věd

Podpora mimofunkčních charakteristik v komponentových modelech

Ing. Kamil Ježek

Disertační práce
k získání akademického titulu doktor
v oboru Informatika a výpočetní technika

Školitel: Ing. Přemysl Brada MSc., Ph.D.
Katedra informatiky a výpočetní techniky

V Plzni, 2012

Prohlášení

Předkládám tímto k posouzení a obhajobě disertační práci zpracovanou na závěr doktorského studia na Fakultě aplikovaných věd Západočeské univerzity v Plzni.

Prohlašuji tímto, že tuto práci jsem vypracoval samostatně, s použitím odborné literatury a dostupných pramenů uvedených v seznamu, jenž je součástí této práce.

V Plzni dne 30.3 2012 Ing. Kamil Ježek

Abstract

Approaches that target software composition are becoming remarkably important with the gradual enlargement of software systems. Together with the adoption of component-based programming to cope with software complexity, extra-functional properties are playing a more important role. This work deals with the problem of insufficient adoption of extra-functional properties among a variety of component models. It builds on the assumption that such insufficient adoption consequently limits the adoption of component-based programming itself. It is particularly noticeable in industrial applications. As a suggested solution, this work proposes a comprehensive mechanism enabling the use of extra-functional properties in existing systems. Thanks to this mechanism, extra-functional properties may be independently applied into the systems that have not contained the properties before. It should lead, among things, to the wider use of component based programming. The mechanism is based on other state-of-the-art approaches. The presented thesis provides their analysis, formally defines the mechanism, and describes its implementation in Java-based technologies. Main building blocks of the mechanism are a layered properties repository, a model assigning the properties to a variety of systems, and an evaluation algorithm. Application of the mechanism to industrial component models, namely Spring and OSGi, as well as a case-study presenting one of the practical applications is also part of this work.

Abstrakt

Metody zlepšující modulární tvorbu software se stávají stále více důležité, tak jak se stále zvětšuje software. Společně s využitím komponentově orientovaného programování, jako prostředek řešící komplexnost software, mimofunkční charakteristiky hrají stále důležitější roli. Problém adresovaný v této práci zahrnuje nedostatečné použití mimofunkčních charakteristik v existujících systémech. Tato práce staví na předpokladu, že toto nedostatečné použití zároveň omezuje využití komponentově orientovaného programování jako takového. Jako možné řešení, tato práce představuje robustní mechanismus, který umožňuje zavést mimofunkční charakteristiky do již existujících systémů. Díky tomuto přístupu, mimofunkční charakteristiky mohou být mnohem rychleji zavedeny v praxi, což vede také k větší rozšířenosti komponentového programování. Uvedený systém je založen na existujících řešeních, formálně definován a naprogramován v Javě. Základní stavební bloky tohoto systému jsou univerzální úložiště charakteristik, mechanismus umožňující přiřadit tyto charakteristky k různým komponentovým systémům a algorimus pro vyhodnocení charakteristik. Uvedené řešení je implementováno do průmyslových systémů Spring a OSGi. Případová studie ukazující možnou aplikaci tohoto systému je také součást této práce.

Contents

Chapter 1

Introduction

Current software products increase in their size. Together with the effort to let computers manage more and more complex information, the software expands to very complicated systems.

Despite that a lot of current software is still created from scratch. It is obvious different applications use the same parts of logic or work-flow. For that reason, partial solutions have been invented to avoid repetition of the same code. One of the solutions are software libraries in a form of source or binary code. Developers use the libraries, though the core functionality must be still hand coded. The amount of libraries and they relations causes that the developers must learn overwhelming amount of information.

On the other hand, software vendors want to decrease the time-to-market as well as the price of the software. It contrasts with the amount of information people involved in the software development must acquire and implement. Therefore, a fast and reliable software development process should solve (i) the inefficiency of the current development process and (ii) decrease the amount of information developers must learn.

As a result, more sophisticated solutions have been invented. Two of the approaches aiming at solving this problem are component based programming and service oriented architecture. A common idea behind both of them is the composition of final functionality from components or services without a need of a "glue" code. The underlying benefit is that components or services may first be prepared by developers who are experts in a particular domain. Then a system architect needs to learn only an overall structure to compose the system. It consequently distributes informations among different persons involved in the software process.

Since no additional code is written and software parts are re-used, the development time and thus the price are decreased. In addition, the re-used parts are prepared by particular experts and they are used and verified among several applications. This process increases their stability and reliability. Despite all these benefits, other issues keep to arise.

1.1 Problem Definition

If any technology composing software from pre-existing parts, either components or services, is adopted, compatibility verification is more important than before. The use of pre-existing software parts requires a strong quality assurance. A bug in a software part would be rapidly distributed to a lot of applications as this software parts is distributed. Additional verification of the composition itself is nonetheless important. The reason is that even a bug free component may not be working in cooperation with other components. As a result, precisely verified isolated components (services) as well as the verification of their composition can guarantee a fully working system.

The functionality of a software is often divided into functional and extra-functional (also referred to as non-functional) characteristics. While the functional characteristics denote fundamental purposes a component or service has been developed for, the extra-functional characteristics denote qualitative aspects of how the components or services do their functionality.

Current research invests considerable effort to describe, implement and use both functional and extra-functional characteristics leading to strong compatibility verification. Since the functional characteristics are not a new concept in software development, they are well supported in current industrial systems. For instance, the provided and required side binding is used in OSGi while WSDL describes web-services in the service oriented architecture. On the other hand, extra-functional characteristics are still a topic undergoing current research.

A lot of research has been done to address extra-functional properties, however, there is still a limited application to industrial component models. As long as developers have no routine support of the extra-functional properties, weak compatibility checks cause slow adoption of component base programming. It leads to connected issues in which developers do not adopt this innovative technique because of its problems while the technique itself cannot improve because of its slow adoption. As a result, this problem limits third-party sharing of either components or services with the consequence that they are still not a mainstream in software development.

Therefore, the main problem addressed in this work is the slow application of extra-functional properties into routinely used industrial component models. Although the industrial component models are widely used, the components exchange across organisations is still rare. This work uses the assumption that this problem is caused by weak compatibility checks caused literally by the weak extra-functional properties support in respective industrial component models.

1.2 Goal of the Work

There have been several attempts at providing extra-functional property support for software systems. They start from describing extra-functional properties [26, 50] through their applicability to other systems and end in the development of complex systems embedding as their part either extra-functional properties [76, 15] or quality of service specifications [101, 38].

A considerable number of approaches to extra-functional properties exist. Although these approaches have already shown directions leading to the successful implementation of extra-functional properties, only a little work has been done to their industrial application. As far as we know, industrial component frameworks with no extra-functional property support such as Spring or OSGi are widely used while industrial frameworks with extra-functional property support are rare. A question remaining open is why, despite their high number, research works in this area are slowly acquired by the industry.

In this work, we try to propose a new approach to this problem. Rather than creating a new complex model which natively supports extra-functional properties, the first goal is to propose an independent mechanism which extends existing systems with extra-functional properties. The rationale is that even a basic extra-functional property support is beneficial for industrial component frameworks. In addition, these frameworks as well as the properties may evolve independently.

A second goal is to base this new mechanism on the existing state-of-the-art and thus it should not be a breakthrough in terms of extra-functional property definition, application and evaluation. The novelty should lay in the innovative applicability to a variety of existing systems instead.

Since the mechanism should cover a wide range of existing systems, a unified sharing and an independent usage of extra-functional properties is a consequent goal.

Finally, a last goal is to implement the output of this work as a toolbox providing an instant applicability of the output to practise. It supports the practical usage and verifies the approach.

The following parts are organised as follows: Chapter 2 overviews fundamentals of component-based software engineering followed by Chapter 3 bringing the survey of the state-of-the-art related to extra-functional properties. The description of the main contribution of this work is given in Chapter 4 while its implementation is described in Chapter 5. A practical application of the approach is demonstrated on a case-study in Chapter 6 followed by a discussion and future work in Chapter 7.

Chapter 2

Background: Component Architectures

2.1 Motivation

Current research and industry adopt Component Based Software Engineering (CBSE) as a promising approach to software development with several benefits that will be summarised in this section.

The overall idea of CBSE has been inspired in other industrial areas. For instance, different houses are build from unified concrete blocks, different cars use unified parts of engines, different types of electronics embed unified chips. Although these industrial areas adapt sharing of pre-building blocks without difficulty, the industrial area of software development often builds software from scratch.

Hence, the goal of CBSE is to adopt the whole idea of building a product from a pre-build blocks – components. The same way as an electronic chip provides a set of pins with a well defined inputs and outputs, the aim of components is to provide well defined input and output allowing to put the components into a system and run. There is a lot of approaches to implement this idea and some of them will be detailed in following sections.

Before the deeper explanation of concepts and terms such as a component, a component model and a component framework, which will be detailed in Sections 2.2, let us start with several motivations of CBSE mentioned by Bachman [11]:

- *Independent extension* – Legacy non-component 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 misused communication protocol in two stand-alone systems).

- *Component markets* – Component models themselves define standards for components. Together with component frameworks, it defines mechanism of component's deployment, running and usage. That is, no explicit definition of 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 specific functionality contains only 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 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 incorrectly design technical system structure.

2.2 Components

The definitions of components vary and, so far, there is not only one general definition of what a component is [17]. It is obvious we build components to compose a variety of final applications. Hence, a one of empirical definitions would say a component is a unit of a composition.

This empirical definition is, however, not sufficient. There is a lot of variants of how the components may look, how they are created and how they compose a final product.

The very term "component" is also used for different areas of software which have a little common characteristics together. On the one hand, a bean defined in Enterprise Java Bean (EJB) [33] or a bundle created for OSGi [85] actually encapsulate functionality that may be repeatedly used in different systems. For that reason, the empirical definition fits to the mentioned empirical definition. On the other hand, vendors of software time-to-time promote components of an application, but the application itself is a monolithic one. The distinction to components is, in this case, only a logical or a commercial one.

Considering discrepancy of terms and understanding, it is desired to find a preciser definition. One of the often used definition is in Szyperski's book [96]. 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."

However, other authors also tried to answer the question of what a component is. Therefore another definition proposed by Bachman [11] and mentioning three important points reads:

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

Finally, the third definition shown in this section has been proposed by Meyer [71]. It reads:

> A component is a software element (modular unit) satisfying the following conditions:
>
> 1. It can be used by other software elements, its "clients."
> 2. It possesses an official usage description, which is sufficient for a client author to use it.
> 3. It is not tied to any fixed set of clients.

Reading these three definitions it may be realised their are partly equivalent. Although these definitions do not present common understanding of components and CBSE, they have been selected to express the understanding used across this work. Due to the fact this work aims at improving partial aspects of CBSE and does not aim at covering all research around CBSE, other component understanding (e.g. modules activated and deactivated based on licensing) will not be provided here.

The rest of this work deals with software components in terms of mentioned definitions. It namely means (1) a software component is assumed as a unit, typically prepared by a third-party, of a software composition, (2) which has a precisely defined communicating counterpart elements (e.g. interfaces) and (3) is deployed into a software system.

2.2.1 Components and Object-Oriented Programming

For better understanding of components Szyperski [96] provides a relation of the components with object-oriented programming. Namely, unlike a component as a unit of deployment, an object is a unit of instantiation. Each object has 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 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. In contrary to these rules, component frameworks such as EJB, Spring [94] 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 program which provides a functionality and users know only its inputs and outputs. The users call the functions with inputs and expect outputs. The inner implementation of the functionality remains hidden. Moreover, the black-box software parts often provide a contract restricting input data for which the software parts are capable of guaranteeing their outputs.

If the user of the program is able to look to the source code, it is a white-box. The white-box is generally more problematic to replace an old program by a new one [96]. 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. A client may e.g. change a sequence of calls, modify somehow input and even output values to reach e.g a maximal performance of the program.

White-box representation of a program poses problems when the program is replaced by another version. Although the new version may work well, some clients may rely on the inner representation of the older version. For that reasons the black-box program better suites for future replacement.

Taking components into account, the black or white-box nature remains valid. All components designed as black-boxes are more suitable for future replacement. Since the components primary goal is to be replaceable, it even more highlights the need for black-box components. Therefore, this work aims at treating components as black-boxes.

2.2.3 Component Interconnections

Considering a component as a unit of deployment, the component needs to communicate with other components. Different systems use different approaches to connect components. Interfaces, events, shared memory, connectors, etc. are used for components to exchange data.

Often, components use the provided and the required role of communicating counterpart elements where compatible provided and required pairs are matched together to establish communication channels. This understanding will be used in this work.

Another question raised with the contract definition is how many functions should a component publish and how many the component should require. An ideal component is fully re-usable with only a set of useful functions. It is, however, often in contradiction. When a component provides a too wide set 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 [96]:

> *Maximizing reuse minimizes use.*

Every component should offer the right set of functions with minimal dependencies on other components.

2.3 Component Models

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 with each other. For that reason, the conformance with a component model is the most important addition to the world of components.

Definitions of what the component model is have been proposed by other authors. First of all, let us cite the work presented by Bachman [11]. It targets a role of a component model in terms of types compatibility, components interaction and resource allocation. The definition says the component model should impose:

- *Component types.* They are 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 interfaces.
- *Interaction schemes.* The component model should specify a component location (e.g. where component are stored and deployed), a protocol to communicate 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.

Another definition addresses a semantic and syntactic role of the component model together with the role of a composition arbiter. The definition has been proposed by Lau [70] and it reads:

> A software component model is a definition of:
>
> - the semantics of components, that is, what components are meant to be,
> - the syntax of components, that is, how they are defined, constructed, and represented, and
> - the composition of components, that is, how they are composed or assembled.

Let us continue with a definition presented by Crnkovic [30] concerning properties and mechanism of component composition. It reads:

> A Software Component is a software building block that conforms to a component model. A Component Model defines standards for (i) properties that individual components must satisfy and (ii) methods, and possibly mechanisms, for composing components.

As a last definition of this section, let us mention the Weyuker's work [99] highlighting a composition arbiter role of the component model. Their definition says:

> A [component is a] software element that conforms to a component model and can be independently deployed and composed without modification according to a composition standard.

Summarising these definitions, the component model gives a uniformity to components and their composition. The component model prescribes component communication means, deployment, binding, resource usages, etc. The component model ensures the components are correctly bound once deployed to a system, the communication means are established and resources allocated.

Another use of the component model is to ensure sufficient quality. The component model may define typical software requirements e.g. a deadlock free computation, manages race-conditions, synchronization etc. The component model may also support requirements such as performance, memory consumption, etc. generally covered by extra-functional properties that are of the main concern of this work.

A success of CBSE depends on a component market [11]. When developers produce a component it is published by a vendor to a marker where it may be bought by an architect of a final application. It is expected that the component works equivalently in an original developer environment as well as in a customer environment. A degree of such assurance depends on a component model and a degree a component model is capable of verifying the conformance.

2.4 Component Framework

A component framework is essentially an implementation of a component model. It supports all mechanisms such as deployment, synchronization, life-cycle, communication of components as long as a component model supports them.

Component framework works like an operating system [11]. It also manages processes (components), life-cycle, receives resource requests and decides their assignment. The framework also allows components to communicate with each other the same way as an operating system does. Operating systems typically run all the time while the processes are variously started and stopped.

Although a lot of component frameworks also run all the time components are started and lately stopped, it is not necessary needed. A component framework may be also an implementation of functionality which components explicitly invokes. In contrary, Bachman [11] says: *"The trend in component technologies seems to be towards framework as independent* *implementation, making the operating system analogy quite apt.".* Consequently, this claim is also supported by practically used frameworks such as Java EJB, Spring, OSGi, which are consistent with the operating systems analogy.

Chapter 3

Extra-functional Properties

While the previous chapter has described fundamentals of component programming, this chapter details extra-functional properties (abbreviated as EFPs) as an important aspect of software development. Although they are used in other fields of software development this work targets mainly their relation to CBSE. This section overviews motivation of taking EFPs into account first, then several definitions and references concerning EFPs are mentioned.

A research concerning EFPs is considerably wide. Purpose of following sections is, therefore, to summarise only a limited set of selected approaches.

3.1 Motivation

Section 2.3 has shown that a component model ensures the component compatibility in different environments as long as the components are conformant with the component model. However, current industrial component models typically guarantee components will work in customer environment only in terms of functional aspects such as correct using of communication channels, resources binding, component's life-cycle etc. that is insufficient for a reliable conformance verification.

Extra-functional aspects such as (1) performance properties: *speed*, *response time*, *memory consumption* or (2) user requirements: *marketability*, *price*, *regular updates*, *technical support* or (3) behaviour properties: *synchronisation*, *concurrent access*, *deadlock free computation* should be also taken into account.

Typically, these properties should be taken into account in the phase of component binding and therefore included in the process of verifying component compatibility. Then, a combination of functional and extra-functional aspects provide verified and compatible components meeting user requirements. In following sections, extra-functional properties attempts will be shown in a form of stand-alone frameworks, languages or formalisations.

3.2 What Others Say

Let us start by pointing out that the very term *extra-functional properties* lacks unified understanding. The definitions available, so far, are vague and varied. However, attempts to collect and classify EFPs have been done. For instance, M. Glinz collected some definitions in his work [42]. Some of them are shown in Figure 3.1 (provided at the end of this Chapter).

In addition to these definitions, Glinz focuses on a main problem of EFPs. The main problems are in *definition*, *classification*, and *representation*:

- *Definition problem* - there is, so far, terminological and also conceptual misunderstanding. Other works use concepts of properties, characteristics, attributes, qualities, constraints and performance, however, they denote incompatible concepts.
- *Classification problem* - there is also no general understanding. The classification problem is in the work [42] summarised as:

  > Davis [31] regards EFPs as qualities and uses Boehm's quality tree [14] as a sub-classification ... The IEEE standard 830-1998 [49] sub-classifies non-functional requirements into external interface requirements, performance requirements, attributes and design constraints, ... The IEEE Standard Glossary of Software Engineering Terminology [48] distinguishes functional requirements on the one hand and design requirements, implementation requirements, interface requirements, performance requirements, and physical requirements on the other. Sommerville [93] uses a sub-classification into product requirements, organizational requirements and external requirements."

- *Representation problem* - each application of non-functional requirements is usage dependent. Another problem is a lack of consensus where to document non-functional requirements. It may be a separate chapter in the software requirements specifications. The Rational Unified Process [52] recommends to define the requirements in use-cases.

Distribution of EFPs presented in another Glinz's work [41] considers classification and concerns. It suggests a classification of the requirements into groups: *kind*, *representation*, *satisfaction*, *role.* In addition, requirements are denoted by *concerns* divided into groups: (1) *project requirements*, (2) *system requirements* and (3) *process requirements.* System requirements build on other sub-concerns: (1) *functional requirements*, (2) *attributes* and (3) *constraints.*

Furthermore, classification of EFPs has been developed by Bachman in his work [11]. It defines three main groups of extra-functional properties:

- *Behavior* concerns the 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. – an explicit verification checking whether a component is thread-safe and synchronised is barely supported.
- *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.

These selected definitions show that there is a lot of understanding, definitions and applications of EFPs. Another detailed survey and attempt to classification has been presented by Chung [27]. A unified definition which would be general but still valid for multiple areas does not exist. Therefore, research tries to address these areas independently. Some of the areas are:

1. a transformation of user requirements, expressed in a natural language, to a formal language: Hussain [47], Nordin [66];
2. a language or any other formalisation of EFPs allowing their general automatic processing: Aagedal [3], Gu [44];
3. systematic way computing how EFPs are influenced by other EFPs: Zschaler [104], Defour [32];

4. a relevance of EFPs to a concrete area of usage: Lammana [63];
5. a simulation or measurement of EFPs: Potužák [87].

The first point covers the domain of collecting and processing user requirements. Since users specify their requirements to a system in a natural language, techniques mining EFPs form the specifications are targeted.

The second point covers a formal writing of EFPs allowing its invocation in the same manner as ordinal source code is compiled and run.

Obviously, a lot of characteristics depend on other characteristics. Performance of one part of a system depends on other parts. For that reason, the third point covers the dependency of EFPs with each other.

The fourth point is important for components. Some qualitative aspects often vary for different usage and EFPs may be relevant in one context while they are barely relevant in another one. For that reason, the context in which concrete EFPs are deployed should be taken into account.

The fifth point covers an area of component measurement to obtain EFPs. It is an alternative approach to model precise behaviour of the components, in which a component is a block-box and its characteristics are explicitly obtained by its measurements and simulations. Such an approach leads to approximated characteristics, however, it prevents a need for a lot of detail behaviour models.

To conclude this section, let us state a new definition of extra-functional properties developed for the purposes of this work:

> An extra-functional property is an attribute holding any information, explicitly provided with a software system, to describe characteristics of the system apart from the system's genuine function, to extend system's contract, supported by technical [computational] means.

3.3 Extra-functional Languages

This section introduces current research approaches to definitions of extra-functional properties expressed as specialised languages. There is a group of languages targeting specialised area of usage. They typically suit specialised components for specialised software. A context of usage is less important. On the other hand, there are other approaches working with general EFPs. They are typically more suitable for general components and their context-independent usage.

This section first focuses on several approaches which define EFPs for concrete areas of usage. They generally better succeed in specialised applications, though their application to general components is limited.

Furthermore, this section provides an overview of more general languages. They are typically stand-alone notations that may be used by various systems ranging from general components, through web-services to specialised systems.

3.3.1 HQML

A Hierarchical QoS Markup Language (HQML) [44] is designed as a XML-based language targeted at the Service Oriented Architecture. HQML uses a XML language for its simplicity and popularity.

It uses three layer structure:

- *User Level* – defines quantitative criteria in textual 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 serves as a specification of all kinds of application QoSs (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 middle-ware entities of the system independently of underlying resources such as hardware, OS, etc.
- *System Resource Level* – defines different resource requirements. When a concrete resource is available it allows to associate it.

Each of these layers have respective elements in the XML writing. For instance, a user requirement (User Level) to a "smoothness" video playback may be written as:

```
<UserLevelQoS> high </UserLevelQoS>
<UserFocus> smoothness </UserFocus>
<Price unit = "$"> 1 </Price>
<PriceModel> flat rate </PriceModel>
```

An application (Application Level) providing a video playback may provide a set of characteristics:

```
<Range unit = "fps">
```

```
<UpperBound> 40 </UpperBound>
<LowerBound> 30 </LowerBound>
</Range>
```

The application may run on a system (Resource Level) defining the set of resources:

```
<Delay unit = "ms"> 100 </Delay>
<LossRate unit = "%"> 3 </LossRate>
<Jitter unit = "ms"> 10 </Jitter>
```

Although the HQML [44] provides a rich description of the XML elements allowing the writing shown above, it is not detailed how these characteristics are evaluated. For that reason, it is unclear how user requirements are mapped to the provided application level together with provided resources.

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

1. The HQML Executor interprets user 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 searches in HQML Profiles (which is Application Level) to find a profile matching user 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 called *QoSTalk* covering the presented solution.

HQML seems to serve as a comprehensive language which targets different level of an application. Although the paper [44] 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 latter one is challenging.

3.3.2 SLang

SLang [63] 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 service provider and a client.

It defines a *horizontal layer* which basically respects 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 differ among layers. For instance, a web service may offer a throughput as well as a database may do so, 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 applications 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 easy the comparison of offers. They define a set of main concepts to reach both goals:

- *Parametrisation* – 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.

The work [63] describes in detail the language itself, however, these points are barely developed in details which limits understanding of how they would be reached.

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: (1) *Application* – between application or web services and components, (2) *Hosting* – between a container and components, (3) *Persistence* – between a container and a storage, (4) *Communication* – between a container and network providers.

The horizontal layer uses the SLA of kinds: (1) *Service* – between components and web services, (2) *Container* – between containers, (3) *Networking* – between network providers.

For instance a *Persistence* layer may be written using SLA as follows:

```
<Provision disk_space="400"/>
<Availability>97%</Availability>
<Reliability>90%</Reliability>
<Maintenance recovery_time="1" scheduled_outages="17"
  routine_maintenances="24"/>
<Query_response_time average="30" maximum="48" minimum="21"/>
<Data_integrity>97%</Data_integrity>
<Security encrypted_storage="true" encryption_method="DES"
  certificate="false"
user_authentication="true" intrusion_detection="true"
  virus_scanning="false"
eavesdrop_prevention="true"/>
```

SLang targets different scales of values in terms 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. Although SLang allows to bind properties to a concrete feature in which properties are valid, the work [63] does not explain a mapping of values with each other. Since concrete feature contains its values, it seems the matching would be performed with incompatible values and for that reason a re-mapping would be desired.

3.3.3 TADL

An Architecture Description Language for Trustworthy Component-Based Systems (abbreviated as TADL) [72] is a specialised language describing the whole architecture of a system. TADL is a language specialised for trustworthy systems and explicitly concerns extra-functional properties as part of an architecture of systems. It specifically targets structural, functional and extra-functional properties to define a system 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 scheduleability. 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.

   The definition of the constraint where the maximum time must be up to 40 looks like:

   ```
   TimeConstraint TC {
     // other definitions related to this time constraint
     float maxSafeTime = 40;
   }
   ```

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

   The Example shows the constraint where the temperature must be in the range $(10, 50\rangle$ (inspired by the example shown in [73]):

   ```
   DataConstraint DC {
     CurrentTemperature current;
     // other definitions related to this data constraint
     10 < current.temperature <= 50;
   }
   ```

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.

   Example (from [73]):

   ```
   Security {
   ```

```
    Administrator admin; // user
    Operator operator;  // user
    SwitchOn switchOn; // user privilege
    SwitchOff switchOff; // user privilege
    ON on; //service
    OFF off; //service
    Privileges-for-services(on, switchOn, admin);
    Privileges-for-services(on, switchOn, operator);
    Privileges-for-services(off, switchOff, admin);
  }
```

The last three lines associate the user privilege for the user *admin* or *operator*. The user gains privilege *switchOn* or *switchOff* for the service *on* or *off*.

TADL deals with extra-functional properties at a design time of an application. However, due to the targeting of TADL to trustworthy systems, the set of supported extra-functional properties is limited. As a result, it defines only two types of EFPs: security and safety.

3.3.4 NoFun

The NoFun [36] language is a representative of a 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.

Attributes may be basic or derived where derived attributes are derived from basic ones. Basic attributes belong to the data type which defines them while derived attributes are computed by the equation: $C_i \Rightarrow P = E_i$. The meaning of the formula is that the value P is a derived attribute which is equal to an expression E_i if the boolean condition C_i is *true*. In addition, E_i yields a value in a P's domain (the domain is a data type of the value).

The example showing the computation of the derived attribute *reliability* depending on two simple attributes: *error recovery* and *fully portable* looks like:

```
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.

Example showing a definition of the behaviour specification is:

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

Let us point out the exact meaning of the lines of the example shown above is not clearly explained in [36]. *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. However, a finite set of these variables and operations is not given.

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.

To sum up, NoFun provides ideas of which information should an EFP contains. It makes NoFun a rich 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, only overall ideas are explained in [36]. Operators and functions in the notation are not briefly defined and hence the semantics and the complete set of allowed operators and functions remains unclear.

3.3.5 QML

A language called QML [37] 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: (1) a specification of QoS is separated from the code of an existing system, (2) it allows to specify provided and required QoS properties, (3) it provides mechanisms to determine whether the client needs for QoS are fulfilled, (4) it supports a refinement of QoS, because the 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 Contract 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 Contract 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 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, contracts 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.

The example of *Reliability* expressed in QML looks like (Example from [37]):

```
CallServerReliability = Reliability contract {
  MTTR {
      percentile 100 <= 2;
      variance <= 0.3
      };
  TTF {
      percentile 100 > 0.05 days;
      percentile 80 > 100 days;
      mean >= 140 days;
      };
  availability >= 0.99999;
  contAvailability >= 0.99999;
  failureMasking == { omission };
  numOfFailure <= 2 failures/year;
};
```

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

QML is a comprehensive language covering creation of EFPs and attaching them to objects. 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 individual environment, but they do no describe a mechanism to configure objects automatically for different environment.

3.3.6 QML/CS

The work [103] introduces comprehensive formal approach to EFPs partially implemented as the language called *quality modelling language for component-based systems* – QML/CS.

That approach introduces formalisation in a form of Temporal Logic of Actions (TLA+) [64] that specifies a system as a set of behaviour consisting of states with traces among the states. The goal of the approach is a semantic framework allowing to describe EFPs on components and then describe how these components are used.

Providing a certain level of abstraction the framework works with five specification types: (1) *services* are units of functionality with EFPs that may be influenced with each other, (2) *components* compose services or other components ans thus compose also EFPs, (3) *resources* are provided by runtime environment, (4) *container* is component's runtime providing EFPs the components require to work properly, (5) *measurements* represent EFP dimension of a system.

Having a set of measurement M a non-functional property is a set of all formulas $\Pi_{Nf}(M)$ constraining the values in M. For instance, a system that guarantees each execution is maximally two times slower than the previous execution may look like: $t^{'}_{execution} \leq 2t_{execution}$. The $t_{execution}$ property refers to the previous execution time of the state automate while $t^{'}_{execution}$ refers to the current execution state.

An extra-functional property is defined as a constraint over measurement with several measurement specifications: (1) *intrinsic* is used for constraints valid for a component implementation, (2) *extrinsic* is used for single services to express user expectation, (3) *resource* and (4) *container* are used for expressing EFPs of required resources and EFPs provided by a container respectively.

Consequently, evaluation of a whole system is a validation if a combination of intrinsic, resource and container specifications implies extrinsic specification. A system fulfilling it is called a *feasible system*. The work [103] furthermore details how all these basic concepts are defined and evaluated using TLA+ that we do not detail here.

Although the work [103] developed very strong formal framework, the transformation to a practical QML/CS language is not complete and only partial approach is provided. The formal model behind may provide accurate results, though a lot of models required to describe each service, component, resource, etc. may burden developers and prevent practical usage.

3.3.7 Ontologies

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 their semantics and relations to each other.

For instance, [98] extends the Web Service Modelling Ontology (WSMO)[1] to better support EFPs and proposes a service comparison method using quality characteristics.

Another work proposed by García [38] develops 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 [39]. They use WSMO to express QoS. The WSM language (WSML) axioms are used for defining EFPs. Each EFP 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.

3.3.8 CQML

An approach proposed by Aagedal is the CQML [3] 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 of how this composition is treated.

CQML defines a few basic constructs concerning EFPs:

QoS Characteristics is a basic building block. One QoS characteristic 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

[1]Available at: http://www.wsmo.org/ (2012)

exist in the time the QoS characteristic is being developed. For instance, a characteristic with a frame output property may look like:

```
quality_characteristic frameOutput {
  domain: decreasing numeric milliseconds;
  mean;
}
```

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 with 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. For instance a QoS statement may be defined:

```
quality high {
  compulsory best-effort frameOutput > 25 with limit 20;
  threshold best-effort delay < 5 with limit 8;
}
```

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. For instance, the above examples may be stored in a profile:

```
profile goodCamera for myFastCamera {
  provides high;
}
```

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.

3.3.9 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 should not be avoided.

The CQML mentioned before has been extended by other authors. An extending language proposed by Röttger and Zschaler is called CQML+ [90]. They aim at 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. As a result, 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 a potential weakness as long as different resources require different treatment.

They introduced another extension to CQML that is a definition of a tuple that allows to associate more resources in a one condition. The semantics is that all resources in the tuple must be available concurrently. An example of a resource with the tuple is pressented in [90]:

```
resource cpu {
  quality_characteristic cpu_demand (r: Resource) {
  domain: tupel {
    period (r),
    execution_time (r)
  };
  invariant: execution_time < period;
}
```

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

3.3.10 Deployment Contracts

Deployment Contracts [68] presented by V. Ukis are focused on detecting possible conflicts among components or a component and its execution environment.

Deployment Contracts (DCs) define 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 component method, a method's parameter or a return value. In the prototype of DC, meta-data are implemented as .NET annotations. For instance, a method may marked as spawning one thread:

```
public class A {
  [SpawnThread(1)]
  public void Method1(...) {...}
}
```

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

DCs is presented in a form of .NET prototype implementation, however, no generalisation of DCs is developed. As a result, DCs provide an implementation of about 100 different deployment contracts specified in [69] with an algorithm of evaluating them. The algorithm, however, branches to evaluate every case of implemented DC rather than generalising in a general purpose algorithm. In addition, the algorithm itself focuses on the conflict prevention rather than selecting the most suitable component candidate.

3.4 Modelling of Extra-functional Properties

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

Since UML [83] has been acquired as a widely used modelling notation, other works introduced the modelling based on it. UML provides the rich palette of diagrams, but a class diagram is most often used. It leads others to prepare diagrams based on UML's class diagram with the UML concept of *stereotypes* allowing to extend the basic UML elements to support EFPs.

3.4.1 The UML Profile for CQML

Aagedal introduces [3], together with CQML, a UML profile covering the expressiveness of CQML. He defines a set of stereotypes shown in Figure 3.2 that correspond to CQML keywords (including *QoSStatement*, *QoSCharacteristics*, *QoSProfile*, *QoSQuality* and *QoSComponent*). The introduction of these stereotypes allows to model EFPs the same way as they are written in the CQML's language.

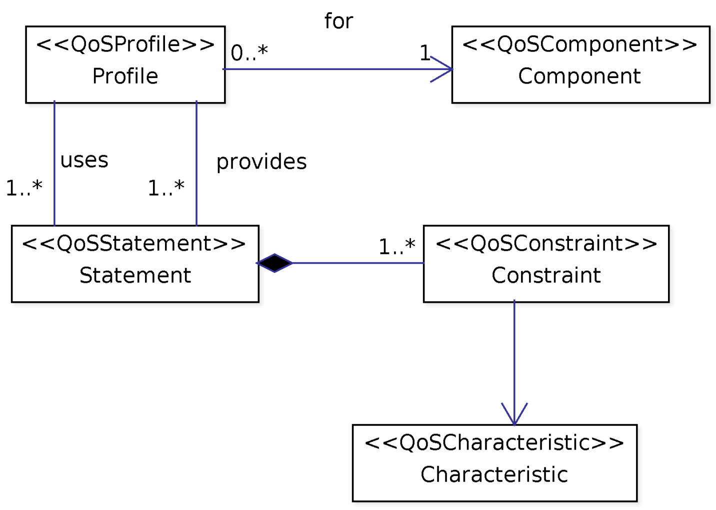

for
<<QoSProfile>> Profile
0..*
1
<<QoSComponent>> Component
uses
1..*
provides
1..*
<<QoSStatement>> Statement
1..*
<<QoSConstraint>> Constraint
<<QoSCharacteristic>> Characteristic

Figure 3.2: CQML UML Profile

3.4.2 The UML Profile for NoFun

Another work presented Guadalupe Salazar-Zárate and Pere Botella [91]. They introduced UML profile covering NoFun.

They first define a stereotype *NF-attribute* for non-functionality expressing an EFP. *NF-attributes* model simple properties as well as derived ones. 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 secondly define stereotypes *NF-Requirement* and *NF-behaviour* with an obvious meaning in terms of NoFun concepts. Furthermore, a set of EFPs expressed in NF-behaviour is attached to a class labelled by the stereotype *ImplementationClass* and the connection is marked by the stereotype *has behaviour*.

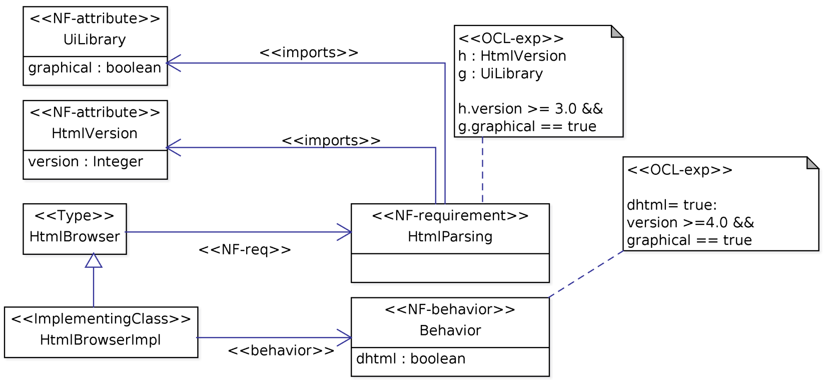


Figure 3.3: NoFun UML Profile Example

Figure 3.3 shows an example definition in which a `HtmlBrowserImpl` browser component requires at least HTML in version 3.0 and a graphical UI library to work. A behaviour of the component guarantees correct dynamic HTML 4.0 processing according to the OCL expression defined for its behaviour.

3.4.3 The Marte UML Profile

UML Profile for MARTE (The Modelling and Analysis of Real-Time and Embedded systems) [82] has been introduced by the OMG group and has already became a standard. It has been developed 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 modelling or analysis support or they may provide both.

MARTE is organised as a hierarchy of profiles and sub-profiles. 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 the time and it also deals with time representation in applications.
- *Resources* – it deals with resources the applications demand from the system.
- *Allocation Modelling* – it is used for allocating functionality to a responsible entity.

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

The 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 Modelling* sub-profile. Notice that component-based systems are explicitly supported by the *Generic Component Model* profile.

The model-based analysis is allowed mainly by *Quantitative Analysis Modelling* 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 modelling an application correspond to the analysing domain.

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

3.4.4 The OMG's Quality of Service Profile

Another work, proposed also by the OMG group, aims at providing a general model for Quality of Service (QoS). UML Profile for Modelling Quality of Service and Fault Tolerance [81] provides the ability to model EFPs by the means of UML. This profile introduces new stereotypes that covers elements of extra-functionality and their relations.

The profile consists of several main building blocks.

QoSCharacteristic represent quantifiable characteristics of a service. It is basically an extra-functional property where the QosCharacteristic may first have a set of parameters (*QoSParameter*). Each characteristic second has a dimension (*QoSDimension*) that stores: a data type, ordering of the values, a measuring unit and a 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*) where 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*) with an assigned constraint that evaluates its validity.

QoSConstraint defines constraints of the *QoSCharacteristic* and it limits values allowed for application requirements. There are other classes inherited from *QoSConstraint*: *QoSRequired*, *QoSOffered* and *QoSConcrat*. Their usage is for application service requirement, 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*s 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 The Component Quality Model

In a case EFPs are described using any presented language or model, the question remaining open is which properties to use. Since the general consensus still does not exist, Alvaro [6, 7] categorises component quality characteristics which can be used as EFPs.

The authors follow the standard terminology defined by ISO/IEC 9126 [50], but they made a few modifications to better suit component-based development. The resulting Component Quality Model (CQM) includes: (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 the 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 and distinguished as either runtime or life-cycle ones.

According to ISO 9126 [50], an attribute is a measurable (physical or abstract) property with defined metric. The metric defines both the measurement method and the scale.

The CQM uses the following metrics: (1) *presence* to indicate whether an attribute is present, if so, the *string* value contains information of 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.

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

3.5 Extra-functional Properties in Component Models

The previous section has introduced several approaches expressing EFPs mostly as stand-alone notations. This section will show component models that include EFPs as their integral part. However, to our best knowledge, none of the frameworks use any presented EFP languages. Why the existing EFP notations are not applied in existing component systems, and if it may point out their insufficiency, remains unclear.

Although a lot of important approaches to EFPs exist, this section shows only several selected ones. The goal is to show important directions to deal with EFPs while another rich survey may be e.g. found in the Crnkovic's overview [30, 29].

3.5.1 Palladio

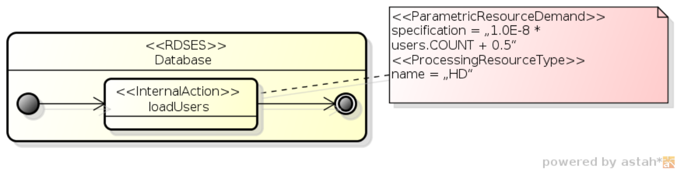
Palladio [13] targets the development process in component-based development. It includes a component developer, a system architect, a system deployer and a domain expert. A system is modelled by a set of models where each model covers different roles in the process.

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

The detailed scenario is: a component developer annotates 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 practice 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 grey-box.

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.

For instance, a RDSES for a simple process loading users from a database may be modelled as:



The UML note shows a stochastic expression specifying the resource demand for a hard-disk (HD) resource. All states and their transitions are modelled in the same manner to describe a Palladio system.

A domain expert role is to define a 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 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.

3.5.2 Robocop

The ROBOCOP model [76, 15] 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 scenarios. This is a limited extent typical for the domain, however, combined with the performance model of hardware blocks it allows the above mentioned analyses.

For instance, a definition of EFPs comprising processor and memory characteristics may in ROBOCOP look like:

```
load_img(imgdata)
  referencecpu = RISC 600MHz
  cycles 1234*imgdata worst-case, 300+imgdata best-case
  memory claim 250B mean-case, release 20B mean-case
```

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

3.5.3 ProCom

An approach to integrate EFPs in component models using structured attributes is presented in [92] 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.

For instance, two values may be assigned to a static memory usage property. Additional meta information distinguishes these values:

```
Static Memory Usage
1) value: 15, kB, version: 2, timestamp: 080220 #10:00,
   source: measurement, platform: X
2) value: 10, kB, version: 1, timestamp: 080120 #17:44
   source: estimation
```

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.

3.5.4 Enterprise JavaBeans

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 e.g. 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 [95] component model is one of the strongest industrial frameworks, used in the application and data layers of enterprise applications. Despite a 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. It 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).

For instance, a shopping cart may be state-full and transactional JavaBean where a user in a role "Customer" may call the beans's `initProduct` method:

```
@Stateful  // state
@Local     // locality
@Transaction(REQUIRES_NEW)  // transaction
public class ShoppingCartImpl implements ShoppingCart {

  @PersistenceContext(type=EXTENDED)
  EntityManager em;

  private Product product;

  @RolesAllowed("Customer") // security
  public void initProduct(String name) {
    product = (Product) em.createQuery("select p from Product p
    where p.name = :name")
      .setParameter("name", name)
      .getSingleResult();
  }
}
```

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).

3.5.5 Koala

Koala component model [28] [97] is used for embedded systems such as TV sets with late binding of interfaces of reusable components. To maximise re-usability, Koala strictly separates components and configuration.

A component is a unit of design with a specification and an implementation. It consists of Interface Definition Language (IDL) to define interfaces, Component Definition Language (CDL) to define components and Data Definition Language (DDL) to specify data in components. The definitions created in Koala are compiled to a programming language, for instance, C.

Several explicit means are used to maximise reuse: (1) *compounded components* are hierarchical components where one component consists of other ones, (2) *module* is a interface-less component used to glue component interfaces. It, in essence, serves as a kind of adapter where the module implements all functions of all interfaces, (3) *functional binding* allows to bind interfaces with different names. The binding is allowed as long as a provided interface fulfils all method required on a bound (provided) interface, (4) *switches* covers a special type of functional binding in which conditions decide interface to be bound.

Although the Koala documentation [97] does not explicitly mention EFPs, Fiukov shows in his work [35] the implementation of the static memory usage property in Koala. This property is provided through an additional analytic interface created and filled for each component existing in the design.

3.5.6 KobrA

Since KobrA [10, 9] is not a formal language, it is not possible to precisely state what KobrA can handle. KobrA is rather a set of principles applicable to mainstream modelling languages such as UML. Hence, there is a certain degree of flexibility of principles which KobrA may be applied to.

One of the advantages of applying KobrA to UML is the reduction of the complexity of each diagram. Here, KobrA introduces a set of views separating concerns and explains which models should be created, what the model contains and how they are related.

Furthermore, KobrA encourages uniform representation of components regardless of their granularity and the concrete location in the component hierarchy. Each component is required to be described at the same level of abstraction.

Since an important aspect of software systems is the ability to define EFPs, KobrA comes with two separated approaches: (1) *extra-functional requirements* and (2) *extra-functional properties.* Requirements cope with specification while properties with realisation. A requirement is a statement a component developer must fulfil. A property is a fact which must be true on a certain component.

There are two ways of adding extra-functionality to KobrA. They can be added to an existing view as constrains or annotations or they can be defined as a separate extra-functional view.

3.5.7 SOFA 2.0

First-class entities in SOFA 2.0 [23] are components and connectors and the key abstraction is meta-modelling using MOF [80]. Main advantage of such an approach is a possibility to automatically generate components, models, editors, a repository etc.

As a lot of other approaches, components in SOFA communicate via provided and required interfaces or connectors. A distinction of components is to grey-box and black-box entities.

The black-box is refined as a *component frame* representing a set of component interfaces. In addition, the component frame includes definitions of component's behaviour which means tracing of events determined by the method calls between provided and required interface pairs.

The grey-box is a component specified as an architecture implementing component frames. SOFA is a hierarchical component model which has two types of architectures: (1) *primitive* and (2) *composite*.

The primitive architecture is in essence the direct implementation of the component in programming language while the composite architecture is a composition of other subcomponents. The execution of interfaces of composite elements is delegated to subcomponents.

Connectors allow to explicitly model different architectural and communication styles. Connectors create all links among interfaces where a set of interfaces may be connected to a one communicating link (a bus architecture).

SOFA also allows the controlled evaluation via a set of predefined evolution patterns: (1) *factory pattern* uses one component as a factory for other components, (2) *removal pattern* controls destroying of dynamically created components and (3) *service access pattern* allows an access to external services.

The component life-cycle in SOFA supports all typical phases: (1) component development, (2) system assembly, and (3) deployment with execution. The development process starts with creating and committing components into the repository. The architecture of a system is described mainly by frames. The frames are refined by particular architectures in the phase of the system assembly. The whole process works in the top-down manner in which top components are recursively composed from other components until the primitive ones are reached. Finally, an application is assembled together with definitions which components have to be executed and connectors are generated.

On the one hand, SOFA does not contain explicit EFPs, on the other hand, SOFA has advanced design elements such as automatic reconfiguration, behaviour specifications on interfaces or dynamic evaluation of an architecture at runtime. These aspects stay behind a simple functionality of services and may be understood as certain extra-functionality.

3.5.8 UML2.0

UML2.0 [83, 25] has been extended of a possibility to model component systems. A component is a modular unit of a system with well-defined interfaces. Behaviour of components is defined by the required and provided interfaces. UML2.0 models the provided interfaces by a lollipop while the required interfaces are sockets.

The advantage of UML2.0 is in a visual modelling provided by several tools in which a software system is composed of components connected via provided and required interface pairs.

In addition, UML2.0 distinguishes between two types of connectors: (1) *assembly connector* (lollipop connected to socket) is used to connect the required interface with its provided counterpart, (2) *a delegation connector* (arrow) is used to forward required and provided interfaces from inner component to outer components. It is well applicable for hierarchical component models.

UML2.0 does not directly support EFPs, however, EFPs may be added via UML profiles extending fundamental modelling means.

3.5.9 PECOS

PECOS [40, 79] is a system in which a component is a unit of design. The component has a form of specification and implementation. Connections of components are made via ports where connectors are connected to the ports.

A component is a computational element with a name, a number of property bundles and ports, and a behaviour. The ports represent data to be shared among components. The behaviour of a component consists of a procedure that reads and writes data available at its ports.

PECOS furthermore distinguishes between two kinds of components: (1) *leaf component* is a black-box implemented in the host programming language and not defined by the model, (2) *composite components* contains a number of connected subcomponents comprising internal and external ports. The external ports are connected to appropriate internal ports while the subcomponents are not visible outside the composite containing them.

In addition, components are of three types: (1) *passive components* do not own threads and must be explicitly scheduled by the active components. They encapsulate pieces of short behaviour running synchronously, (2) *active components* do own their threads and perform long-lived activities, (3) *event components* are used to model hardware emitting events.

Ports are shared variable allowing components to communicate. A port specifies: (1) *a name* which is unique for one component, (2) *a type* which characterises the data type, (3) *a range* of values that can be passed on the port, (3) *a direction* of three kinds "in", "out" and "inout" indicating whether the component reads, writes, or reads and writes the data.

A system designed in PECOS uses Petri nets to model timing and synchronising behaviour of the system. This may be understood as a certain kind of extra-functionality where developers explicitly model behaviour to prevent dead-locks, broken synchronisation etc. Each component may be simulated with Petri nets separately first, then a scheduler is generated as components are composed to coordinate each Petri net. In addition, each component explicitly sets information such as ordering in which it should be invoked or worse-case execution time and this information is used to generate the scheduler.

3.5.10 Fractal

A component in Fractal [20, 21] is an encapsulated and distinct run-time entity with a set of interfaces. Interfaces have two forms: (1) *server interface* which accesses incoming operations and (2) *client interface* which accesses outgoing operations.

Fractal is composed of a *membrane*, which supports interfaces to introspect and reconfigure its internal features, and a *content*, which consists of a set of other components.

The membrane can have external and internal interfaces. External interfaces are accessible from outside a component. Internal interfaces are accessible only within a component for communication of subcomponents.

The membrane typically contains a set of controllers which superpose a control behaviour of subcomponents. In addition, controllers play the role of *interceptors* to export the external interfaces of subcomponents as external interfaces of the upper-component.

Two means to describe the architecture of an application are supported. First is the nesting of components leading to a hierarchical component system. Second is binding of components via interfaces. Fractal allows two kinds of bindings: (1) *primitive binding* and (2) *composite binding*.

The primitive binding is used for communication between client and server interface pairs in the same address space (may be, for instance, created using pointers or Java references). The composite binding denotes a communication path between an arbitrary number of connected interfaces.

In contrary to other component models, Fractal allows to share one component among several other parent components.

Furthermore, Fractal provides two *level of controls*: (1) *the lowest level* represents a black-box components that do not provide any introspection capability, (2) *upper level* exposing internal structure of components. Different introspective features are provided: (1) *attribute controller* accessing attributes of components via a set of getters and setters, (2) *binding controller* allowing to bind and unbind client interfaces from respective server interfaces, (3) *content controller* allowing to add and remove subcomponents, (4) *life-cycle controller* including methods to start and stop the execution of components.

Fractal component model does not contain means to explicitly define EFPs on components, however, an extension FractalBPC [22] was created to port the SOFA behaviour protocol to Fractal. Hence, Fractal may explicitly define behaviour limitations on interfaces that are certain kind of EFPs.

Feljan [34] mentioned another mean to define EFPs on Fractal components through the notion of attributes. An attribute is a configurable property of a component that may hold any information including EFPs.

3.5.11 Component Systems Constructed From Requirements

The process of construction software systems typically collects user requirements. The requirements must be in CBSE transformed to components and their connections. Although manual developer's interaction traditionally embodies this transformation, Azlin [66] in her work aims at an automatic transformation of the user requirements into a final application composed of components.

The fundamental idea of this process is an incremental transformation of the requirements into components. The components are repeatedly being added to a systems being build until the system is completed. Achieving this process, they use the component model [65, 67] allowing the incremental composition. This component model allows the incremental composition in terms of: (1) the addition of components into existing architecture, (2) behaviour and properties within the incremented architecture is presented.

Fundamentally, the transforming algorithm consists of analysing of verbs and nouns in the requirements. Verbs are divided into three groups: (1) a *computation* verb denotes a data transformation, (2) a *state* verb for computation denotes states, (3) an *event* verb denoting an event that trigger computations. Furthermore, nouns are divided into four groups: (1) a *conceptual component* noun abstracts a candidate component, (2) a *state* noun is used for finding data and (3) a *computation* noun expresses transformation to be done by components.

In addition, the most important phrases are found in the requirements: (1) a *descriptive expression* expressing specific operations, (2) a *control structure* indicating a flow of a control and (3) a *predicate* phrase to identify true/false expressions.

Analysing requirements, the system is incrementally composed in several steps: (1) it identifies nouns and verbs to find out components and actions respectively, (2) once a set of components is identified, the control flow of the components is created, (3) first two steps are used for composing partial architecture, (4) this partial architecture is incrementally added to an existing one, (5) the final architecture is finished and optimised.

The steps used in this process targets mainly functional aspects of requirements, however, the user requirements typically contain also extra-functional properties. Hence, there is a considerable opportunity to extend the algorithm to support extra-functional properties.

3.6 Summary of EFP Languages and Component Models

Figure 3.4 summarises the survey of the presented approaches. Columns in the table shows several aspects of these approaches:

- Specialisation – it shows whether EFPs express only specialised or general properties,
- Semantics – it shows whether EFPs provide explicit means to express the semantics of the EFPs itself or their values,
- Composition – it marks an ability of EFPs to be composed with each other,

- Dependency – it summarises whether EFPs may be parametrised according to different deployment environment.

3.7 Evaluation of Extra-functional Properties

The purpose of this section is to overview some approaches targeting evaluation of EFPs. The evaluation is in this section understood as a process in which EFP values are compared, any formulas concerning these values computed and the result of the comparing is returned. Despite the fact the component models from previous sections contain EFP evaluation means and the fact the evaluation means have been mentioned, the purpose of this section is to more focus on the evaluation process. The rationale is to highlight complexity of the evaluation process arising once EFPs are taken into account.

3.7.1 EFP Comparison and Selection

QoS Negotiation

A run-time selection of a component or a service based on its Quality of Service (QoS), is denoted as QoS negotiation.

Mulugeta and Schill introduce in their work [75] 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) user 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+ *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 service 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 services.

When QoSProfiles and a user profile are established, *Negotiator* tries to find an appropriate service. The task is to find an appropriate service and select the best one in a case there is more suitable services. *Negotiator* relies on Constraint Satisfaction Optimization Problem (CSOP) comprising variables, constraints and objective functions. The task is to assign a value to each variable to satisfy all constraints. All suitable results that satisfy the constraints 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. *Contract* holds mainly information about the selected client profile, the server-side profile and the user profile.

The QoS-Based Web Service Discovery

Yan's work [101] proposes an extension of UDDI (Universal Description Discovery and Integration) of the selection of the most suitable web services based on an algorithm concerning QoS and a ranking mechanism. The fundamental idea is to enrich web services in UDDI of QoS characteristics first, then user requirements are matched with the QoS taking a relevance factor into account. The matching results in two normalised matrices with QoS offers and QoS requirements. Finally, differences of each element is computed and a service with the lowest difference is selected.

In detail, QoS offers are attached to services in a form of *attribute name*, *attribute type*, *attribute value*, *attribute unit* and *constraints*. QoS requirements are expressed using the same attributes with three additions: *weight* indicating an importance for a user, *direction* is an expected tendency of the value, and *relationship* is used for expressing relations between attributes.

Once QoS offers are stated on services and requirements are stated by users, the matching algorithm starts. The algorithm works with the matrix of provided QoS: A_P which is the $m \times n$ matrix meaning that m QoSs are attached to n services. Another matrix A_R is the $1 \times n$ matrix (vector) expressing n user requirements. In a real situation, a size of required and offered QoS vector is not the same. The work [101] solves this problem by normalisation mechanism which is omitted here.

Taking both matrices A_P and A_R, a difference vector is computed between the matrix A_R and each row of the matrix A_P taking the weight into account: $diff_j(A_P, A_R) = \sqrt{\sum_{i=1}^{m} W(A_{P_{i,j}} - A_{R_{1j}})^2}$ where $j = 1..n$. It results in a vector of differences and a service with a lowest difference is selected.

3.7.2 EFP Dependencies

The EFPs Dependency in CQML

Zschaler introduces in his work [104] 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 parameter 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. 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 EFPs are defined, it may be preferred to use approximated values rather than precise numbers. For instance, when a system requires a service with a response time equal to 5ms, it may also accept a response time equal to 6ms. This rationale leads to an idea of defining intervals, or – in other worlds – limiting values.

Properties Synthesized from Components

Hamlet [45] summarises in his work several methods of synthetically obtaining EFPs form components. Firstly, the work points out that the dependency of components must be taken into account when calculation EFPs. For instance, behaviour of one component depends on its input provided from another component.

It is proposed to divide a major function of a component into a set of sub-domains with probabilities of each sub-domain. A set of steps a component developer must do: (1) the developer must decide appropriate sub-domains, (2) the runtime characteristics of sub-domains are measured, (3) a handbook with all sub-domains and their measurement is created, (4) a system developer calculates connection of components, (5) a first component is run and the propagation of values in the connected component is observed, finally (6) weighting behaviour of each combination of components the behaviour of the final system is estimated.

The Extra-functional Contract

The work [32] proposes a new language called QoSCL created as an extension of UML 2.0. It allows to explicitly describe EFPs and their dependencies. The main features of this language are: (1) definitions of qualities on services, (2) a level of the quality on the required and provided side of the service, (3) the dependency of EFPs on a provided side influenced by the required side.

The main building blocks of QoSCL are: *ComponentQoSCL* is an extension of the UML2.0 Component Model containing provided and required *ContractType*. *ContractType* is a specialized interface having attributes with *Operations* and *Dimensions*. *Operation* specifies behaviour of each component while *Dimension* specifies measurement of the quality level. Furthermore, dimensions allow to define pre-condition, post-condition and a body of the contract. It may optionally include its own *ContractType* expressing EFPs dependencies with other qualities.

The QoSCL allows to express three kinds of relations: numerical constrains, mathematical functions, or empirical rules. Dependencies of EFPs may be expressed using functions of the provided properties on required properties. For instance, a mathematical function determines the resulting provided property depending on a required property: $memory\_consumption_{prov} = \sum_{i=O}^{N} memory\_consumption_{req_i} + 10$.

3.7.3 Treatment of EFPs from Design to Run-time

Ainger [5] overviews in his work 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.

References of extra-functional properties	
IEEE 610.3-1989 [1]	The degree to which [a component] meets requirements or customer/user needs or expectations
IEEE 610, 9126 [1, 50, 6]	EFPs are qualified: "quality characteristic, factor and/or attribute"
IEEE 830 [2]	The terms "performance" and "attributes" are used
Franch, NoFun [36]	It uses the name extra-functional characteristics
Anton [8]	EFPs describe the non-behavioural aspects of a system, capturing the properties and constraints under which a system must operate
Alan [31]	The required overall attributes of the system, including portability, reliability, efficiency, human engineering, testability understandability, and modifiability
Jacobson [52]	A requirement that specifies system properties, such as environmental and implementation constraints, performance, platform dependencies, maintainability, extensibility, and reliability. A requirement that specifies physical constraints on a functional requirement
Kotonya [62]	Requirements which are not specifically concerned with the functionality of a system. They place restrictions on the product being developed and the development process, and they specify external constraints that the product must meet
Mylopoulos [77]	Global requirements on its development or operational cost, performance, reliability, maintainability, portability, robustness, and the like. (...) There is not a formal definition or a complete list of non-functional requirements.
Ncube [78]	The behavioural properties that the specified functions must have, such as performance, usability
Robertson [88]	A property, or quality, that the product must have, such as an appearance, or a speed or accuracy property
Wiegers [100]	A description of a property or characteristic that a software system must exhibit or a constraint that it must respect, other than an observable system behaviour
Glinz [42]	A non-functional requirements is an attribute of or a constraint on a system

Figure 3.1: Overview of approaches referring EFPs

Approach	Specialisation	Semantics	Composition	Dependency
NoFun	General		Logical formulas	NoFun Behaviour
CQML	General			
CQML+	General			Resource definitions
Ukis's DC	Specialised			Static values
TADL	Specialised		First-order predicate logic	Architecture per usage
HQML	Specialised	Named values		System resource level
SLang	Specialised	Named values		
QML/CS	General	Models	Constrain satisfaction optimisation problem	Resource model
Palladio	Specialised		Models	RDSES
Robocop	Specialised			
ProCom	General	Validity constraints	Attribute composition	Validity Constraints
EJB	Specialised			
Koala	Specialised			
KobrA	Specialised	Views	UML Inheritence, aggregation	Architecture per usage
SOFA	Specialised		Behaviour protocols	
PECOS	Specialised		Petri nets + scheduler	
FRACTAL	Specialised			

Figure 3.4: Important attributes of existing approaches

Chapter 4

An Independent Extra-functional Property Mechanism

The survey of the state-of-the-art has shown that a lot of approaches define how an extra-functional property may look. A structure of an EFP has been presented by means of specialised languages such as QML, CQML or HQML. Other approaches have instead addressed EFPs as part of component applications. Namely, there are component models such as Palladio, Robocop, ProCom, etc. Additionally, several approaches entail mainly evaluation and dependencies of extra-functional properties.

One of the drawbacks of all these approaches is their slow application and integration with existing industry frameworks. The proposed approach is more targeted at a direct integration with existing and widely used component frameworks. It consists of several modules, described later in this chapter, and generally named Extra-Functional Property Featured Compatibility Checks abbreviated as EFFCC. The EFFCC abbreviation will be used within the next sections to refer this approach.

4.1 The Type-Based Evaluation Approach

Current approaches vary from those verifying whole component compositions to those separately verifying each component connection (binding). Although the whole component assembly testing is widely used (e.g. in Palladio, Pecos, QML/CS), techniques determining compatibility on each component binding promise rapid and straightforward development process. The tests of the whole assemblies is the time and computational resources consuming task in which it is difficult to check every individual state of a complex system. On the other hand, the testing of each binding is a linear process with a fast computational time.

For instance, in Palladio example, system behaviour evaluation requires developers to prepare a considerable amount of behaviour models with parameters to be filled later depending on the concrete application of the components. The complex models are then evaluated to verify the whole component system. Type-based conformance [12, 19] does not target the behaviour of each component, which is in essence a detailing of the inner implementation, but it uses public interfaces as contracts between the components. The type-based conformance explicitly describes which behaviour a component guarantees on the provided interfaces once the assumptions on the required interfaces are fulfilled. Hence, the type-based conformance preserves the black-box nature while the behaviour based evaluation does not.

The main obstacle of the evaluation of the behaviour based on models is the computational complexity – the state space explosion problem. In contrary, the type-based evaluation has relatively low resource needs [12].

The mentioned reasons lead us to follow the type-based approach. It typically matches two components as compatible ones if interfaces are of the same types or sub-types. In this work, the compatibility of interfaces includes (i) the compatibility of method signatures: e.g. 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 [12] or [19].

The rationale behind taking EFPs into account is to improve compatibility checks of components. The basic idea is shown in Figure 4.1 in which the compatibility of a few components is examined. The figure shows the component A and three other components X, Y, Z expressing the question whether the interface of one of these three components is compatible with the interface of the component A. In this work, the compatibility of the components is based on the compatibility of their types supplemented by EFPs, expressed on the communicating elements.

The type-based approach is sufficient for the connections of components where only connected component interfaces need to be considered for the evaluation. However, EFPs need to take complex component graphs into account as long as the behaviour of one component may influence any other component in the graph. A problem of some presented languages is that they treat EFPs independently of each other. However, EFPs are typically influenced by EFPs connected through other components. Therefore, the approach presented in this work allows to express the graph dependencies of

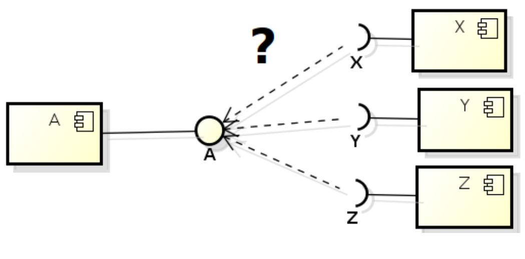


Figure 4.1: Type-based Binding Motivation

EFPs. A mechanism allowing to describe EFPs dependency using composing formulas will be introduced. Although it slightly limits the pure type-based approach, involving behaviour specification on components, it still provides this information stated on the communicating interface as part of the type.

This work aims at a mechanism consolidating discrepancy of existing approaches. For that reason, a lot of aspects of the solution were simply re-used from other approaches presented in this work. Other sections of this work describes EFFCC as a set of modules consolidating approaches of other works in a system aiming at a direct application to existing industrial component models.

4.2 The Mechanism Overview

Component-based development comprises several typical phases and activities which should be also covered by a comprehensive extra-functional property mechanism.

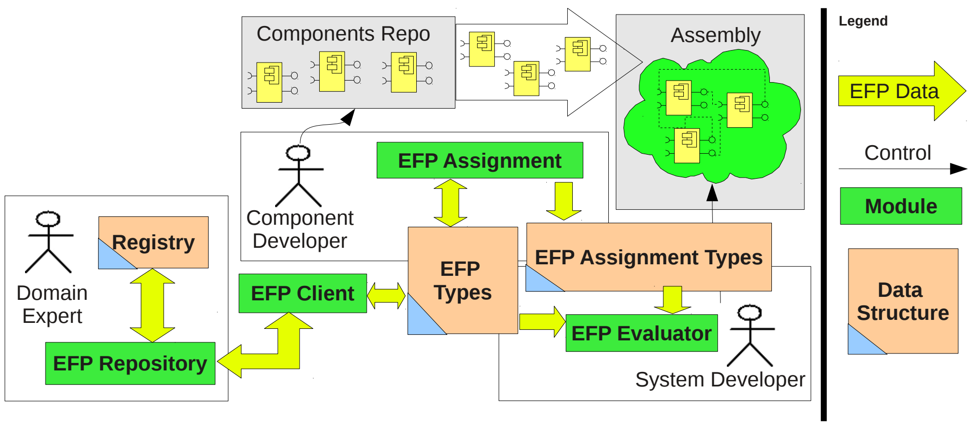


Figure 4.2: EFFCC Overview

The phases may be summarised in several steps. A domain expert or architect first designs the extra-functional properties to be used across a range of components and applications. In traditional component-based programming, this role comprises a domain expert developing components for a concrete domain. Afterwards a developer defines the concrete properties of components and estimates the ranges of their values. It is the same role as the role of a component developer in the traditional component-based programming. Finally, when the component software is being composed, an application assembler needs to verify the compatibility of components which should form the application. This role is equivalent to the software assembler in the traditional component-based programming.

Actually, EFFCC proposed here allows to manage all these phases. It allows to (1) declare EFPs for a domain of usage, (2) store their definitions and values in a repository concerning the domain, (3) assign EFPs to particular components by a component developer and (4) evaluate their compatibility by the system assembler.

Achieving the mentioned goals, EFFCC is a modularised system where the conceptual structure of EFFCC consists of four modules as depicted in Figure 4.2.

Domain definitions of EFPs are hold in a Repository. The Repository stores EFP definitions and is accessed by other modules to obtain, create or modify the properties. There is a domain expert who role is to fill in the repository defining existence of EFPs in a domain. We assume a per-domain repository, because there would be probably impossible to consolidate EFPs through all domains. A set of domain specific repositories e.g. domain of schools, automotive, libraries may exist.

The EFP Assignment module uses the Repository so it can attach the declared EFPs to each component. It covers a role of a component developer who prepares each component. He or she loads EFPs from the Repository and attaches them to components. As long as all component developers use properties from the same repository, they attach compatible properties. Hence, the role of the Repository is to consolidate understanding of the properties.

Once components are enriched with EFPs the EFP Evaluator takes care of comparing attached EFPs when verifying component compatibility during their binding process. It covers the role of a system assembler who composes a final system. He or she uses EFPs to determine behaviour of the final system in respect to functional and extra-functional properties. His or her expectations of the final system are expressed in terms of the properties from the repository and assigned to components. While the EFP Assignment module works with separate components, the EFP Evaluator covers a set of components which compose a final component application.

The interchange of extra-functional properties between the modules of EF-FCC requires shared understanding of EFP data. This is implemented by the module called EFP Types. It expresses EFPs, their relations to one another, their relation to a system of registries and the EFPs structure. Their detailed form will be described in Section 4.3. Furthermore, a relation and application of the EFP Types to particular components is implemented by the EFP Assignment Types data structure. It is an aggregation of the EFP Types covering the EFP data from the repository and meta-information of each component to connect EFP and the components.

In the following sections, we respectively describe the details of the structure of EFPs, the Repository, the structure of EFP Assignment Types and the EFP Assignment module itself. In addition, the means the EFP Assignment module achieves component framework linking will be presented. Finally, the process of EFP evaluation is detailed in a form of an algorithm.

4.3 An Extra-functional Property Structure

The fundamental part of the whole EFFCC are extra-functional properties, stored in Registry. The structure of extra-functional properties has been created following other approaches. Namely extra-functional languages such as NoFun [36], HQML [44], CQML+ [90] have been selected for their general description of extra-functional properties.

4.3.1 An Extra-functional Property Formalisation

Providing a detailed information of extra-functional properties, this section describes a structure of the properties in a precise formal manner. First of all, let us formalise extra-functional property itself:

$$E = \{e \mid e = (n, E_d, \gamma, V, M)\} \tag{4.1}$$

where

n is the name of a property,

$V \in T = T_c \cup T_s$ is a value type of a property,

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

$T_c = \{(T_1, \cdots, T_N) \mid N > 1, T_i \in T\}$ is a set of complex types containing non primitive values,

γ : $V \times V \to Z$; $Z = integer \cup \{$"n/d"$\}$ is a function which compares two instances $x, y \in V$ of the property with type T, stating which of two values is better. The meaning of the return values is:

Value	Meaning
$\{-\infty, \cdots, 0\}$	x is worse than y
0	x is equal to y
$\{0, \cdots, +\infty\}$	x is better than y
"n/d"	not-defined.

The function may not be explicitly defined for each type T 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 results in the same value. When an explicit rule does not exist and the comparison cannot be determined by the implicit rule, the value "n/d" is returned.

$E_d \subset E, e \notin E_d$ is a set containing other properties composing this EFP. This set is empty for simple properties while it contains deriving properties for a derived property,

M is a record containing any additional information meaningful in 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 replace the values.

The following example shows two simple extra-functional properties *time_to_process* and *data_transferred* of the integer data types measured in milliseconds and mega bytes respectively. They do not contain any explicit gamma functions and their values are assumed to be divided into groups of "low", "average" and "high" where the names denotes semantics of the underlying values. Implicit gamma function will be used for their comparing.

```
(time_to_process, default-gamma, integer,
  META {unit:‘‘ms’’, names: {low, average, high}})
(data_transferred, default-gamma, integer,
  META {unit:‘‘MB’’, names: {low, average, high}})
```

According to these two simple properties, a derived property *performance* may be defined. It has no measuring unit and its result is an item from an enumeration with values *poor*, *fine* and *good.*

```
(performance, {time_to_process, data_transferred}, default-gamma,
  enum {poor, fine, good}, META {})
```

4.3.2 An Extra-functional Property Meta-Model

In this part, the extra-functional property structure is presented as a meta-model shown in Figure 4.3. This model is basically a UML expression of the mathematical formalization.

Following Figure 4.3, `EFP` is an abstract class that represents a general extra-functional property. This class holds a name, a date type, comparing function γ and a block of meta-information mentioned as the set M in the formalisation. The γ function is modelled by the class `Gamma` while the M set is the class `Meta`. According to the design used in NoFun [36] the formalisation models simple and derived properties expressed in the meta-model by classes `SimpleEFP`, `DerivedEFP` for simple and derived properties respectively. According to the formalisation, the `DerivedEFP` adds an aggregation to other deriving properties. The form of the properties allows also to define deployment contracts [68], which express relations between components and a runtime environment, since the deployment contracts are also mapped to a name and a value of a certain data type.

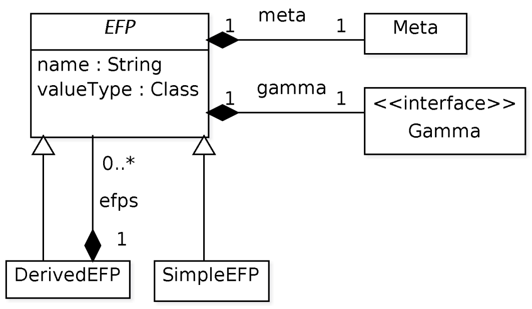


Figure 4.3: Extra-functional Property Meta-model

Figure 4.4 shows a hierarchy of data types that may be assigned to an EFP. Basic classes `EfpNumber`, `EfpBoolean`, `EfpString`, `EfpNumber` model standard data types known from programming languages such as Java or C++. Furthermore, additional data types were created to better suit EFPs. `EfpRatio` models percentages, `EfpSet` is a simple set of other types and `EfpComplexType` are pairs with names and data types. Let us also highlight the type `EfpNumberInterval` implementing interface `EfpInterval`. It allows to explicitly model generic intervals using the interface. Moreover, the

`EfpNumberInterval` implements an interval of numbers using instances of `EfpNumber`. Any other kinds of intervals are left for a future extension.

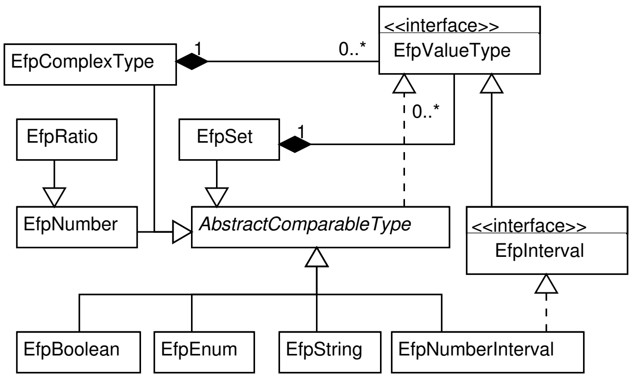


Figure 4.4: Data Types Meta-model

Comparing this hierarchy with other systems (e.g. data types in Java or C++), it adds a considerable improvement – all types implement one interface `EfpValueType`. Hence, all data types in the system belong to a unified hierarchy and the user may extend this model to add a new type by implementing one interface. The implementation of the interface forces the user to define a gamma function or use a default one and thus each data type in the hierarchy is comparable with another one.

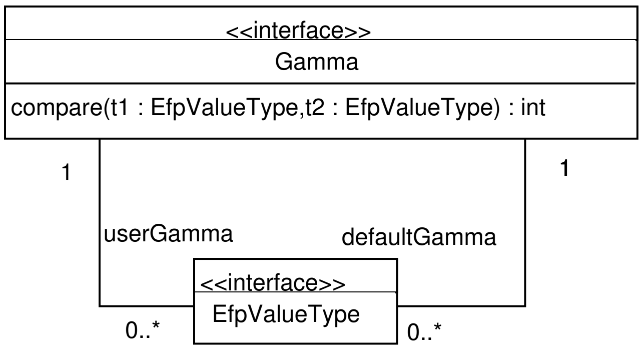


Figure 4.5: Comparing Function Meta-model

Figure 4.5 details the comparing function gamma. The function simply takes two instances of `EfpValueType` on input and compares them with each other. Important is the way the function gamma is assigned to a concrete data type. The interface `EfpValueType` contains one method that returns a default gamma function and one method that allows to inject a user defined gamma function. This mechanism guaranties that each instance of `EfpValueType` has a default comparing function or a specific user defined comparing function. Hence each instance in a system is comparable. The class `AbstractComparableType` from Figure 4.4 implements the above mentioned algorithms.

4.4 A Universal Extra-Functional Repository

This work proposes a novel idea that there should be shared understanding of extra-functional properties among component vendors. It allows to work with comparable properties. Following the component-based programming with the components stored in repositories, we suggest to use a repository also for EFPs. Apart from the use of standards such as the CQM [6], such understanding can be helped by a technical infrastructure which comprises a general repository containing all available properties in the domain. It allows to assume that properties are defined before a component is created and vendors can therefore use them to attach properties with the same meaning to different components.

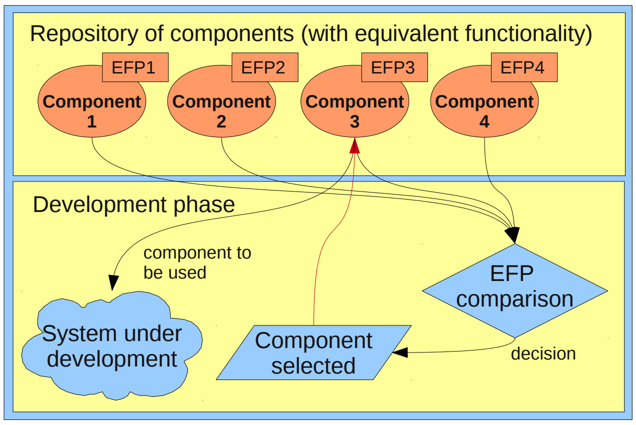


Figure 4.6: EFP-based Component Selection

Let us demonstrate this idea in Figure 4.6. There, it is assumed that different vendors provided components with the same functionality. The components differ only in their extra-functional properties. Hence, the compatibility decision is based on the properties. If each vendor used the properties developed on his own, the compatibility decision would not be meaningful. The properties would likely hold incompatible values, measuring units or differ in their semantics. For instance, a property *speed* may have different measuring units as *km/h* or *mph* or it may even differ semantically if one property denotes internet connection speed measured in *Mb/s*. Preventing this discrepancy, the repository consolidates meaning of EFPs. Once all vendors use EFPs from one repository, the possibility of misinterpreted properties rapidly decreases. These all lead to more meaningful and accurate compatibility decisions.

Consequently, the repository should (1) store unified properties, (2) encapsulate context-dependent values for each context (that may be also called a sub-domain), (3) improve understanding of values to reach all the mentioned requirements.

We invented a repository fulfilling all these requirements as a layered storage, named Registry. Its core idea is shown in Figure 4.7.

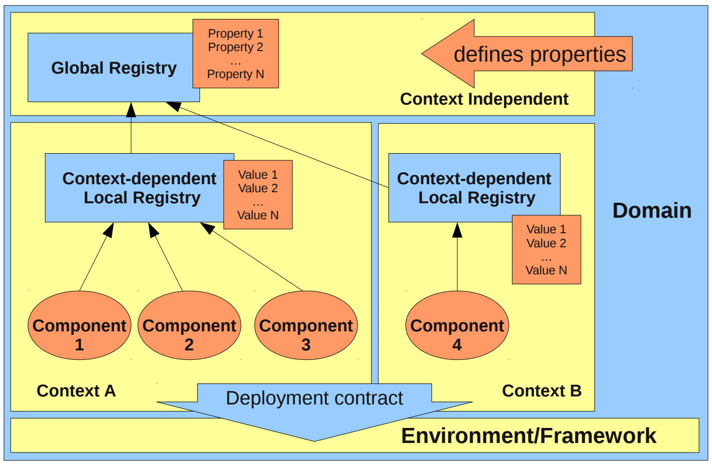


Figure 4.7: A Relation of Registries and Components in Contexts and a Domain

Component based programming always deals with a domain for which the components are developed and a set of particular contexts in which the components may run. The domain is a particular area with specific tasks and typical procedures to solve the task. Users involved in the domain have specific needs related to any operations with the tasks. Therefore, each domain most probably has its specific extra-functional properties.

Components developed for a particular domain solve the domain related tasks, however, a certain variability within this domain may be required. In other words, several contexts-of-usage may be found in one domain. Each context may have specific requirements where some of them are more important while some of them less. Therefore, a variability of extra-functional properties in terms of their ranges, scales and meanings should be taken into account.

A complex example covering the domain of Web Browsers with its different usage contexts is provided in Section 6 as a case-study.

Global registry (GR) is a store with definitions of EFP. Its usage concerns a domain where a domain expert defines existence of EFPs. The purpose is to state which properties are meaningful in a domain, but not to consider ranges of their values since they may differ for concrete usage. For that reason, GR defines the properties without their concrete extra-functional values. The properties are defined mainly in terms of their names and data types.

Local registry (LR) is concerned with a contextual meaning of EFPs. Its purpose is to attach scales and semantics to values for the properties from GR. As a result, each context of usage is defined by LR with the value definitions.

As it will be detailed in Section 4.4.1 the properties in GR and the values in LR are linked via names. In practice, it creates symbolic names for values giving them semantics and better understanding from a user point of view. In addition, the names split continuous values to several disjunctive intervals. The values from one interval may be understand as equivalent for the evaluation.

One advantage of this solution is that it encapsulates context-dependent values denoted by names. The names remain the same while concrete values differ. E.g. a value labelled as “small” gathers very different numbers for portable electronic devices and multi-core servers. However, a developer may think of the semantics (the meaning of “small” is obvious) rather than sorting out concrete values.

Another advantage of this solution is that continuous intervals of values are divided into disjunctive sets of named intervals, values or subsets. Hence, values in one interval may be treated as equivalent in the binding process (e.g. memory consumption in an interval $\{1, +\infty\}GB$ may be considered as “too high” for small portable devices. It does not mater whether the real value would be $1GB$, $1.2GB$ or $1.5GB$ – they all belong to one named group).

The deployment contract holds 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.

4.4.1 An Extra-functional Registry Structure

The same way as extra-functional properties structure, the repository is developed in a form of meta-model and formalisation described in this section.

An Extra-functional Registry Formalisation

In this section the structure of EFP registry is formalised providing a reader with a precise details of the structure of the registry. Global Registry is formally defined as a tuple:

$$GR = (id, name, E) \tag{4.2}$$

where:

id : Integer is the registry's unique identifier,

$name$: String is a human readable name of this GR,

E is a set of extra-functional properties (Section 4.3).

As it may be seen from the definition, Global Registry lists definitions of extra-functional properties from Equation 4.1. Although the EFPs defined in GR are defined simply by their names and data types with no explicit semantic model, we assume each GR exists for a limited domain in which semantics of the properties is implicitly known.

To complement the Global Registry definition, let us furthermore formalise Local Registry. It is a tuple:

$$LR = (id, GR, name, LR_A, S, D) \tag{4.3}$$

where:

id : Integer is the registry's unique identifier,

GR is Global Registry this LR is linked to,

$name$: String is a human readable repository name,

LR_A $= \{LR_1, LR_2, \cdots, LR_{N-1}\}$ is a set of other LRs (excluding the current one) integrated in this LR. It allows to aggregate LRs in hierarchies. The semantics is that a value from an aggregated LR is inherited unless this LR overrides the value. All possible value clashes caused by the aggregation must be resolved by overriding the impacted values. N is a total number of all LRs valid in actual GR,

$S = \{(e, name, v) \mid e \in E \wedge name \in String \wedge v \in V\}$ is a set defining context dependent values for simple properties,

- e is a property from GR,
- $name$ is an assigned name of the value which must be selected from the list of available names given in the M part of the definition of the property in GR. Hence $name \in names \in M \in e$,
- v is an assigned value from a value type defined for a respective property $v \in V \in e$,

$D = \{(e, f) \mid e \in E \wedge f : E_d^n \times S^m \rightarrow V\}$ is a set of derived property definitions, where each derived property e is governed by the function f:

- e is a derived property with a non-empty $(E_d \in e) \neq \emptyset$ set of deriving properties,
- f is a function which takes deriving EFPs from the E_d set on its input. The dimension is $n \leq |E_d|$ meaning that all or only part of the properties from E_d can be referred in the formula f. The set of simple context-dependent values defined in this LR may also go on the input. The dimension of the number of these values is limited by a number of deriving properties and their simple context-dependent value definitions in this LR: $m \leq |E_d| * |\{S' \in S \mid (name \in S') = (name \in M \in e \in E_d)\}|$.

  An output is a value from the e property value type. Notice that the input of the function is abstract representation of an EFP, because a concrete value of the property is not known in the time the deriving rule is established in the registry. A concrete value of a property will be known later when the property is applied on a component. Therefore, a value is obtained by a function:

$$\nu : E \rightarrow V \tag{4.4}$$

  in the evaluation process when concrete values of all properties are known. A value from the S set may be obtained directly because it comes from the current LR. Taking it together a final value is computed by the composition of the functions: $f \circ (S \rightarrow V) \circ \nu : V^n \rightarrow V$ transforming all values of the deriving properties to a result value of the derived property. Hence, a list of deriving properties determines a value of the derived property with respect to the f and ν functions. While the function f is defined on the EFP registry level, the ν function is a matter of the evaluation process implemented by another module. The dimension is set to $n \leq |T|$ because the maximal number of value types used in all

E_d properties may be used. It also does not exceed the number of currently existing types.

For instance, a logical formula $f(secure) : true \Leftrightarrow protocol = HTTPS$ may be assigned to the derived property *secure* where *protocol* is a simple property with an enumerated type containing items $[HTTP, HTTPS]$. Obviously, a value of the *protocol* property is not known in the time the formula f is defined in the registry. Once the properties are applied on a component, the function ν will return an assigned value. For this particular example, the formula f is satisfied if and only if the *security* property has a value $HTTPS$ assigned on a component.

The range and semantics of a property in GR is in the formalisation defined only by its name, data-type and a measuring unit. Whereas Yap [102] attempts in his work to solve discrepancy of different measurement and scales of values on the basis of transforming rules, technically defined in a XML file, we attempt to solve the discrepancy limiting EFPs valid for a domain with the same measuring unit for one GR and the scales divided into sub-domains by LRs. As a result, it is assumed the properties and their values are comparable in a concrete context-of-usage.

An Extra-functional Registry Example

For example, let us consider the properties from Section 4.3. Those definitions will be stored in a Global Registry. Then, a Local Registry for smart phones with GPRS-only connection may contain the following value definitions:

```
time_to_process: low = 10, high = 5000, ...
data_transferred: low = 1, high = 100, ...
```

while a Local Registry for desktop computers may require faster connections with values:

```
time_to_process: low = 100, high = 50000, ...
data_transferred: low = 10, high = 1000, ...
```

This solution distributes different scales of the values among disjunctive LRs. Each LR limits values for a sub-domain in which the values are meaningful and have the correct scales.

An Extra-functional Registry Meta-model

Providing another overview of the EFP registry, a meta-model has been implemented according to the formalisations. The meta-model is shown in Figure 4.8.

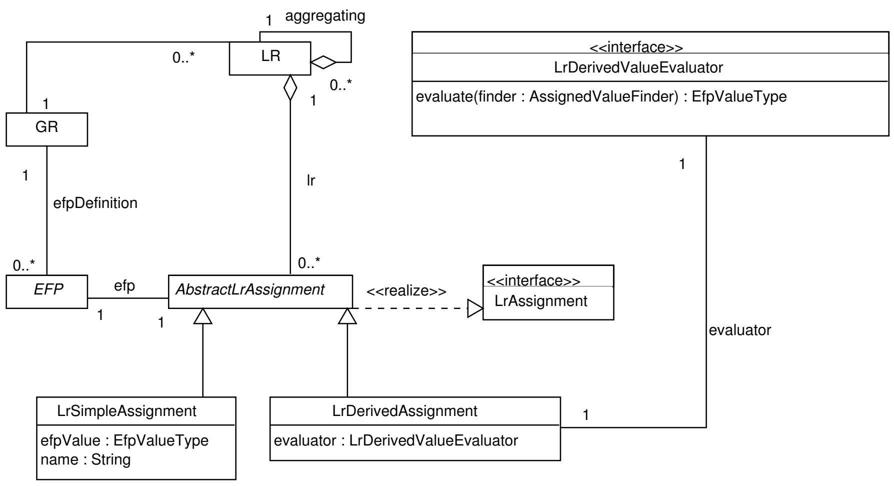


Figure 4.8: Registry Meta-model

Only a brief description of each element is provided here because details of all elements may be found in the formal Section 4.4.1.

The class `GR` represents Global registry that is a storage holding all EFP definitions. GR collects instances of classes `EFP` from Figure 4.3.

The class `LR` represents Local registry that is a per-environment storage of EFP values also mentioned in the formal block. Each LR may have a link to its parent LR that it extends. Each LR has a link to its GR for which EFP values are defined. In Local registry, values are assigned via an instance of the class `LrAssignment` with two implementations for simple and derived properties. Its purpose is to put an EFP, LR and a value into one relation.

The two assignment implementations represent classes `LrSimpleAssignment` and `LrDerivedAssignment` respecting the sets S and D from the formal definitions.

According to the definition of the S and D sets, the class `LrSimpleAssignment` holds a value for the simple EFPs, the S set. The value is of the same type as the respective simple EFP value type. In addition, this class holds a name for this value creating a context dependent named value. This means that an EFP, a value and a name for the value is stored in this simple assignment which in practice corresponds to the formal definitions.

Working with number intervals in simple assignments, a data type modelled as `EfpNumberInterval` from Figure 4.5 is typically used. For instance, the *memory consumption* property is modelled by `EfpNumberInterval` and split to three sub-intervals with names: *small*, *average*, *high* defined for three instances of `LrSimpleAssignment`. In practice, this interval is divided into three named sub-intervals where a user may think of the semantics using the names while an evaluation process works with the numbers encapsulated by the names.

Let us note that, for evaluation of mathematical formulas with number intervals, we use an interval computing with computation rules defined in [74].

The other implementation of `AbstractLrAssignment`, the class `LrDerivedAssignment`, represents an assignment of a derived value for a derived EFP which is the model for the D set from the formal definition. Its purpose is to hold a deriving formula for a derived EFP. Since the requirement is to have a possibility to implement a variety of formulas, an abstract interface `LrDerivedValueEvaluator` exists. The implementation of this interface represents any kinds of formulas. For instance, mathematical or logical formulas are representative examples of possible implementations and they have been actually implemented in the prototype. In practice, any implementing instances represent the function f defined in the formal block. The way the function is written and evaluated is up to a concrete implementation. A particular instance must only implement the function `evaluate` shown in the model. As it may be seen, the method `evaluate` uses another interface `AssignedValueFinder` on its input. It is a call-back interface in which implementing classes represent the function ν (Equation 4.4) also mentioned in the formal block.

The function ν has a simple definition in the `AssignedValueFinder` interface

```
public interface AssignedValueFinder {
    EfpValueType findValue(EFP efp);
}
```

respecting its formal definition. An implementing class must assure the value for a requested EFP will be returned from a particular component which the evaluation is made for. Then, the implementation of `LrDerivedValueEvaluator` may compute a result. The meta-model for the registry contains only this interface while concrete implementation is made by the evaluator module that has an access to values assigned to concrete components.

4.5 Application of the Approach to a Variety of Systems

The previous section has described the repository as a stand-alone data storage of EFPs. At this point, another part of EFFCC will be described. It is a module managing application of EFPs to particular components.

Since EFFCC aims at applicability to a wide spectrum of component models, this module contains a set of specialised means to reach this requirements. Practically, this module uses two separate sub-modules shown in Figure 4.9 with the module itself called EFP Assignment.

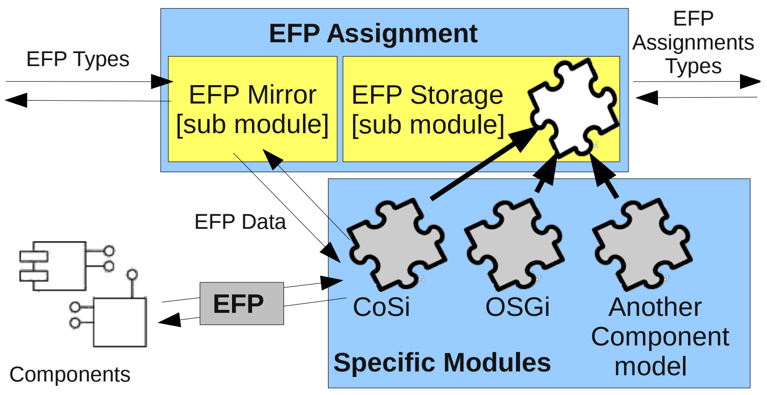


Figure 4.9: EFP Assignment Module

The EFP Mirror sub-module shown in Figure 4.9 represents an independent part of the EFP Assignment module responsible for mirroring EFP data between Registry and components. The reason is that in the phase of attaching EFPs to a component, a developer loads EFPs from the remote EFP repository and applies them to the component. It would be impractical to call the EFP repository every time the data are needed later on. As a result, the EFP Mirror sub-module stores complete information of the attached EFPs together with the component.

The benefit of this approach is that it creates a general mechanism usable for all supported component models. The detailed structure of EFP data is hidden from (or at least irrelevant to) the “plain” component framework and this sub-module provides an interface transparently accessing EFPs in a component model independent format. A small drawback is that a considerable amount of extra information may potentially need to be stored together with the component in case the EFPs are numerous or their structure is deep.

The EFP Storage sub-module from Figure 4.9 provides an extension point where implementations for supported component models are plugged. This sub-module brings the desired flexibility and applicability for different component models in a form of lightweight plug-ins. Obviously, components look different in different component models. For that reason, each implementation decides (1) the location where and how to store EFP data, (2) how to link the data with concrete features of the component.

The assignment of EFPs to components actually consists of two phases of EFP manipulation. In the first phase a developer attaches EFPs to components, loading the properties from the repository mirroring them on the component. In the second phase, the EFP Assignment module provides the previously attached EFPs to other systems (see Figure 4.9) loading them from the mirror. Hence, this module connects extra-functional repository from Section 4.4.1 and an evaluator described later in Section 4.6. The form of transferred data are so called EFP Assignment Types introduced in Section 4.5.1 below.

Two typical scenarios may be found. In the first one, developers integrate EFP Assignment module into their application which allows to attach EFPs to components in the phase of component development. In the other one, developers integrate EFP Assignment module into application evaluating components compatibility (e.g. component framework or tools evaluating components) to enrich the process of the evaluation with extra-functional properties.

The typical usage of this module is the cooperation together with extra-functional repository from which it loads data. However, this module may be also used independently of the repository where the only part attaching and loading EFPs from/to components is used. It is useful, for instance, in situations in which existing component frameworks do not need to rely on the common EFP storage.

4.5.1 EFP Assignment Types

The purpose of this section is to introduce a mechanism of transferring data between components and the evaluator.

Since the EFFCC aims at generality, EFP data assigned to components must cover a wide spectrum of component models. For that reason, EFP Assignment Types is the generic representation of EFPs attached to components. It aggregates EFP Types and the information about assignments of EFP values to components. The corresponding sub-module in EFFCC is able to serve this data to its other parts; in particular, modules working with EFP data on components receive the data from the EFP Assignment module via

EFP Assignment Types while the data from EFP repository are transferred in a form of EFP Types.

An EFP Assignment Formalisation

Providing a detailed information of a form of the EFP assignments, this section lists a formal approach to the structure and application of EFP to components in any component model. Let us also highlight that despite the fact this work targets component and component programming, the assignment formalisation is also usable in other software elements. So, it may be as well used for services in a Service Oriented Architecture.

First of all, let us define EFP Assignment Types that are formally defined as a set:

$$AT = F \times E \times (F \times E_d \times V_A)^n \times V_A \tag{4.5}$$

where F is a set of all generic representations of component features. Component models use a variety of communicating means such as *Interface*, *Event*, *Port*, *Component*. All these elements serve as communicating interfaces the compatibility of which should be checked when components are connected in the binding process. Since there is not a finite set of such elements among component models, the feature is capable of holding any such element in a generic manner. The set of features is defined as:

$$F = \{f \mid f = (name, type, role, mandatory)\} \tag{4.6}$$

with the following meaning of the tuple elements:

$name$: String is the name of the feature.

$role \in \{$“*required*“, “*provided*“$\}$ expressing if the feature is put on the required or the provided side of a component respectively.

$mandatory$: Boolean determines whether this feature is required to be bound to another feature in the binding process. A missing mandatory feature in the binding process should violate an error while a non-mandatory one should be skipped.

$type$ is the meta-type of the feature including a name (for instance, “interface”, “package”, “component”), parameters (for instance, inputs and outputs of methods). For instance, a component publishing its interfaces provides a set of features with the type name *Interface*.

Very often, the type is extended with a version of the feature where $version = (s_1 ... s_N), N > 1$ and s_i is a scalar expressing part of a full feature version information. Typical form of the version is a tuple concerning major, minor and micro change in interfaces together with a release note. E.g. a version may read $1.0.1.RC1$.

Splitting the set of features F into two disjunctive sets $F = F_p \cup F_r$ denoting provided and required features respectively, a function matching the features may be defined as:

$$\mu : F_p \times F_r \rightarrow Boolean \tag{4.7}$$

The purpose of this function is to determine which provided-required element pairs should be connected with each other. In other words, it says which provided and required elements are connected in a component binding process.

An implementation in our prototype defines the μ function as matching two features only if following rules hold:

- Names are equal for both features,
- a provided feature matches only with a required one or vice versa,
- a mandatory required feature must have a provided counterpart,
- if the type compatibility is explicitly expressed as versions, then the version on the provided side is equal or greater than the version on the required side or vice versa.

However, a more sophisticated matching μ function can be provided. For example, compatibility on interfaces using subtype relation [19, 12] would reach more accurate results. In a nutshell, a required feature must be a sub-type of the provided feature. It is reached by examining each interface finding out whether their method headers match. Since instances of the features come from the EFP Assignment module, the extension is straightforward: the re-implementation of a component-specific sub-module in the assignment module provides different μ function while other algorithms remain unchanged.

Moreover, in a lot of situation the μ function will be defined separately from EFFCC. Usually, component frameworks have their own means to bind components. As long as EFFCC aims at applicability to existing frameworks, the ideal approach is to let the matching function be implemented by the host component framework. For instance, OSGi binding process is quite complicated. However, a set of listeners and resolver hooks allows to observe the binding process without the need of understanding or even re-implementing it.

Another subset of the AT set is $(F \times E_d \times V_A)$ that denotes the assignment of deriving properties. This set is a non-empty set if the property $e \in E$

is a derived one. Once a derived property is assigned to a component, its deriving properties must be also assigned to the component. Hence, the dimension n is $n = |E_d|$ meaning that all deriving properties must be also assigned to the component providing information to evaluate the derived property in the evaluation process.

As long as the information of the assignments of the deriving properties is stored, the function ν (Equation 4.4) may evaluate a concrete value assigned to each property leading to the evaluation of each deriving rule.

Obviously, if a property is a simple one, the dimension $n = |E_d|$ is 0 and no deriving property is part of the assignment.

Continuing with the definition of EFP Assignment, E is a set of extra-functional properties from Section 4.3 and V_A is a set of values assigned to the properties which has a following form concerning several possible values:

$$V_A = V \cup S \cup D \cup C \tag{4.8}$$

Where $V \cap S \cap D \cap C = \emptyset$. For a value $v \in V_A$ then holds:

if $v \in V$ then the value represents a directly assigned value. Such a value is typically independent of a context of usage and remains constant independently of a runtime environment. In this case, V defines a value type of the respective e property as it was stated in its definition.

if $v \in S$ or $v \in D$ then the value is valid for a particular context of usage as defined by LR. The D and S sets are the same sets defined in the LR formalisation in Section 4.4.1. A component can thus contain multiple values of a given property for different contexts. Evaluating components, one must select the context in which a result should be computed for and the evaluator then uses only values valid for the selected context.

if $v \in C$ then it expresses a formula, declared directly at the component, determining a value of an EFP from other EFPs. This kind of value allows to compute EFPs on the provided side of components based on those on the required side. In other words, it determines how an output of a component is influenced by its inputs. This kind of value is the most flexible one allowing to model situations where component EFPs depend on other components bound in a component binding.

Crnkovic [29] overviews the complexity of composing EFPs that must take other properties, runtime and configuration into account. For that reason, the value $v \in C$ is supposed to hold simple as well as quite complex formulas and EFFCC is generic with possibility to implement comprehensive composing rules.

The C set is defined: $C = \{(f, e, h) \mid f \in F_{prov} \wedge e \in E \wedge h : F^n_{req} \times E^m \rightarrow V\}$ meaning that each feature and EFP assigned to this feature has assigned a formula taking a value from a required feature and its EFP and computing a resulting value. As it may be seen, the formula is assigned to a provided feature and computing its result using a required feature. This naturally follows the basic idea of computing a provided side of a component from its required side.

The dimension n does not exceed the number of required features actually existing on a component $n \leq |F_{req}|$ while the dimension m does not exceed a number of EFPs defined in current GR $m \leq |\{e \in E \mid E \in GR\}|$. This means that tuples for all required features and all existing EFPs may be used in a formula to compute a value.

Notice that the usage of these formulas is similar to the usage of the deriving formulas in Section 4.4.1. These formulas also work with abstract EFPs instead of their concrete instances. In addition, the formulas use features on the component required side, because provided features on connected components are not know when the formulas are defined. Therefore, the values are to be known in the time the component is connected in a graph with other components.

According to rationale introduced for the deriving formulas, the same way a function is defined for the assignment. Its purpose is to obtain the values as they are available in the evaluation process:

$$\nu_A : F_{req} \times E \rightarrow V \tag{4.9}$$

Despite its simplicity, the ν_A function is crucial when the components are bound. Its composition with the assignment function h allows to evaluate a value: $h \circ \nu_A : V^m \rightarrow V$. While the h function is directly applied on an assignment, the function ν_A comes from another module responsible for the connecting and evaluating components in a graph. Hence, the result is computed as soon as the components are bound. In practice, equivalent mechanism is used for derived properties. The dimension is $m \leq |T|$ because the maximal number of value types used in all EFPs may be used.

For instance, a component may declare its speed-up by the Amdahl's law, defined as a $h(s, a)$ function with an EFP s applied to a feature a: $h(s, a) = \frac{1}{(1-P)+\frac{P}{s_a}}$. P expresses the amount of a code which may be paralleled and it is constant for a particular component (e.g. 30%). s is a number of processors depending on a runtime environment. Hence, the component claims its speed-up based on the input parameter, which is a number of processors, from the runtime. The writing s_a is used for marking that a property s is assigned to a feature

a. Therefore the EFP s is evaluated by the ν_A function once a component is bound to a particular runtime and its value is known. Let say the component is deployed to a system with a dual core processor. Then, the formula is evaluated as $h(s,a) = \frac{1}{(1-0.3)+\frac{0.3}{\nu_A(s_a)}} = \frac{1}{(1-0.3)+\frac{0.3}{2}}$

An EFP Assignment Example

Finally, let us demonstrate the assignment mechanism on an example. It shows the EFP from Section 4.3 attached to a feature using several different values:

```
(  # feature
 ("DataAccess", "interface", "provided", true,
   "matched-by-name"),
   # EFP
 (time_to_process, ... ),
   # values
 (LR.1::low, LR.2::average, direct::20,
    math::(2 * DataAccess::data_transferred) )
)
```

In the example, `matched-by-name` denotes a function which matches two features with the same names. Its definition would be defined using means of concrete implementation language. For instance, it may be one method in Java. `LR.1` and `LR.2` denotes two local registries identified by their IDs. These IDs are loaded from the EFP repository. `direct` is a direct value – 20ms in this example. The meaning is that *time_to_process* would have attached this value for all contexts. `math` defines a math formula. It says the *time_to_process* depends on *data_transferred*.

Let us note that in a typical usage only one type of a value (either LR, direct or formula one) is defined in an assignment. It is the purpose of this example to show all the possibilities.

An EFP Assignment Meta-Model

This section summarises EFP Assignment Types in a form of a meta-model. The meta-model is shown in Figure 4.10 where the class `EfpAssignment` holds each assignment of extra-functional properties (class `EFP`) to features (interface `Feature`) assigning values that are modelled as the `EfpAssignedValue` interface.

The `Feature` interface denotes an abstraction of component communicating counterparts used in a process of component binding. An initial abstract implementation `AbstractFeature` exists to have; the most typical attributes of a lot of component models. Namely, it holds (1) a name, (2) information of a required or provided role, (3) a version, (4) a type (it covers e.g. *Interface*, *Event*, *Port*, *Component*), (5) a link to a parent feature (it is used for grouping features to its parents. For instance, if two features representing an *interface* have the same names but they are in different packages, they essentially represent different features. The link to parents distinguishes them), (6) the mandatory attribute saying whether a feature must be present in a binding processes. These attributes practically follow a definition from the formal section. Since a lot of component models may use extended information of their features, it is generally expected that specialised subclasses will be created.

Another element of the model is a value assigned to the feature and EFP represented by the interface `EfpAssignedValue`. The assigned value has three implementations made via sub-classing the convenient abstract class `AbstractEfpAssignedValue`. The sub-classes represent three kinds of values for the direct, context-dependent and formula-computed values.

According to a formal definition, the class `EfpDirectValue` serves for context independent values. For that reason, it only contains a `value` attribute holding actually assigned value modelled as `EfpValueType` from Figure 4.8.

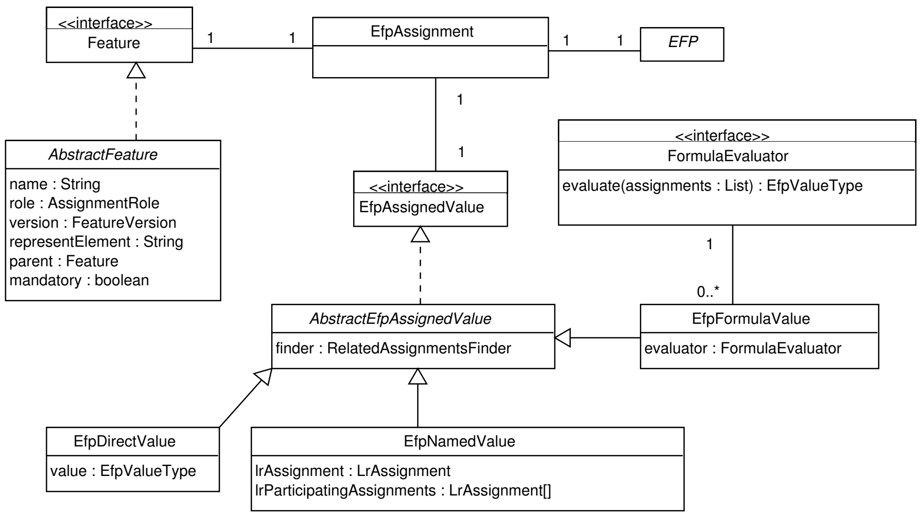


Figure 4.10: EFP Assignment Type Meta-Model

Another class, `EfpNamedValue`, holds a context independent value with a `lrAssignment` attribute that actually holds a concrete LR-based value. The class `LrAssignment` comes from the model in Figure 4.8. The attribute `lrParticipatingAssignments` is used only for derived EFPs where it contains any LR-based values used in a deriving formula for respective EFPs.

The last and most comprehensive value described in the formal approach is modelled by the class `EfpFormulaValue` allowing to use complex formulas computing EFPs from a set of dependent EFPs. The only attribute of this kind of value is a reference to an evaluator modelled by the interface `FormulaEvaluator` that takes a list of `EfpAssignment`s on its input and computes an EFP value `EfpValueType`. How a formula is written and computed it is up to a particular implementing class, however, prototype implementations for mathematical and logical formula have been created. According to the formalisation, `FormulaEvaluator` models the function h mentioned above.

In addition, as it has been mentioned in the formal approach, the function h cooperates with the ν_A function to call-back values from other components connected in a binding of components. This function is implemented by the interface `RelatedAssignmentsFinder` also shown in the model. The call-back method looks as follows:

```
public interface RelatedAssignmentsFinder {
  EfpAssignedValue getConnectedAssignments(
     Feature requiredFeature,
     EFP requiredEfp);
}
```

This method takes a feature and an EFP on its input and loads a value assigned on a connected component. The connected components are connected via their provided-required element pairs. Since the assignment module has no information of connected components as long as it works only with isolated components, a particular implementation of this interface is made by another module (see Section 4.6) that works with all components connected in a binding graph.

As a result, the composition of functions h and ν_A is modelled as the call of methods from `RelatedAssignmentsFinder` and `FormulaEvaluator` interfaces.

Notice that `RelatedAssignmentsFinder` is modelled on the level of `AbstractAssignedValue`. Despite the fact that this interface is used only by the `EfpFormulaValue`, it is actually put on the more abstract level. The rationale is in a future extensibility where any other type of a value may be added and be able to load values from connected components.

4.6 An EFP Evaluation Process

This section complements the description of EFFCC introducing the last module which is the evaluator of extra-functional properties.

The main purpose of the EFP Evaluator is to load a set of components and verify their compatibility in terms of extra-functional properties. The module first calls the EFP Assignment for each component to obtain EFPs. The received data are then composed to a graph representing components and their bindings, which serves the evaluator to find problems in component compatibility. Shortly, binding problems have forms of missing edges in the graph while EFP incompatibilities show as mismatches on respective edges. Sections 4.6.1 and 4.6.2 provide more details.

Unlike other modules, the EFP Evaluator working with EFP Types is not customizable for multiple component models. Instead it works on a generic model of EFPs and component application architecture. The variability of features and forms in component models is addressed by the EFP Assignment module that is a customisation point of EFFCC. Once a user desires to apply the evaluation process to another component model, he or she must create a plug-in to the EFP Assignment module while the evaluating process of the Evaluator remains unchanged.

The mechanism of component evaluation uses the EFP Assignment Types from Section 4.5.1 and works in two steps: (1) the graph of components is created by means of binding the components by their provided and required counterpart elements (in the model denoted as *Features*) and (2) the evaluator checks EFPs attached to the connected elements. It creates a sequence in which the results of comparing are stored for all connected required and provided elements. Therefore, an easy verification of compatible or incompatible connected pairs is allowed by checking results in the respective items in the sequence.

Let us point out that the goal of the mechanism is not to prescribe which steps should be taken for incompatible components. It is the decision of a concrete application whether an incompatible result is e.g. written to a log, prevents a component framework to start, sends warning message to an administrator inbox etc.

4.6.1 The Structure of the EFP Graph

Once the EFP Evaluator obtains a set of EFP Assignment Types, it can compose a graph representing the application structure annotated with properties. The graph contains components to be evaluated in which the components, component features and extra-functional properties create vertexes while edges represent their binding.

The rationale behind composing components into graph is that it allows to evaluate not only isolated EFPs but it also allows to evaluate EFPs connected with EFPs on other components. Hence, the evaluation is capable of providing more accurate results considering influences of EFPs with each other.

Figure 4.11 shows an example graph generated by the algorithm in which different shapes denote different kinds of vertexes.

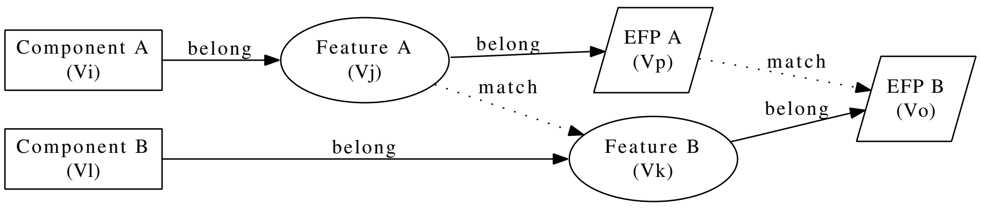


Figure 4.11: Example Graph

Before the evaluation algorithm will be detailed, let us formalise the structure of the graph created by the evaluator. The evaluator generates an oriented graph $\overrightarrow{G} = (V, E)$ where V is a set of vertexes and E is a set of edges. In addition to the well know version of the oriented graph, the algorithm used here introduces an extended definition based on specialized types of vertexes and edges:

$$\begin{aligned} V(\overrightarrow{G}) &= V_{component}(\overrightarrow{G}) \cup V_{feature}(\overrightarrow{G}) \cup V_{efp}(\overrightarrow{G}) \\ E(\overrightarrow{G}) &= E_{belong}(\overrightarrow{G}) \cup E_{match}(\overrightarrow{G}) \end{aligned} \tag{4.10}$$

Three types of vertexes denote a component, a component feature and an extra-functional property which compose the extra-functional graph. More formally, the following rules hold for each vertex v:

if $v \in V_{component}(\overrightarrow{G})$ then the vertex v represents a component,

if $v \in V_{feature}(\overrightarrow{G})$ then the vertex v represents a feature,

if $v \in V_{efp}(\overrightarrow{G})$ then the vertex v represents an EFP.

Furthermore, there are two types of edges. In the graph, different types of vertexes are connected by different type of edges.

The first kind of edges represents the relation of components, component features and extra-functional properties of one component. These edges express: (1) extra-functional properties attached to features of a component and (2) features belonging to the component. The direction of the edges denotes provided or required elements.

If a component has a set of features which have attached extra-functional properties, they will appear as a cloud of vertexes connected by edges in the graph.

Formally denoted, the following rules hold for each edge e:

$$e \in E_{belong}(\overrightarrow{G}) : \begin{cases} (v_x, v_y) \mid v_x \in V_{component}(\overrightarrow{G}) \wedge v_y \in V_{feature}(\overrightarrow{G}) \\ \;:\text{ required feature,} \\ (v_x, v_y) \mid v_x \in V_{feature}(\overrightarrow{G}) \wedge v_y \in V_{component}(\overrightarrow{G}) \\ \;:\text{ provided feature,} \\ (v_x, v_y) \mid v_x \in V_{feature}(\overrightarrow{G}) \wedge v_y \in V_{efp}(\overrightarrow{G}) \\ \;:\text{ required EFP,} \\ (v_x, v_y) \mid v_x \in V_{efp}(\overrightarrow{G}) \wedge v_y \in V_{feature}(\overrightarrow{G}) \\ \;:\text{ provided EFP.} \end{cases}$$

The vertexes connected by the E_{belong} edges create isolated clusters of components first.

The binding of the features and the extra-functional properties is based on matching all provided and required feature pairs with one another. The extra-functional properties are matched only with bound features. While features are bound by the matching function μ (Equation 4.7 from Section 4.5.1), EFPs are matched via their names and their relation to a feature. It means that one EFP may be attached to multiple features, but only once to the same feature.

$$e \in E_{match}(\overrightarrow{G}) : \begin{cases} (v_x, v_y) \mid v_x \in V_{feature}(\overrightarrow{G}) \wedge v_y \in V_{feature}(\overrightarrow{G}) \\ \;:\text{ binding features,} \\ (v_x, v_y) \mid v_x \in V_{efp}(\overrightarrow{G}) \wedge v_y \in V_{efp}(\overrightarrow{G}) \\ \;:\text{ matching EFPs.} \end{cases}$$

Using this model, the EFP Evaluator generates the graph in several steps. It first creates component vertexes ($V_{component}(\overrightarrow{G})$) from a set of components a user desires to evaluate. Secondly, the EFP Assignment Types are loaded for each component. The vertexes for features ($V_{feature}(\overrightarrow{G})$) and EFPs ($V_{efp}(\overrightarrow{G})$) are added and the vertexes are connected using the belonging edges ($E_{belong}(\overrightarrow{G})$) to express which features and EFPs are attached to the components. This way complete representation of one component is created.

The attempt to compare this graph with approaches used in industrial Spring and OSGi has been made. However, both frameworks provide vague documentations of their implementation. Whereas OSGi provides at least some examples in its documentation [84], Spring describes only its dependency injection process. No hint of an internal structure is provided. Despite that, this work assumes that the $V_{component}$ and $V_{feature}$ vertexes and their connecting edges are sufficient to model the component binding. It should cover a lot of practical applications built on the concept of having components and means to express their dependencies. In addition, the V_{efp} vertex allows to enrich the dependencies of EFPs. Typically, a subgraph consisting of $V_{component}$ and $V_{feature}$ vertexes follows the structure of the respective component model binding while the V_{efp} vertex is additional information.

4.6.2 Evaluation of EFPs

Having graph representation of component connections, the evaluation is quite straightforward. The evaluator must go through the graph and find possible problems in vertex connections first, then it uses the values attached to EFPs to compare the value pairs of two connected EFPs.

Two components are incompatible in the following situations: (1) a required feature or extra-functional property is not connected with a provided counterpart or (2) values on connected EFPs are not compatible – it means the function γ from Section 4.3 results in incompatible values.

Examination of the EFP Graph

The algorithm which computes the values of attached EFPs as well as checks the connection of components with one another uses a modified depth-first-search graph algorithm. It has the following steps (let us use a notation v_x, e_{xy} where $x, y \in I$ and I is a finite index set for indexing vertexes and edges respectively). The indexing of the vertexes used in the algorithm is also shown in Figure 4.11 to help the reader to understand the algorithm properly:

```
1. $V_{component}(\overrightarrow{G})$, $V_{feature}(\overrightarrow{G})$ and $V_{efp}(\overrightarrow{G})$ created, $prev := []$, $remain := V_{component}(\overrightarrow{G})$, $v_i := remain[0]$

2. $remain := remain - \{v_i\}$
   for $v_j$ in $V_{feature}(\overrightarrow{G})$ {
      if $(v_i, v_j) \in E(\overrightarrow{G})$ goto 3
   }
   goto 5

3. $v_k := null$
   for $v_k$ in $V_{feature}(\overrightarrow{G})$ {
      if $(v_j, v_k) \in E(\overrightarrow{G})$ goto 4
    }
    if  $v_k == null$ and mandatory $v_j$ ERROR
    if  $v_k == null$ and not mandatory $v_j$ goto 2

4.  for $v_l$ in $V_{component}(\overrightarrow{G})$ {
      if $(v_k, v_l) \in E(\overrightarrow{G})$ {
        $prev$ := $\{v_i\} \cup prev$
        $v_i$ := $v_l$
        goto 2
      }
    }

5.  // select EFPs connected to features
    $V_{efp_o}$ := $\{v_o \in V_{efp}(\overrightarrow{G}) \mid \forall o \exists k : e_{ok} \in E(\overrightarrow{G}) \wedge v_k \in V_{feature}(\overrightarrow{G})\}$
    $V_{efp_p}$ := $\{v_p \in V_{efp}(\overrightarrow{G}) \mid \forall p \exists j : e_{jp} \in E(\overrightarrow{G}) \wedge v_j \in V_{feature}(\overrightarrow{G})\}$

    // check all EFPs are connected by edges
    if $\forall p \exists o : e_{po} \in E(\overrightarrow{G}) \wedge v_o \in V_{efp_o}(\overrightarrow{G}) \wedge v_p \in V_{efp_p}(\overrightarrow{G})$ {
      // evaluate EFPs on connected edges
      evaluate $\{e_{po} \in E(\overrightarrow{G}) \mid \forall p \exists o : v_p \in V_{efp_p}(\overrightarrow{G}) \wedge v_o \in V_{efp_o}(\overrightarrow{G})\}$
      $V_{feature}$ := $V_{feature} - \{v_j, v_k\}$
      goto 6
    } else ERROR

6.  if size $prev == 0$  {
      if size $remain == 0$  END
        else $v_i$ := $remain[0]$
    else {
      $v_i$ := $prev[0]$
      $prev$ := $prev - \{prev[0]\}$
    }
    goto 2
```

The algorithm verifies any inconsistency in the graph in terms of component bindings: (1) all required mandatory features that are not connected to any provided features, (2) all required EFPs that are not connected to any provided EFPs.

A decision whether EFPs on connected vertexes are compatible is in the algorithm denoted by the function named `evaluate` and detailed in the following section.

Evaluation of the EFP Values in the Graph

The set $P = \{(v_x, v_y) \in E(\overrightarrow{G}) \mid \forall x \exists y : v_x \in V_{efp}(\overrightarrow{G}) \land v_y \in V_{efp}(\overrightarrow{G})\}$ contains EFP vertexes to be compared. A set of EFP values attached to these vertexes is obtained applying the function:

$$value : V(\overrightarrow{G}) \to V \tag{4.11}$$

where V is a set of EFP value types the instances of which have been obtained from the graph creation.

Having the AT set (Section 4.5.1, Equation 4.5) this function may be defined as follows (for the sake of simplicity, let us assume all sets are associative arrays, E and F were indexed by respective vertexes, AT by its subsets, and tuple elements are accessible via the '.' notation. In addition, the associative arrays allow wildcat selection resulting in sub-arrays with matched elements):

```
// global declarations
$LR$ := configuration parameter
$E$ := array with all EFPs indexed by respective vertexes
$F$ := array with all features indexed by respective vertexes
$AT = \{(e, f, v_a, AT_d) \mid e \in E \land f \in F \land v_a \in V_A \land AT_d \subset AT\}$


// start
$value$ := func $(x \in V_{efp}(\overrightarrow{G}))$ {
  $e$ := $E[x]$
  $f$ := $F[y], y \in V_{feature}(\overrightarrow{G}) \land ((x,y) \in E(\overrightarrow{G}) \lor (y,x) \in E(\overrightarrow{G}))$
  call $\sigma(AT[e,f])$ }

// function resolving one assigned value -- called recursively
$\sigma$ := func $(x \subseteq AT)$ {
  for $at$ in $x$ {
    $val$:=$at.v_a$
    if  $val \in V$ return $val$ // direct value
    if  $val \in S$ and $S \in LR$ return $val.v$ // LR simple value
    if  $val \in D$ and $D \in LR$ {
     // LR derive value
      return call $val.f(LR.S, \{\nu(val.e.e_{d_1}), \cdots, val.e.e_{d_N})\})$
    }
    if  $val \in C$ {
     // computed value
      $X$ := $\{AT[val.f, val.e, val]_1.AT_d, \cdots, AT[val.f, val.e, val]_N.AT_d\}$
      return call $val.h(\{\nu_A((x.f, x.e)_1), \cdots, \nu_A((x.f, x.e)_N) \mid x \in X\})$
    }
  }
  return null
}
$\nu$ := func ($e \in E$) {
  $e_p$:= $E[x, x \in E \wedge e \in x.E_d]$ // parent EFP
  return call $\sigma(AT[e_p].AT_d[e])$
}
$\nu_A$ := func ($f \in F, e \in E$) {
  $f_{prov}$ := $F[E(\overrightarrow{G})[f].v_y]$
  return call $\sigma(AT[e, f_{prov}])$
}
```

A function $\gamma : V \times V \to Z$ (from Section 4.3, Equation 4.1) compares value pairs, returning a numeric result. As a result, the set of vertex pairs is transformed to a set of numbers using a composed function.

$$\gamma \circ value : V(\overrightarrow{G}) \times V(\overrightarrow{G}) \to Z \tag{4.12}$$

This composed function applied to connected vertex pairs results in a set of numbers denoting quality on the respective vertex pairs.

A quality vector is generated from the input vertex set calling the functions 4.12 for each set item:

$$z_k = \gamma(value(x_k))), x \in P, k = 1..|P|$$

If a k^{th} item ($k \in 1..|P|$) in the vector contains a non-negative number, it means a quality has been satisfied. Consequently, non-negative values on all items mean the quality of the whole component composition has been satisfied. In any other case, EFPs attached to respective vertexes are incompatible.

Formally, the evaluation of the quality results in a compatible decision if and only if the following evaluation holds:

$$\forall k \exists z_k : z_k \in [0, \infty), k = 1..|P|$$

An Evaluation Example

Let us conclude this section with an example. Suppose there is a property *time_to_process* with numeric values and a γ function defined as: $\gamma(x, y) = x - y$ (shorter processing time is better). There are two direct values attached to EFP vertexes. The values are: $v_p := 10$ and $v_o := 30$. The evaluation mechanism calls the gamma function for these two vertexes: $\gamma(value(v_p, v_o)) = \gamma(value(10, 30)) = 10 - 30 = -20$. Since the result is a negative number, the evaluation results in incompatible EFPs. For different values $v_p := 40$ and $v_o := 30$ the evaluation succeeds with the result $\gamma(value(v_p, v_o)) = \gamma(value(40, 30)) = 40 - 30 = 10$.

Let us note that this simple example has been selected for the sake of brevity. The evaluation of mathematical formulas or LR values leads to comparing two instances of EFP data types the same way.

4.6.3 Resolution of Incompatible Components

Section 4.6.2 has introduced the algorithm finding incompatibility in a component assembly. Since the approach is based on the depth-first-search graph algorithm, it returns a result in a polynomial time. Despite that, incompatibility found by the algorithm obviously requires a user to replace invalid components. For a complete assembly, it leads to a time consuming operation.

Let us assume n components exist in a component repository from which m components can be selected to compose an application. Despite the fact the assembly may contain only one incompatible component, all m components may have to be replaced to create a compatible assembly. Assuming only one instance of one component may exist in the system, there is still $\frac{n!}{(n-m)!}$ variations of components to verify whether they fit to the assembly. Finding compatible components, all these variations may have to be verified by the algorithm from Section 4.6.2. As a result, the finding of a working assembly has a time complexity $O(\frac{n!}{(n-m)!})$.

Dealing with the mentioned complexity, this part of the work introduces an approach finding compatible components in a polynomial time. The approach represents a heuristic rather than an exact method. It means that the method does not have to find a solution though it may exist. On the other hand, a solution found by this approach is always correct. It essentially means the assembly composed and verified by this approach has compatible components.

The core idea of the method is that each incompatibility is solved by replacing one component. Although in a general case an incompatibility may need to replace more than one component, in practical applications, it is assumed that an incompatibility is resolved by changing one component.

There are at least two motivations for such an assumption. The first motivation is in a repository containing more versions of one component with improving quality following the increasing versions. The idea is that new versions of a particular component are released. Another case is a set of components provided by competitive vendors where a particular component has a better quality characteristics than components from other vendors.

According to these facts, a replacement of one component by a “better“ component can solve any incompatibility.

The second motivation for replacing one component to solve one incompatibility follows a typical manual process in which a user replaces components one-by-one to find a compatible assembly. The reason is obvious, a user can hardly predict the impact of replacing multiple components, so he or she tends to replace and verify each component in discrete steps. As a result, this manual approach may be overtaken by a tool implementing the approach presented here first. Then a user may manually solve the incompatibility that this heuristic approach will not solve.

The method presented in this section consists of two approaches: (1) incremental composition of components and (2) a reconfiguration of an existing composition. However, both these phases may be used separately since they do not depend on each other.

An Incremental Composition

In general, system composition [4][89] is an approach to build a system from discrete parts to reach a final functionality. As it may be seen, this approach has been overtaken by the CBSE concept. In addition, the incremental composition is an approach in which a system is composed in steps so that each step consists of adding one part of a system followed by its verification. If the verification succeeds the composition proceeds to the next step repeating the same process until the application is completed [66].

The incremental composition is a linear process, in which each step creates a verified sub-composition and it prevents the need to verify all variations of candidate components. This process is depicted in Figure 4.12 in which several components have been already added to a composition. The last component to add is incompatible, causing other components from the repository to be checked whether they can replace the incompatible component.

Let us define a sequence $A = (C_1, C_2, C_3, \cdots, C_m)$ where $C_i, i \in \{1, 2, \cdots, m\}$ are components composing an assembly and m is the total number of components of the assembly.

The incremental process of the assembly creation consists of steps in which all components already put in the assembly are verified and compatible. If a new component is compatible with the components already being in the assembly, this component may be added to the assembly. If the component is not compatible, another component must be selected from the repository.

Formally, the assembly creation repeats in steps for $j \in (1 \cdots m)$ so that an assembly creation process incrementally creates a sequence $A = (C_1, C_2, \cdots, C_{j-1}) \cup \{C_j\} \cup (C_{j+1}, C_{j+2}, \cdots, C_m)$ where components with

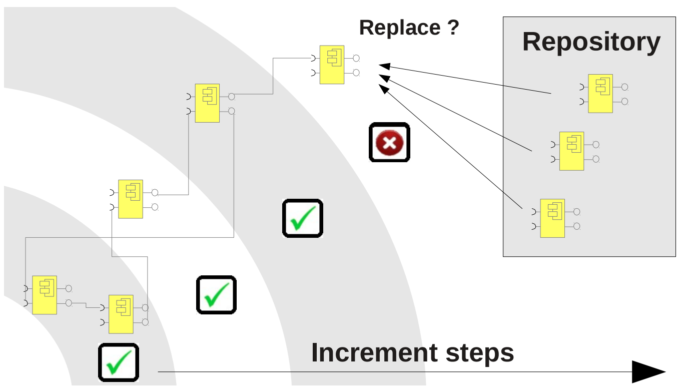


Figure 4.12: Incremental Composition

indexes from the set $\{1, 2, \cdots, j-1\}$ have been verified and are compatible components, the component with the index j is currently being added and the components with indexes from the set $\{j+1, j+2, \cdots, m\}$ will be added in later steps.

Using the incremental composition, there are m components composing the assembly. In each step of the composition, a total number of n components in the repository is tried. As a result, the incremental composition needs $m \times n$ steps to build a verified assembly. Hence the time complexity of the method is $O(mn)$.

The algorithm from Section 4.6.2 may verify newly added EFPs in each step of the assembly creation instead of the evaluation of the complete assembly.

On the other hand, if a sub-composition has no compatible component to be added, a re-configuration of the actual sub-composition may change the sub-composition so that the new component happens to be compatible. It leads to the method proposed in this work.

A Reconfiguration of an Existing Composition

Let us have a sequence of components $A' = (C_1, C_2, \cdots, C_{j-1})$ composing a partial assembly. There is an attempt to add a component C_j to this assembly. If the component C_j is not compatible with the assembly and there is no other compatible component in the repository, the incremental composition ends. However, a reconfiguration of the existing assembly may solve the problem.

Whereas general reconfiguration of all current components would lead to the mentioned time complexity $O(\frac{n!}{(n-m)!})$, this section introduces a different approach with a lower time complexity.

It is practical to have the components in the A' sequence in the topology order. It has positive impact known from the back-tracking algorithm: if a component C_i is replaced, all components $\{C_j \mid j < i\}$ cannot be affected by the replacement while all components $\{C_j \mid j > i\}$ may be affected. It reduces the time complexity because only several components must be re-verified if a replacement of the component C_i has its index i close to the index j of the component C_j.

There are two ways of obtaining this sequence depending on the input assembly: (1) if the assembly has been created using the incremental composition, the sequence order is equal to the order in which the components have been added to the assembly, (2) the topology order is the same as the order of finished vertexes visited by the back-tracking algorithm.

Although a topology creation requires no circular dependency in a graph, circular dependencies of the component features are not problematic. In that case, the topology is simply created by putting the vertexes in the order they were finished by the back-tracking algorithm. On the other hand, circular dependencies of EFPs are problematic and the method presented here cannot deal with them.

Let us assume a resulting sequence has compatible components $(C_1, C_2, \cdots, C_{j-1})$ ordered following their binding dependencies. There is a component C_j that cannot be added because of its incompatibility and it is attempted to replace one component of $(C_1, C_2, \cdots, C_{j-1})$ to solve the incompatibility.

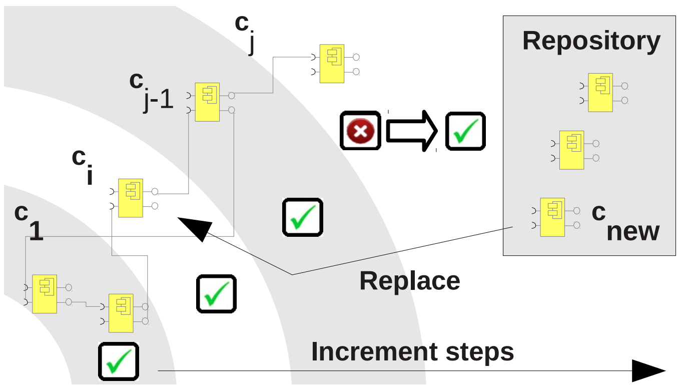


Figure 4.13: Incremental Composition with Reconfiguration

A re-configuration process attempts to replace one component with the index $i \in \{1, \cdots, j-1\}$ in the assembly. The step in which one component is replaced by another one from the repository is shown in Figure 4.13.

The algorithm expressed in pseudo-code is defined as follows (For the simplicity, the algorithm does not check extreme limits where e.g. a replaced component is first or last in the sequence):

```
1. $A'' := (C_1, C_2, \cdots, C_{j-1}) \cup \{C_j\}$, $Repository$ is a component repository,
   $C_{new}$ := null.

2. for $i = j - 1 \cdots 1$ {
     $C_i$ := $A''[i]$

3. for $k = 1 \cdots n - j$, $C'$ := $Repository[k]$ {
     if type_compatible $C'$, $C_i$ then {

4.     if not efp_compatible $C'$,$C_{i-1}$ then goto 3.

5.     $A^* := (C_1, C_2, \cdots, C_{i-1}, C', C_{i+1}, \cdots, C_{j-1})$
       for $l = i \cdots j - 1$
           if not efp_compatible $A^*[l]$,$A^*[l+1]$ then goto 3
       $C_{new}$ := $C_i$ := $C'$
       goto 6
     }
   }
   }

6. if $C_{new}$ == null then ERROR
     else $A' := (C_1, C_2, \cdots, C_{i-1}, C_i, C_{i+1}, \cdots, C_{j-1}, C_j)$
```

The method `type_compatible` represents the μ function from Section 4.5.1, the method `efp_compatible` evaluates compatibility of EFPs using the means from Section 4.6 and their bodies are omitted here.

If this algorithm finds a new C_{new} component, it is replaced in the assembly and the C_j component may be added. If there is a set of other incompatible components $(C_{j+1}, C_{j+2}, \cdots, C_m)$ to add, the algorithm must repeat for each component.

As it may be seen from the algorithm, it tries to replace each of j components by n components from the repository. All j components are re-verified. All of m components in the assembly may be potentially replaced. As a result, the algorithm requires $j \times n \times j \times m$ steps and its time complexity is $O(n^4)$.

To sum up, this section has provided an overview of two methods allowing to create a verified component assembly in the polynomial time. The method is based on the assumption that each incompatibility is solved by the replacement of one component. Despite the simplification, it is assumed that the method is usable in real scenarios since it follows a use-case replacing and verifying components in discrete steps.

Chapter 5

Implementation and Application

The whole EFFCC approach which has been, so far, presented using formal means, is also implemented[1]. The purpose of this chapter is to present the implementation over viewing main architectural decision to help a reader to understand main concepts of the implementation. The implementation serves as a proof of the concept, however, it is capable of application to practise.

The following pages will respectively show the implementation of EFP Repository Server, EFP Assignment and EFP Evaluator in Sections 5.1, 5.2 and 5.3,. Since these three modules represent core parts of EFFCC, several important details about the implementation will be provided. In opposite, Section 5.4 will not target an implementation design, but it will show graphical tools from a user perspective. The rationale is to show how user may work with the EFFCC rather than detailing common programming. Finally, Sections 5.5.1, 5.5.2, 5.5.3 are brief introduction of the EFFCC application to one research and two industrial component frameworks demonstrating a practical applicability.

5.1 The EFP Repository Server

The EFP repository has been implemented as a web 3-tier server shown in Figure 5.1.

From a technological point of view, the server stores EFP data in a relational database accessed from the `DAO` (data access object) layer which maps

[1]Source code available at: http://subversion.assembla.com/svn/efps/ (2012)

relation data to Java objects using the Hibernate framework[2]. The `Service` represents a business facade encapsulating computation upon EFP data and transferring the data between the data and view layers.

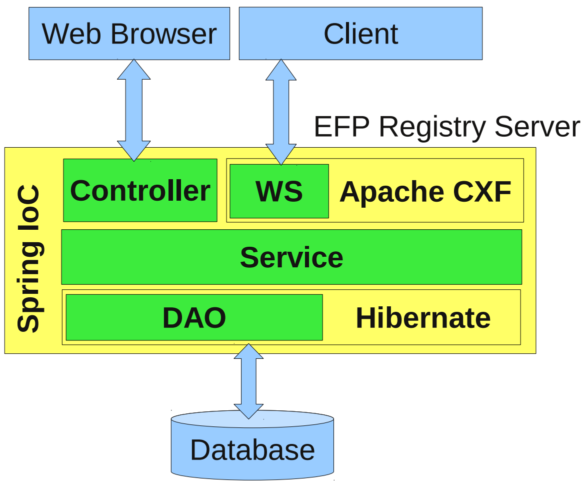


Figure 5.1: EFP Registry Server – Architecture Design

The view layer is implemented using two different means. A first possibility to access the data is via a web browser using the `Controllers` layer. Its purpose is to provide a modern fashioned and comfortable user access point from a web browser. The other layer uses web-services (`WS` in Figure) allowing to access the data from other applications. Its purpose is to have an access point for other application. Currently the data from the repository are loaded by the EFP Assignment and the desktop client using the web-services, however, any other application may connect and participate in using the EFP common storage. Let us note the Apache CXF[3] framework is used for the implementation of the web-services.

Moreover, the implementation of the EFFCC is supported by the Spring inversion-of-control and model-view-controller framework[4]. In practice, the structure used for the server implementation is nothing more than the structure suggested by Spring.

According to fundamentals of the web-service technology, it produces raw output via web services using the Service Oriented Architecture protocol (SOAP). SOAP transfers a low-level XML data via standard HTTP protocol with WSDL files describing a structure of the XML data.

[2]Available at: www.hibernate.org/ (2012)
[3]Available at: cxf.apache.org/ (2012)
[4]Available at: www.springsource.org/ (2012)

On the one hand, the used technology allows a low-level access by several clients working with the SOAP, on the other hand, the plain XML data may be awkward for other modules. In order to solve this problem, we have created a sub-module named the EFP Client, shown in Figure 5.2, which basically turns XML data into EFP Types and vice versa. Hence, client modules access data via EFP Client using familiar EFP Types first, then the repository transparently receives XML data.

On the one hand, the current Java-based implementation limits the EFP Client usage to other Java-based application, on the other hand, the pure SOAP may be still used to integrate different platforms.

The repository serves as a generally usable storage of the extra-functional properties. Although it is originally developed for other parts of EFFCC, it may be as well used as a stand-alone storage available to third-party applications.

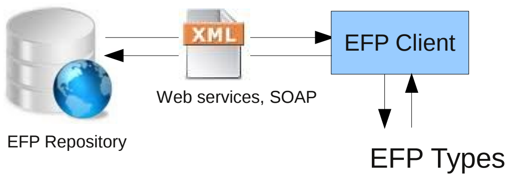


Figure 5.2: Server-Client Communication

5.2 The EFP Assignment

The two modules for transferring data (EFP Types and EFP Assignment Types) as well as the EFP Assignment module have been developed in Java. Implementation design of the EFP Assignment follows the idea of splitting component model to dependent and independent parts. A couple of interfaces to access EFPs on components by third-party applications are provided with a convenient abstraction layer and a default out-of-the-box implementation.

In detail, a core part of the implementation design is shown in Figure 5.3. Two interfaces `ComponentEFPAccessor` and `ComponentEFPModifier` are the most abstract representation of the EFP Assignment module. They respectively provide API to read and modify EFP data in a form of the EFP Assignment Types. The former one is typically accessed by the EFP Evaluator module because the evaluator only needs to read the data while the latter one is typically accessed by an assignment tool to modify the EFP data on components.

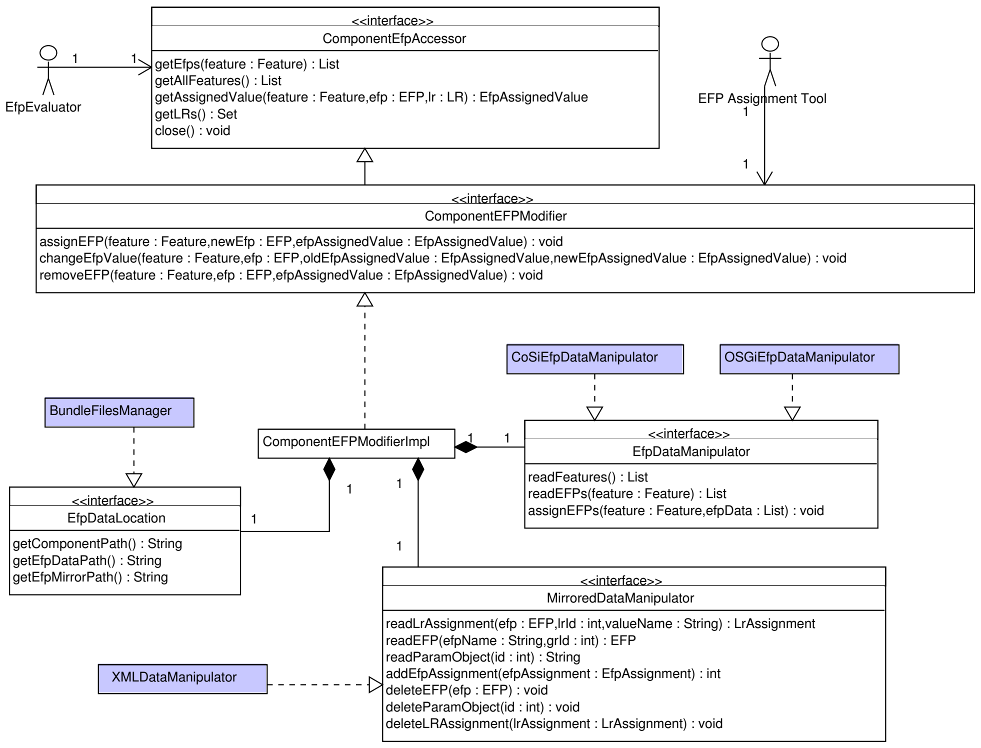

<<interface>>
ComponentEfpAccessor
getEfps(feature : Feature) : List
getAllFeatures() : List
getAssignedValue(feature : Feature,efp : EFP,lr : LR) : EfpAssignedValue
getLRs() : Set
close() : void
EfpEvaluator
EFP Assignment Tool
1
1
1
1
<<interface>>
ComponentEFPModifier
assignEFP(feature : Feature,newEfp : EFP,efpAssignedValue : EfpAssignedValue) : void
changeEfpValue(feature : Feature,efp : EFP,oldEfpAssignedValue : EfpAssignedValue,newEfpAssignedValue : EfpAssignedValue) : void
removeEFP(feature : Feature,efp : EFP,efpAssignedValue : EfpAssignedValue) : void
CoSiEfpDataManipulator
OSGiEfpDataManipulator
BundleFilesManager
ComponentEFPModifierImpl
1
1
<<interface>>
EfpDataManipulator
readFeatures() : List
readEFPs(feature : Feature) : List
assignEFPs(feature : Feature,efpData : List) : void
<<interface>>
EfpDataLocation
getComponentPath() : String
getEfpDataPath() : String
getEfpMirrorPath() : String
1
1
1
1
<<interface>>
MirroredDataManipulator
readLrAssignment(efp : EFP,lrId : int,valueName : String) : LrAssignment
readEFP(efpName : String,grId : int) : EFP
readParamObject(id : int) : String
addEfpAssignment(efpAssignment : EfpAssignment) : int
deleteEFP(efp : EFP) : void
deleteParamObject(id : int) : void
deleteLRAssignment(lrAssignment : LrAssignment) : void
XMLDataManipulator

Figure 5.3: EFP Assignment – Core Architecture Design

Another model element, `ComponentEFPModifierImpl` is an out-of-the-box convenient class and implementation of `ComponentEFPModifier` splitting concerns of component model dependent and independent parts. Each request to load or modify EFP data on a component is captured in this class and distributed to the dependent or independent part. The component model independent part is an already mentioned EFP data mirror represented by the `MirroredDataManipulator` interface while an independent part is represented by the `EfpDataManipulator`. For instance, once a user attaches an EFP to a component, the `ComponentEFPModifierImpl` class divides the data to part to be stored in the EFP mirror and to part to be stored directly on the component. It is a task of concrete implementation of correctly storing the data for concrete component model. Finally, the class `EfpDataLocation` allows the EFP Assignment module to recognise where the EFP data are persisted on the component (e.g. where XML files are).

Classes highlighted by the grey color represents the points where light-weighted plug-ins are connected to extend the EFP Assignment for concrete component model. The `XMLDataManipulator` is a default implementation of the EFP data mirror persisting the data in XML files. This implementation may be used for most of component models, however, a developer is free to reimplement it for different kind of storage. Two classes `CoSiEfpDataManipulator` and `OSGiEfpDataManipulator` are two example implementations for CoSi [17] and OSGi component models. These two implementations connect EFP data from the mirror to concrete features of the component models. Together with CoSi and OSGi implementation, the `BundleFilesManager` class is implemented to connect the EFP Assignment with the CoSi and OSGi bundle structure.

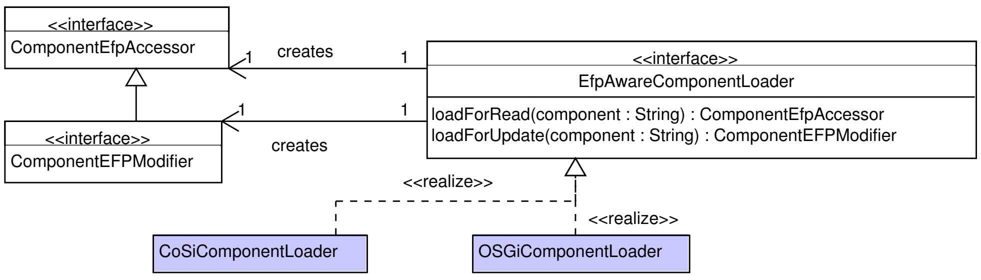


Figure 5.4: EFP Assignment Loader – Architecture

The implementation also provides a mechanism shown in Figure 5.4 creating EFP Assignment instance for concrete component model. Once a developer prepares implementations for a concrete component model in a form of classes mentioned above, he or she packs it together in an implementation of the `EFPAwareComponentLoader` interface. Implementing this interface a factory class is created. An implementation of the factory may return classes directly implementing the `ComponentEFPAccessor` and `ComponentEFPModifier` interfaces, however, in most situations convenient implementations from Figure 5.3 are used with the advantage of implementing very little piece of additional code. The Figure 5.4 shows two example implementations `CoSiComponentLoader` and `OSGiComponentLoader` for OSGi and CoSi component models.

As it has been demonstrated, an extension of EFP Assignment module basically means an implementation of three interfaces which instances are packed together and returned by an implementation of a factory class.

5.3 The EFP Evaluator

The EFP Evaluator prototype is implemented in Java as a module embeddable to other Java-based component models. Together with the EFP Assignment module, it composes a module seamlessly managing EFPs on any component models which the EFP Assignment module has been prepared for. In this cooperation, the EFP Evaluator works with independent EFP data abstraction while the EFP Assignment is customised for a concrete component model.

Due to the fact the EFP Evaluator implements the algorithm from Section 4.6.2 it provides API following the same two steps algorithm: (1) construction of a component graph and (2) evaluation of the graph. Any application integrating the EFP evaluator will therefore call this API to process the algorithm the way it has been formalised in Section 4.6.2. The details are, however, hidden for clients which only access the high level API .

A first method available in the API loads components and their EFP representation via the EFP Assignment module. The method is stored in the `ComponentLoader` class:

```
GraphCreator loadComponents(
  List<String> components,
  String assignmentModule,
  Integer... lrID);
```

where the `components` argument represents a set of component identifications which are to be evaluated. These abstract component identifiers are sent to the EFP Assignment module that loads concrete component representation.

The `assignmentModule` argument expresses a name of a class implementing the `EfpAwareComponentLoader` from Section 5.2. A class with this name is instantiated as the module starts. This mechanism allows to customise the EFP Evaluator module to operate with different implementations of the EFP Assignment for multiple component models. This is, therefore, a point in which the contract between EFP Assignment and Evaluator is stated.

The last argument, `lrID`, represents unique identifiers of LRs which all evaluation will be made for. Typically, these identifiers are loaded from configuration remaining unchanged at runtime. The reason is that one installation of a component framework obviously run in the same environment at all the time. Therefore, each evaluation of components works with defined LRs ignoring all values for other LRs. For instance, an OSGi installation in a mobile phone will have installed LR for mobile devices ignoring values for servers, desktops, etc.

A result of the method is a `GraphCreator` class which instance generates the graph of the component binding. The graph is generated calling the method:

```
DirectedGraph<GraphVertexRepr, DefaultEdge>
  create(MatchingFunction matchingFunction);
```

As a result of this method, the first part of the algorithm from Section 4.6.2 is finished. The component graph is created.

The interface `MatchingFunction` on the input of the `create` method represents the function μ (Equation 4.7 from Section 4.5.1). Therefore, users may put any matching function on the input which essentially impacts the feature binding in the graph creation. Either user defined or implicit function may be used, however, in a lot of situations a matching algorithm from concrete component model is called. The function is simple:

```
public interface MatchingFunction {
    boolean matches(Feature feature1, Feature feature2);
}
```

Two features on the input are examined and `true` is returned if they should be bound.

Having graph representation, the second part of the algorithm starts. It must evaluate the graph to find an incompatible binding and incompatible EFPs. The evaluation is provided by the `ComponentEvaluator` class with the method:

```
List<EfpEvalResult> evaluate(
  DirectedGraph<GraphVertexRepr,  DefaultEdge> graph);
```

As may be seen, the method takes the graph created by the method above on its input. The output is a set of `EfpEvalResult` classes that hold the evaluation result in a structured form allowing its later processing by other tools. Each `EfpEvalResult` comprises of the compatibility decision and an aggregation of EFP pairs together with components and features the EFPs are attached to. In addition, any incompatibility decision is expressed in terms of missing required elements or incompatible values on matching EFP pairs. Receiving the information of affected EFPs and components, a user may easily determine incompatibility.

Taking it all together, the EFP Evaluator API provides clients with the compatibility algorithm in a short sequence of method calls:

```
List<String> components = ... // a list of components to load
Long lrID = ... // a LR ID from configuration
// a specific component model implementation
// An OSGi implementation provided here
String module = OSGiComponentLoader.class.getName();
MatchingFunction mu = ...; // mu function
ComponentLoader loader = new ComponentLoaderImp();
```

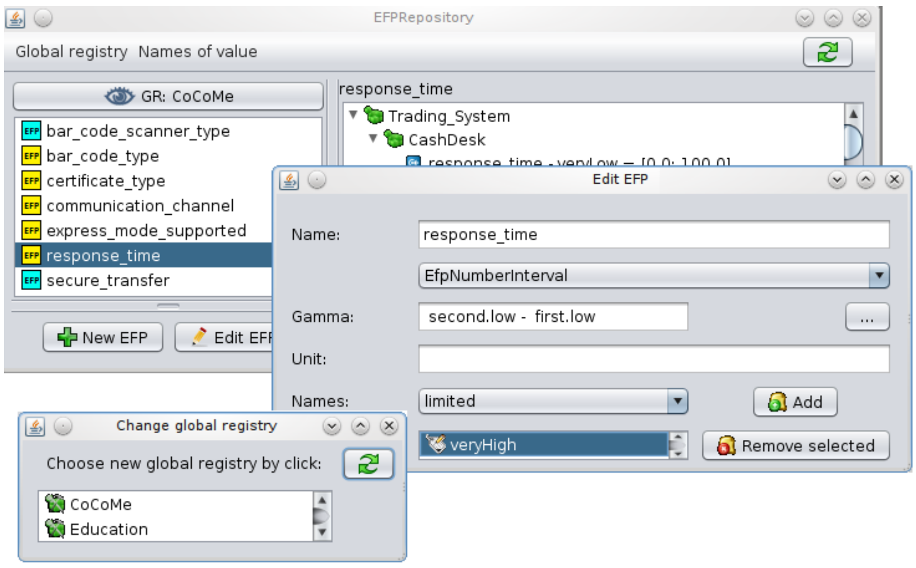
```
GraphCreator creator = loader.loadComponents(
  components, module, lrID);
DirectedGraph<GraphVertexRepr, DefaultEdge> graph = creator.create(mu);
ComponentEvaluator evaluator = new ComponentEvaluatorImp();
List<EfpEvalResult> result = evaluator.evaluate(graph);

// 'result' has a list of compatibility decisions
```

A part of the implementation working with the graph uses the third-party library JgraphT[5] for both creating and discovering vertexes of the graph. It has lead to an easier implementation of the evaluating algorithm.

5.4 Tools

Since core modules composing EFFCC aim at being used in other systems, their direct access by users is not assumed. Therefore, a management of EFPs in terms of their manipulation either in the repository or on the components should be supported by tools. For that reason, tools offering comfortable graphical interfaces have been developed.



bar_code_scanner_type
bar_code_type
certificate_type
communication_channel
express_mode_supported
response_time
secure_transfer

response_time
Trading_System
CashDesk
response_time - veryLow = [0.0; 100.0]

Name: response_time
EfpNumberInterval
Gamma: second.low - first.low
Unit:
Names: limited
veryHigh

Choose new global registry by click:
CoCoMe
Education

Figure 5.5: EFP Repository Tool

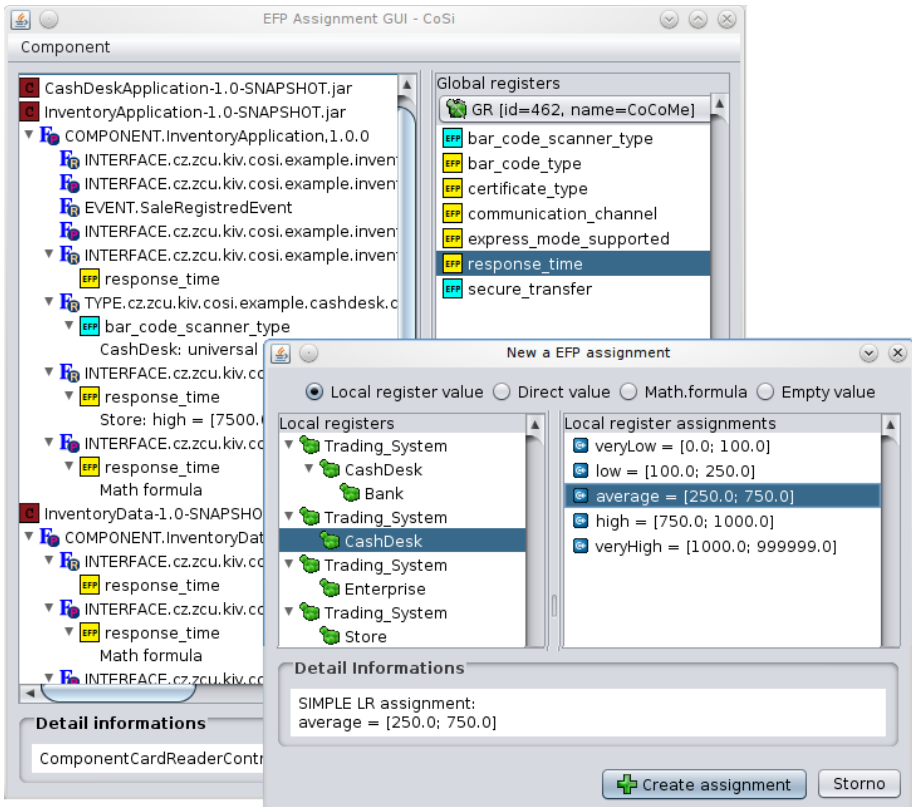
The first tool shown in Figure 5.5 works with the EFP repository to provide

[5]Available at: www.jgrapht.org/ (2012)

control upon the repository data. The implementation is a Java JFC Swing client communicating with the server via web-services.

A rich palate of edit controls of the tool follows structure of EFPs and registries. Forms to edit EFPs in GR, their values in LR, management of existing GRs are essential parts of the application.

Although the web client is also part of the EFP repository server, the choice to developed also a desktop client has been practical for test purposes, because the process of the development simultaneously tests the web-services.



EFP Assignment GUI - CoSi

Component

```
C  CashDeskApplication-1.0-SNAPSHOT.jar
C  InventoryApplication-1.0-SNAPSHOT.jar
▼ F COMPONENT.InventoryApplication,1.0.0
    F INTERFACE.cz.zcu.kiv.cosi.example.inven
    F INTERFACE.cz.zcu.kiv.cosi.example.inven
    F EVENT.SaleRegistredEvent
    F INTERFACE.cz.zcu.kiv.cosi.example.inven
  ▼ F INTERFACE.cz.zcu.kiv.cosi.example.inven
      EFP response_time
  ▼ F TYPE.cz.zcu.kiv.cosi.example.cashdesk.c
    ▼ EFP bar_code_scanner_type
        CashDesk: universal
  ▼ F INTERFACE.cz.zcu.kiv.cc
    ▼ EFP response_time
        Store: high = [7500.
  ▼ F INTERFACE.cz.zcu.kiv.cc
    ▼ EFP response_time
        Math formula
C  InventoryData-1.0-SNAPSHO
▼ F COMPONENT.InventoryDat
  ▼ F INTERFACE.cz.zcu.kiv.cc
      EFP response_time
  ▼ F INTERFACE.cz.zcu.kiv.cc
    ▼ EFP response_time
        Math formula
  ▼ F INTERFACE.cz.zcu.kiv.cc
```

Detail informations

ComponentCardReaderContr

Global registers

GR [id=462, name=CoCoMe]

- EFP bar_code_scanner_type
- EFP bar_code_type
- EFP certificate_type
- EFP communication_channel
- EFP express_mode_supported
- EFP response_time
- EFP secure_transfer

New a EFP assignment

[● Local register value] ○ Direct value ○ Math.formula ○ Empty value

Local registers

```
▼ Trading_System
  ▼ CashDesk
      Bank
▼ Trading_System
    CashDesk
▼ Trading_System
    Enterprise
▼ Trading_System
    Store
```

Local register assignments

- veryLow = [0.0; 100.0]
- low = [100.0; 250.0]
- average = [250.0; 750.0]
- high = [750.0; 1000.0]
- veryHigh = [1000.0; 999999.0]

Detail Informations

SIMPLE LR assignment:
average = [250.0; 750.0]

Figure 5.6: EFP Assignment Tool

The other tool shown in Figure 5.6 is a Java JFC Swing client accessing the EFP Assignment module. The client's main functionality is to show EFPs available in the repository and to attach them to components. The users drag-and-drop EFPs from the panel with all EFPs to the other panel with component features. Moreover, each drag-and-drop operation opens a dialogue to select an EFP value. Either direct, context-dependent or computed values may be selected.

Since the EFP Assignment module may be implemented for different component models, the tool is also capable of switching to a particular implementation. A list of all implementations is configured in a properties file first, then a user selects a concrete implementation once the tool is starting.

5.5 Application of the Approach

So far, EFFCC has been introduced using both formal means and metamodels with implementation details. Due to the fact the main goal is to embed EFFCC to existing component models, a few prototype applications have been developed. For that reason, this section aims at showing the ability of EFFCC to be applied to other system consequently fulfilling the main goal.

5.5.1 EFPs in the Spring IoC Container

One of the widely used component frameworks is Spring. Components in Spring have forms of, so called, Beans where one Bean is one Java class. Dependencies between components are explicitly expressed in configuration XML files which in essence define a so called Application Context [94].

Let us note that the XML file based configuration comes from the very first versions of Spring. A configuration of Beans in XML files provides a static dependency binding configuration that is established before an application starts. Later versions of Spring added the support of a dynamic binding configuration expressed in XML files or even newer versions allow a configuration in the Bean source code using Java 1.5+ annotations.

For the sake of clarity, this section shows the application of the EFFCC using the most traditional XML-based configuration. Application of EFFCC to other Spring's extensive configuration means will be omitted. An extension of the presented approach to support e.g. the annotation driven configuration is a straightforward re-implementation of the approach to respective parts of the Spring framework, however, the principles remain the same.

In addition, Spring allows either a setter or a constructor injection of the dependencies. Since the setter injection is a recommended approach in Spring, it will be used also in this section. An implementation considering also the construction injection would follow the same principles to be presented here for the setter injection.

Spring Beans defined in the application context may be considered as provided features in terms of the component binding. Each setter of a Bean denotes the required side. Hence, the binding of the provided to required side is equivalent to the examination of values (objects) injected into the setters of the Bean.

Extending Spring of extra-functional support, several steps must be taken to integrate EFFCC. First of all, the EFP Assignment module must be extended to attach EFPs to Spring Beans.

Working with the Spring XML-based configuration, the EFP extension may be achieved by extending Spring's XML configuration files using XML namespaces. Then the XML configuration file will contain other elements declaring EFPs for each Bean. In addition, we suggest a solution in which the EFP data mirror is a stand-alone XML file – as it has been mentioned in Section 5 – and the links between the mirror and Spring Beans are stored in the extended Spring XML files.

The main advantage of this solution is that the new XML tags do not clash with existing ones and the Bean definition is separated from the definition of EFPs. Hence, the Application Context configuration is enriched with EFPs without touching original implementation of Spring.

Example of the solution based on the extended XML files:

```
<bean id="data"
 class="cz.zcu.kiv.example.DataAccess" >
 <property name="jdbc" ref="jdbcDriver" />
 <efp:name="response-time" property="jdbc">
    <efp:values>
      <efp:lr id="1" value="average" />
      <efp:direct value="100" />
    </efp:values>
 </efp:name>
</bean>
```

If the annotation driven configuration is used in Spring, a different approach should have to be used. For instance, standard Spring annotation can be extended of new annotations expressing EFPs. Either new annotations may be implemented or even existing approaches may be integrated. E.g. JSR 305 [46] (Java Specification Requests 305) is already targeting annotations concerning a few extra-functional properties with annotations such as `@NotNull`. Although it targets only simple annotations.

In order to evaluate EFPs attached on Spring Beans, the EFP evaluator must be aware of the Bean binding. To obtain the binding information directly from the Spring framework, the suitable solution is to participate in the container life-cycle. Spring contains a set of so called Bean Post Processors providing developers with a rich spectrum of call-back methods allowing to observe and even modify the container life-cycle.

Although there is a wide spectrum of the Bean Post Processors, for the purposes of components matching, the `InstantiationAwareBeanPostProcessorAdapter` is the most suitable one. It allows to observe which Bean has been instantiated and which values have been injected via setter methods.

The implementation of a Bean Post Processor evaluating EFPs on Spring Beans is here introduced in a sequence of steps:

1. A class extending the `InstantiationAwareBeanPostProcessorAdapter` is created:

   ```
   public class EfpAwareBeanPostProcessor
     extends InstantiationAwareBeanPostProcessorAdapter {

     /** A map of all beans represented as EFP Assignment features.  */
     private Map<String, Feature> beanFeatures =
       new HashMap<String, Feature>();

     /** This map holds all bound features. */
     private Map<Feature, Feature> boundFeatures
       = new HashMap<Feature, Feature>();

     /** A local registry ID from the configuration. */
     private Integer lrId;
   ```

   The map `beanFeatures` holds all Beans instantiated in the Application Context while the map `boundFeatures` holds information of Bean bindings. While the former map will be used to inform the EFP Evaluator which Beans to evaluate the latter map will be used for implementing a matching μ function. A property `lrId` is a pre-configured identifier for LR.

2. A method inherited from the super class is created:

   ```
   public PropertyValues postProcessPropertyValues(
            final PropertyValues pvs,
            final PropertyDescriptor[] pds,
            final Object bean,
            final String beanName)  {
   ```

   Input parameters contain complete information of a one Bean binding. Namely, `pvs` contains a list of values to be set to the `bean` with the name `beanName`. `pds` is a list of descriptions for each value to be set. This description holds among others the name of each attribute to be set.

3. A Bean may generally have a setter to set another Bean as well as another Java value (e.g. an Integer number). Any attribute does not even have to be set remaining `null`. Since the EFP evaluation process is interested only in the Bean binding, the implementation must skip all `null` or non-Bean reference values. On the other hand, all injected Beans must be concerned.

   ```
   for (PropertyDescriptor pd : pds) {
     PropertyValue prop = pvs.getPropertyValue(pd.getName());

     // a property has no value to be set.
     if (prop == null) {  continue;  }

     Object value = prop.getValue();

     // we are interested only in a bean binding
     // we omit e.g. value binding.
     if (value instanceof BeanReference) {
       Feature firstBean = findBeanFeature(beanName);

       String featureName = prop.getName();
       Feature required = new BasicFeature(
         featureName, Feature.AssignmentSide.REQUIRED,
         "service", true, firstBean);

       BeanReference secondBeanRef = (BeanReference) value;
       Feature secondBean = findBeanFeature(
         secondBeanRef.getBeanName());

       Feature provided = new BasicFeature(
         featureName, Feature.AssignmentSide.PROVIDED,
         "service", true, secondBean);

       boundFeatures.put(required, provided);
   }
   ```

   The presented code skips all `null` values first, then all values only being instances of Beans are processed. Having a Bean and another Bean to be injected to this Bean, the respective EFP Features are created. In the presented code, two features are created so that both features have the same name as the name of the bound attribute. The feature on the Bean holding the attribute to be injected is the required one while the feature of the Bean to be injected is the provided one. Finally, these two features are put to the `boundFeatures` map as a matching pair.

The method `findBeanFeature` creates an EFP Feature Bean representation for the input Bean and stores it in the `beanFeatures` map. It stores information which Beans are actually in the Application Context. As a result, this information is later passed to the EFP evaluation. Using these steps, both the information of Beans existing in the Application Context and information of their binding is obtained.

The presented code shows the process of obtaining information of Beans running in the system together with their binding. The binding information is then used for the implementation of the μ function. The implementation itself is straightforward: each input feature pair is tried to be located in the binding map. If the pair is found, the features are bound and the matching function results `true`.

4. Creating the EFP evaluating mechanism, values obtained in previous steps are passed to the EFP Evaluator. Hence, the same class has an evaluating method calling the EFP Evaluator:

```
public List<EfpEvalResult> evaluateEfps() {
  List<String> beans = new ArrayList<String>(
    beanFeatures.keySet());

  return EfpComparator.evaluate(
    beans, SpringAssignmentImpl.class.getName(), mu, lrId);
}
```

   As it may be seen, the Beans obtained from the Application Context are passed to the EFP Evaluator. The `SpringAssignmentImpl` class is the EFP Assignment plug-in for Spring. The matching function implementation (a variable `mu`) is omitted here, however, an implementation has been summarised in the previous step. The value `lrId` is an already mentioned pre-configured LR identifier. `EfpComparator.evaluate` is a simple facade method internally calling sequence of methods from Section 5.3.

To sum up, the implementation of EFP Assignment plug-in and the Bean Post Processor will enable EFP support in Spring. Every time a Spring-based application is started the Bean Post Processor determines Bean binding and passes this information to the EFP Evaluator. To enable this mechanism, the Bean Post Processor must be defined in the Application Context the same way as any other Bean is defined. Hence, the mechanism is non-intrusive and providing the EFP support only if it is needed with the option of enabling or disabling it.

The implementation using the XML-based Application Context configuration with the setter injection supports a static Bean binding where all Beans and their relations are defined before the application starts. In a lot of areas, EFPs are more suitable for dynamic binding (in Spring called *autowiring*). In the dynamic binding a Bundle only states required types (in terms of Bean types) and the framework automatically binds dependencies according to all available types. In this process, more than one type may satisfy a particular dependency. In that case, a user must manually resolve the dependency clash (removing or qualifying the duplicated types). These manual corrections may be partly replaced using the EFPs that serve as another qualification influencing the binding. On the other hand, the static binding is most traditional and most often used in Spring and the overall idea of integrating EFPs to Spring is still the same. For that reason, the EFP integration has been explained on the traditional XML-based static configuration while this approach preserves generality applicable to other parts of Spring framework.

5.5.2 EFPs in OSGi

The OSGi framework [86] components are called Bundles, distributed as Java JAR files. Bundles may register a set of services (Java classes) that other Bundles may call. Bundle services are grouped into packages which in OSGi must be explicitly imported (required) or exported (provided). The specification of each Bundle, exported and imported packages, is written in a text form as part of the manifest file.

Considering information stored in the manifest files, a first option to extend OSGi by EFPs is to supplement the content of the manifest file. For instance, a database layer of an application may be enriched with extra-functional information as follows:

```
Manifest-Version: 1.0
Bundle-Name: Data
Export-Package: cz.zcu.kiv.osgi.example.dao;
  efp:=1.db_engine=LR.2.memory
```

meaning that a property `db_engine` from GR with the identifier 1 has assigned a name `memory` from the context of a LR with the identifier 2. It is assumed that the meaning of these identifiers is stored separately in the EFP data mirror attached to the Bundle. A suitable storage for the EFP data mirror is a XML file stored as part of the Bundle in the META-INF folder where also the manifest file is located. Therefore, the distribution of Bundles will also contain EFP data information.

This concept is similar to OSGi *capabilities* (OSGi release 4 specification [86]) which use name-spaces in similar manner to LR value names and EF-FCC is even capable of implementing OSGi's capabilities since they are nothing more than name-value pairs with restricting formulas covered by our deriving formulas. However, the capabilities lack unification such as provided by the Registries. Moreover, both approaches may be too coarse-grained since the provided and the required elements are on the package level. Therefore, this EFP assignment option does not necessarily prevent incompatible services (Java classes in practice) to be run.

Another innovative concept of the OSGi 4 are Declarative Services (DSs). DSs provide Bundles with a fine tuned declaration of particular services stored in XML files. Hence, EFPs can be in detail defined for services using the idea equivalent to the XML name-spaces, developed in this paper for Spring, applied to DSs.

Extending the manifest file with a link to a Declarative Service specification

```
Manifest-Version: 1.0
Bundle-Name: Data
Service-Component: OSGI-INF/dao.xml
```

the `dao.xml` file contains the declaration of one particular service `DataAccess` implemented by a `DAImplHSQL` class. This can be enhanced with EFPs:

```
<component name="dao">
<implementation class="cz.zcu.kiv.osgi.app.dao.DAImplHSQL"/>
  <service>
    <provide interface="cz.zcu.kiv.osgi.app.dao.DataAccess"/>
      <efp:name="response-time" gr_id="1">
        <efp:values>
          <efp:lr id="2" value="average" />
        </efp:values>
      </efp:name>
    </provide>
  </service>
</implementation>
</component>
```

According to the principles mentioned already for Spring, the evaluation process of Bundles enriched with EFPs would take part in a component life-cycle. OSGi provides a `BundleListener` which may be used by a Bundle to observe changes (starting, stopping, installing, etc.) of other Bundles. Hence a Bundle invoking the EFFCC modules for each Bundle will determine compatibility in the phase of starting or installing other Bundles [18].

5.5.3 EFPs in CoSi

CoSi [17] is a research component model developed in a respect to several motivations: (1) strong black-box components, (2) minimal feature specification, (3) very simple infrastructure (no distribution, remoting, security, dynamic updates, ...), (4) support of weakly typed languages (Groovy). In addition, the model follows ideas from OSGi with components represented as Bundles and provided and required elements stated in the manifest file. Whereas the distribution form is the same as the OSGi form, some shortcomings of OSGi has been eliminated as much as possible.

The main mean to have a strong black-box component model is achieved so that every provided and required elements to be bind must be explicitly defined in the manifest (`manifest.mf`) file. The most significant distinction from OSGi are definitions of provided and required services on the level of the manifest file. Whereas OSGi allows to export and import packages with a set of service, CoSi requires all services to be explicitly imported and exported (although OSGi has brought the concept of declarative services). In the binding process, only these services are connected and no other services are accessible. Other features: types, attributes, events must be explicitly specified in the manifest in order to be accessible as well.

Since a complete Bundle contract is defined in the manifest file, it is also worth putting the EFP definitions here. The same way as it has been introduced for OSGi, the EFP data mirror are stored in the `META-INF` directory in a form of a XML file while the linkage of the data to concrete features goes to the manifest file.

First, the manifest file has been extended to hold EFP information. A simple extension of the header definitions is defined by the grammar:

```
header ::= header-def (’,’ efp-assign)*
efp-assign ::= ’efp=’ gr-id ’.’ efp-name ’.’ efp-value
efp-value ::= direct-value || lr-value || formula-value
direct-value ::= ’V{’ id ’}’
lr-value ::= (’LR{’ lr-id ’.’ efp-value-name ’}’)*
formula-value ::= ’C{’ id ’}’
efp-value-name ::= String
gr-id, lr-id, id := Integer
```

`header` and `header-def` are original headers of the CoSi manifest file [16] that are extended of the EFP assignment `efp-assign` option. Each EFP assignment consists of a GR identifier `gr-id` a name of an EFP `efp-name` and a concrete assigned value `efp-value`. According to three types of existing values, the manifest may hold a direct `direct-value`, LR `lr-value` and computed `formula-value` value. Since direct or formula values may have a long form, they have only an identifier `id` in the manifest file while the value itself is stored in the XML data mirror. A LR value has assigned its name `efp-value-name` and an identifier `lr-id` of LR this name comes from.

For instance, a line of the manifest file with a `response_time` property and a formula assigned to the `PersistenceIf` service may look like:

```
Provide-Services:
 cz.zcu.kiv.cosi.example.inventory.inventorydata
 .PersistenceIf;efp=(462.response_time=C{3})
```

Secondly, a plug-in for the EFP Assignment module has been developed to use the module in connection with CoSi. The plug-in is capable of unpacking a CoSi Bundle to read the XML data mirror and the EFP data from the manifest. It in practice respects component model dependent and independent part of the EFP Assignment module as it has been already explained. The plug-in implements pre-prepared interfaces to open and modify the EFP data. The implementation itself will not be detailed here.

Finally, the evaluation of EFPs on Bundles has been implemented. CoSi has a rich support of listeners to observe a life-cycle of each Bundle. Therefore, a suitable way is to call the EFP evaluation for each bundle installed in the system. As a result, a simple Bundle that listens Bundles life-cycle and calls the EFP Evaluator has been implemented. The implementation is straightforward:

```
public class StartEvaluationListener extends BundleListener {

  private BundleContext context;
  public StartEvaluationListener(BundleContext context) {
    this.context = context;
  }

  synchronized public void bundleChanged(BundleEvent event)
    throws BundleListenerException {

    if (event.getType() == BundleEvent.AFTER_INSTALL_OK) {
      Integer lrId = loadLR();
      SystemService service = (SystemService) context
        .getService(SystemService.class.getName());

      List<String> components = toList(service.getBundles());

      MatchingFunction mu = MatchingFunction.DEFAULT_MATCHING_FUNCTION;
      List<EfpEvalResult> results = EfpComparator.evaluate(
        components,
        CosiAssignmentImpl.class.getName(),
        mu, lrId);

     // any processing of the results
  }
}
```

The class presented in the code implements `BundleListener` and for each installed bundle it calls the EFP evaluation. The LR identifier `lrId` is loaded from a configuration first, then a list of components running in the system is obtained and transformed to a list of Strings. Finally, the evaluation is called. Due to the fact the design of CoSi does not allow to observe which Bundle features are bound together from the outside, the default μ function from Section 4.5.1 is used. However, its usage is sufficient for the CoSi binding mechanism.

To sum up, this section has shown the ability of EFFCC to extend existing component systems of extra-functional property support with only a little of code to implement. The support is added without need to change or re-compile the original systems and the EFPs may be enabled or disabled as it is required.

Chapter 6

Case-study: A Web Browser Product Line

Providing a complex approach, this section demonstrates a sample case-study. The case-study detailed in this section has been inspired by the Nokia's successful adaptation of the product line developing their web browser's family [51]. While the Nokia product line consists of several software modules written in C, the case-study presented here has been prepared as a set of re-usable components. However, the rationale of using the product line is the same in both cases. The first goal is to have a set of components providing fundamental functionality needed in the field of web browsers, then a set of browsers in different modifications may be developed.

6.1 Overview

Building the case-study, a typical structure of web browsers [43] has been used to create fundamental components that the browsers consist of. The components with their bindings are shown in Figure 6.1. In a complete implementation, the components would probably be more complex with other interfaces. However, several simplifications have been taken to keep simplicity and clarity of the example. Moreover, a few components (e.g. the connection to the network and the data storage) have been avoided not to complicate the example.

The components named *HTML*, *JS* and *CSS* with their respective interfaces *HTMLParser*, *JSParser* and *CSSParser* provide functionality to process HTML, CSS and Java Script elements of web pages from their textual to program data representation. In addition, the *JS* component allows to interpret Java Script code.

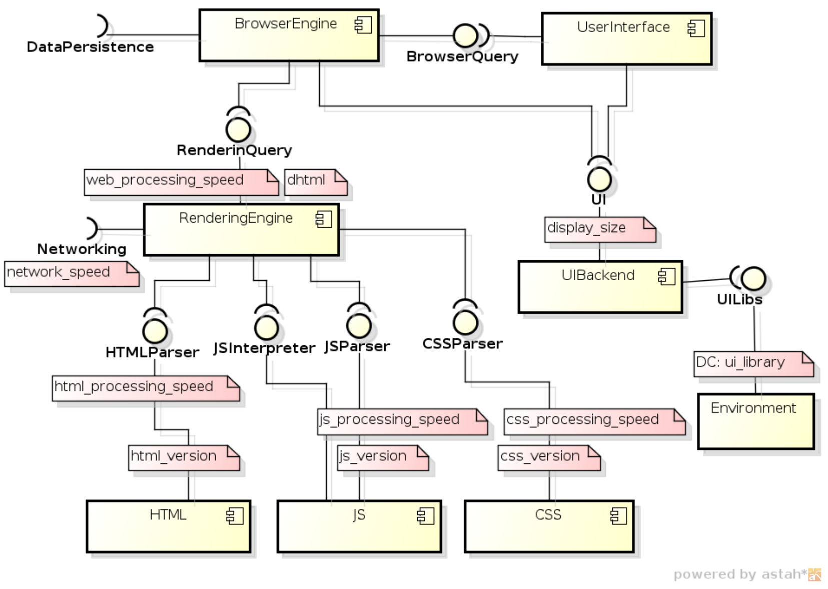


Figure 6.1: Product Line Components

Using these three components, the component *RenderingEngine* loads HTTP data via the *Networking* interface, processes HTML, CSS and Java Script elements of the web pages and creates its program data representation. The processing of the pages is queried through the interface *RenderingQuery* called by the *BrowserEngine* component to obtain the pre-processed web pages. While the *BrowserEngine* component provides a high-level engine to process the web pages in a form of the program data, the connected component *UserInterface* turns this data into a user interface. The connection is created via the *BrowserQuery* interface. The component *UIBackend* provides a low-level library allowing *UserInterface* to create complex user interface calling the *UI* interface. The box labelled *Environment* is not actually a component but it represents the environment in which the components run. Its purpose is to show a possibility to model dependencies of components on their runtime.

The presented components may create different web browser applications replacing or customizing each component. For instance, a web browser for desktop PCs would use a different *UserInterface* component than a web browser for mobile devices. However, components parsing HTML, CSS and Java Scripts would remain the same. Having a set of components with different characteristics, browsers e.g. for PCs, mobile phones or tablets may be created with the advantage of sharing common components.

6.2 Extra-functional Properties

A variety of different browsers requires different characteristics. For instance, a page rendering speed would differ among devices. A lower speed would be expected on a small resource limited device while a faster speed would be expected on a desktop PC. On the other hand, a lot of requirements would not vary. For instance, supported W3C standards put the same importance on all devices.

As an example, several EFPs expressing some important characteristics in the web browser domain have been created. The EFPs are shown in Figure 6.1 using the UML notes on respective interfaces. According to the examples mentioned above, groups concerning EFP applications and types have been stated. These groups are summarised in Figure 6.2. Although there may be likely found other EFPs for this domain, the selected EFPs aim at demonstrating the advantage of the approach in modelling different EFPs based on their concrete application.

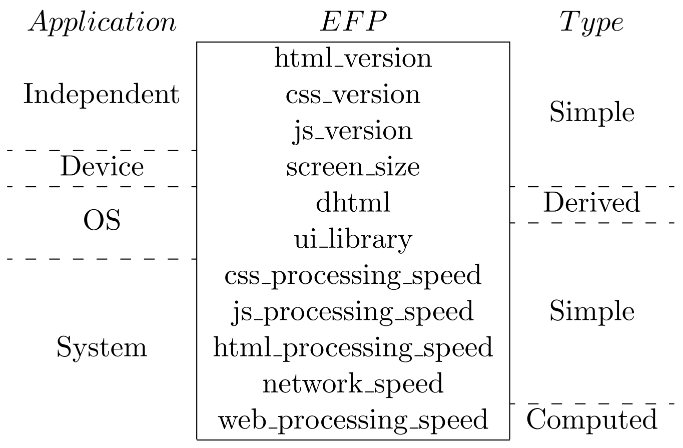


Figure 6.2: Product Line EFPs

Let us start with the group of EFPs following their application to a concrete context of usage.

Independent The first area concerns EFPs that do not vary in their applications. Their values remain the same for the whole domain of web browsers. As the members of this group, the properties *html_version*, *css_version* and *js_version* have been created to represent supported HTML, CSS and Java Script versions respectively. Obviously, values of these EFPs would be the same e.g. for a desktop PC as well as a mobile phone and thus are application independent.

OS The second area concerns EFPs differing among operating systems. As a member of this group, the property *ui_library* has been created. More precisely, this EFP is a deployment contract (DC) requiring a concrete graphical library to draw UI elements. This property differs among OSs since different OSs provide different libraries. For instance, Win32 API may be used on MS Windows while the GTK or Qt library may be used on Linux. In addition, *dhtml* is also an OS depended EFP. It is a derived property depending on *ui_library* and stating whether a browser handles dynamic HTML pages. Since the dynamic pages require both the Java Script support and a graphical UI library to paint dynamic elements, the *dhtml* property depends on concrete OS capability.

System The third area concerns EFPs differing among systems and system performance. The EFPs of this group are *css,js,html,web_processing_speed* and *network_speed.* The meaning of the first four properties is to express a speed of processing CSS, Java Script, HTML and a complete web page respectively. The last property expresses the network speed also in Mb/s. Essentially, values of these properties vary for different systems. For instance, an ARM processor based mobile device is slower then an Intel i5 2.6GHz processor based desktop PC. On the other hand, both devices may be considered fast from a user point of view, though absolute performance values differ.

Device The last area concerns different devices a web browser may run on. The EFP of this group is *screen_size* denoting the display size of a device. The meaning and the scale of such a property also vary among different applications. For instance, a 24″ LCD monitor may be considered high the same way as a high 10.5″ screen of a tablet. On the other hand, 10.5″ screen of a LCD monitor would be considered very small and for that reason these values must be treated separately for a concrete application.

The other group follows types of EFPs defined in this work. They are context-of-usage independent and depended values. Moreover, they are EFPs with directly assigned values, values computed from deriving properties and values computed by formulas assigned on components. The distribution of the presented EFPs to these categories is denoted by the third column of the table in Figure 6.2.

In terms of this work, different EFP applications are encapsulated in the system of registries (Section 4.4.1). The tree structure of registries developed for the browsers product line is shown in Figure 6.3. The vertexes of the graph denote registries while the edges registries inheritance. The bottom part of the graph is a legend mapping different applications to the tree registry structure above the legend. It divides the graph into vertical layers

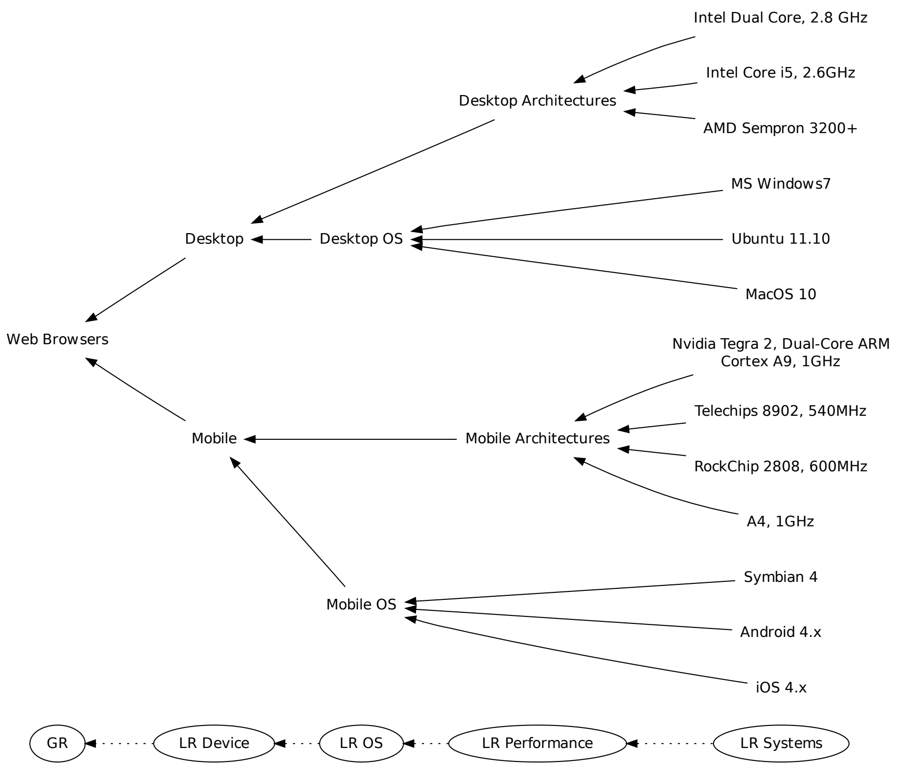


Figure 6.3: Product Line Registries

where the left one shows GR and other layers show LRs for the applications.

Web Browsers is the name of GR defining all EFPs for the browser domain. Two sub-domains for *Mobile* and *Desktop* applications are created as two LRs representing different kinds of devices. Whereas these two LRs contain values for the whole area of mobiles and desktops respectively, other specialised sub-domains are created to have values for *Mobile OS*, *Desktop OS*, *Mobile Architectures* and *Desktop Architectures*. The *Mobile OS* and *Desktop OS* hold values for different operating systems while *Mobile Architectures* and *Desktop Architectures* hold values for different systems. Such a distribution fits the distribution shown in Figure 6.2. Since EFP values for different operating systems as well as different architectures may widely vary, a set of fine-grained LRs complements the graph. Their names (e.g. *Intel Dual Core, 2.8 GHz* or *Android 4.x*) should be self explaining.

6.3 Distribution of the Properties to Registries

This section will show EFPs defined in GR and values assigned in LRs. A pseudo-code will be used for expressing the definitions in a textual form.

The EFPs must be first defined in GR in terms of their names, data types and value names. In the example, it is assumed all EFPs use the default comparing function:

```
GR: name "Web Browsers"

ui_library:  enum {GTK, Qt, Win32API, curses},
             names {"graphical", "textual"}

html_version,
css_version,
js_version: number

html_processing_speed,
css_processing_speed,
js_processing_speed,
network_speed : number-interval, unit "Mb/s",
                names {"fast", "average", "slow"}


display_size: number, unit "inches",
              names {"high", "average", "low"}

dhtml: boolean, derived {js_version, ui_library}
```

Secondly, a set of LRs containing concrete values is defined. Only a sub-set of definitions is provided to prevent long listings difficult to read.

The LRs for Device-dependent values contain values of the *display_size* property:

```
LR: name "Mobile", parent-gr "Web Browsers"
  display_size : low [0..5), average [5..9), high  [9..15)

LR: name "Desktop", parent-gr "Web Browsers"
  display_size : low [10..18), average [18..21), high  [21..30)
```

The LRs for OS-dependent values may, in short, contain values of the *ui_libray* and *dhtml* properties. The *dhtml* property is in GR generally defined according to a function $f : js\_version \times ui\_library \rightarrow dhtml$ that is in this example implemented as a logical formula:

```
LR: name "Desktop OS", parent-lr "Desktop"
  dhtml: true <=> js_version > 0.0 && ui_library == graphical
  ui_library: null # to override

LR: name "Linux", parent-lr "Desktop OS"
  ui_library: graphical {Qt, GTK}, textual {curses}

LR: name "MS Windows7", parent-lr "Desktop OS"
  ui_library: graphical {Win32API}, textual {}

LR: name "Mac OS 10", parent-lr "Desktop OS"
  ui_library: graphical {AppKit}, textual {curses}
```

The last group of LRs contains the values for different System configurations. The values presented in this example cover performance characteristics that may have been obtained e.g. from simulation and testing and thus they are valid for the same or a similar configuration as a configuration described by each LR. For instance, a shortened list of LRs may look like this:

```
LR: name "AMD Sempron 3200+, 1GB RAM",
    parent-lr "Desktop Architecture"

  html_processing_speed :
    slow [0..5), average [5..50), fast [50..+inf)
  css_processing_speed : ...
  js_processing_speed : ...
  web_processing_speed : ...

LR: name "Intel Core i5, 2.6GHz, 4GB RAM",
    parent-lr "Desktop Architecture"

  html_processing_speed :
    slow [0..10), average [10..100), fast [100..+inf)

LR: name "Telechips 8902, 540MHz",
    parent-lr "Mobile Architecture"

  html_processing_speed :
    slow [0..1), average [1..10), fast [10..+inf)

LR: name "Nvidia Tegra 2, Dual-Core ARM Cortex A9, 1GHz",
    parent-lr "Mobile Architecture"

  html_processing_speed :
```

```
slow [0..5), average [5..50),  fast [50..+inf)
```

6.4 Component Evaluation and Binding

Having EFPs and values defined in registries, the values may be assigned to existing components expressing their capabilities. In the following part of this section, four sample component bindings, which may occur according to the four applications will be presented. The examples target the four mentioned applications from Figure 6.2.

- *Independent* – Let us have a request to create a modern HTML 5 compliant web browser. Such a browser must have the *RenderingEngine* component capable of processing HTML in the version 5, CSS in the version 3 and Java Script in the version 1.8.5. For that reason, the required interfaces *HTMLParser*, *JSInterpreter*, *JSParser* and *CSSParser* would have attached the EFPs *html_version*, *js_version*, *css_version* to their respective interfaces with respective version values. The *RenderingEngine* component may guarantee a handling of modern HTML 5 compliant pages as long as the bound components provide the interfaces with EFPs with the required version information. For instance, Figure 6.4 shows two *HTML* components where the *HTML-A* is compatible while *HTML-B* incompatible. As a result, *HTML-A* may be bound while *HTML-B* cannot.

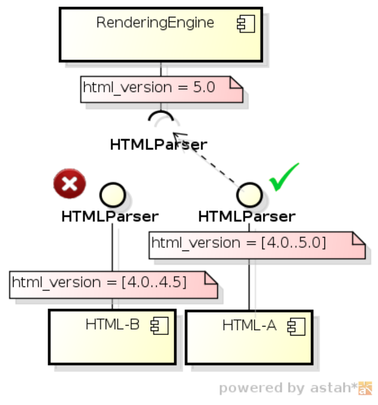


Figure 6.4: Example Binding with HTML Version

- *OS dependent* – A light-weighted text-only web browser may want to be created for system administrators accessing remote configuration of network devices in which configuration menus are created as simple textual web pages. Such a system does not need the *JS* component at all and the *UIBackend* component requires only a textual library from its runtime. For that reason, *UIBackend* requires a value *textual* (library) assigned to the *UILibs* interface via the *ui_library* EFP. Since network devices such as routers often run on Linux, the implementing library may be in this case *curses*. Example of a component binding in Figure 6.5 shows two environments for Linux and MaxOS where both can be bound to the *UIBackend* component because they provide the needed textual libraries.

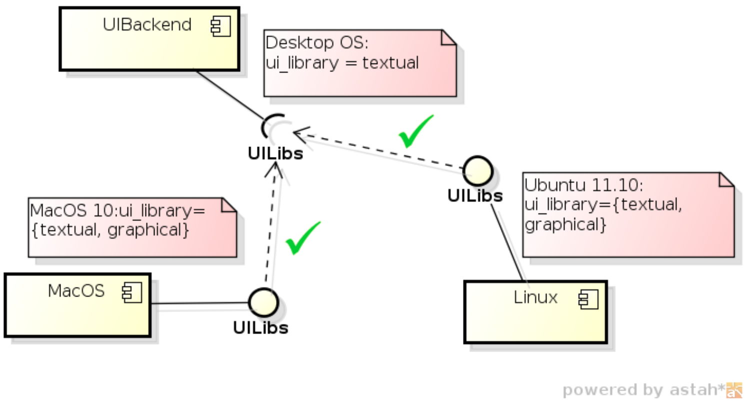


Figure 6.5: Example Binding with UI Library

- *Device dependent* – Considering the cost of developing a web browser, it may be decided to support only the most common devices with an average screen size. For that reason, the *UIBackend* library would provide the *display_size* EFP via its *UI* interface having the value *average*. In this case, web page rendering will be optimised for 18″ - 21″ desktop screens and 5″ - 9″ mobile screens only. A binding to a *UserInterface* component requiring a different screen size is inappropriate. It is shown in Figure 6.6. *UIBackend-A* is bound while *UIBanckend-B* is not recommended for the binding since the binding would cause an incorrect page view.

- *System dependent* – The most complex characteristics represent performance based on the system configuration and input data. For a concrete *RenderingEngine* component, it may be estimated that a speed to process one HTML page depends on a speed of HTML, CS and JS parsing together with loading of all resource (such as image, video or music data) of the page. Such a dependency may be addressed from a simple to a complex point. Although the goal of this work is to

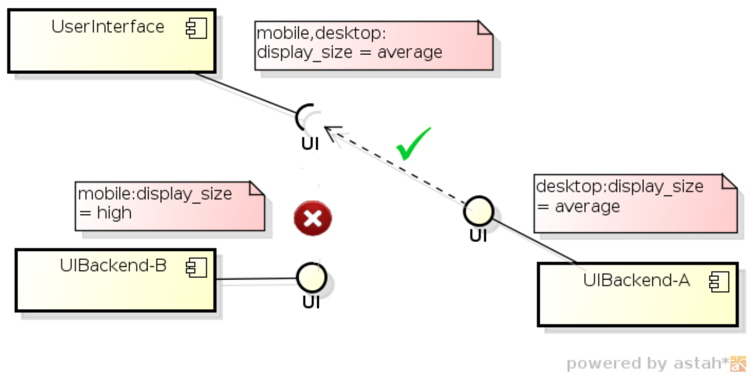


Figure 6.6: Example Binding with Display Size

propose a general mechanism to define complex EFP formulas rather than defining the formulas themselves, some possible approaches to this concrete problem will be shown in this section.

EFP Formulas: A Simplified Example

For a simpler writing, let us rename EFPs so that: $n = \mathit{network\_speed}$, $h = \mathit{html\_processing\_speed}$, $c = \mathit{css\_processing\_speed}$, $j = \mathit{js\_processing\_speed}$ and $w = \mathit{web\_procesing\_speed}$.

A fundamental definition of a function computing the speed is: $f : n \times h \times c \times j \rightarrow w$. Finding a concrete formula, it may be realised that the function should consider number of CSS, Java Scripts and other resources included in a page. Naming x, y and z number of CSS, Java Scripts and other resources of a page respectively, the formula may look like this:

$$f(x, y, z, n, h, c, j) = \frac{n + h + x(n + c) + y(n + j) + zn}{2 + 2x + 2y + z}$$

The meaning is that $n + h$, $n + c$ and $n + j$ is a speed to load and parse HTML, CSS and Java Script respectively. In addition, $n+c$ and $n+j$ must be done x and y times according to the number of CSS and Java Scripts in the page and other z resources must be loaded. The resulting value is normalised by $2+2x+2y+z$ to obtain a speed unit per second. Let us note that the formula omits the time needed by the *RenderingEngine* component itself to process the data. An assumption that this component speed is not changed based on the data is taken as long as this component delegates the data processing to other components.

The most simple approach to deal with the formula is to expect ideal components that process all pages with the same speed. In that case the *HTML*, *CSS* and *JS* component would claim their speed independently of a page complexity. In other words, the complexity of a page does not impact the speed. As a result, h, c and j are simple independent variables measured as numbers. Since the page complexity does not matter a formula may treat x, y and z as constants. For instance, $x = y = z = 1$. Taking it all together, the formula is simple:

$$f(n,h,c,j) = \frac{n + h + (n + c) + (n + j) + n}{7} = \frac{4n + h + c + j}{7}$$

For instance, *HTML*, *CSS* and *JS* components provide their speed 1000, 3000, 4000 "Kb/s" respectively for interfaces *HTMLParser*, *CSSParser* and *JSParser*, the network speed is "10Mb/s", then the *RenderingEngine* component claims to provide web pages with the speed:

$$w = \frac{4 \times 10^3 + 1000 + 3000 + 4000}{7} = 6.85 \times 10^3$$

This means that the binding of the *RenderingQuery* may be done only for an appropriate required side of the *BrowserEngine* component requiring the same or faster speed.

Assuming these values have been measured for LR "Intel Core i5, 2.6GHz, 4GB RAM", the respective value name pairs are loaded from this LR. Comparing the values with their assigned names, the speeds to parse HTML, CSS and JS would be considered *slow* and the network speed *average*. The resulting web processing speed on *RenderingEngine* is *slow* meaning that the only *BrowserEngine* component requiring the *slow* web processing speed may be bound. Such an example is shown in Figure 6.7 where *BrowserEngine-A* can be bound while *BrowserEngine-B* cannot be bound because it requires a faster speed.

EFP Formulas: A Realistic Example

Due to the simplicity of the previous evaluation, its application is limited to idealised components. In practice, components are likely to differ in a speed according to a page complexity. The complexity may be e.g. measured in terms of a number, a type and a size of web page objects proposed in [24]. Moreover, this complexity may not be linear and even not increasing. One component may be fast for processing simple pages and slow for complex ones while another component may be optimised for better performance on the complex pages instead of simple ones. The suggested solution to this problem may be in using different speed values for different page complexity. Still, the approach presented here assumes the components are enriched with

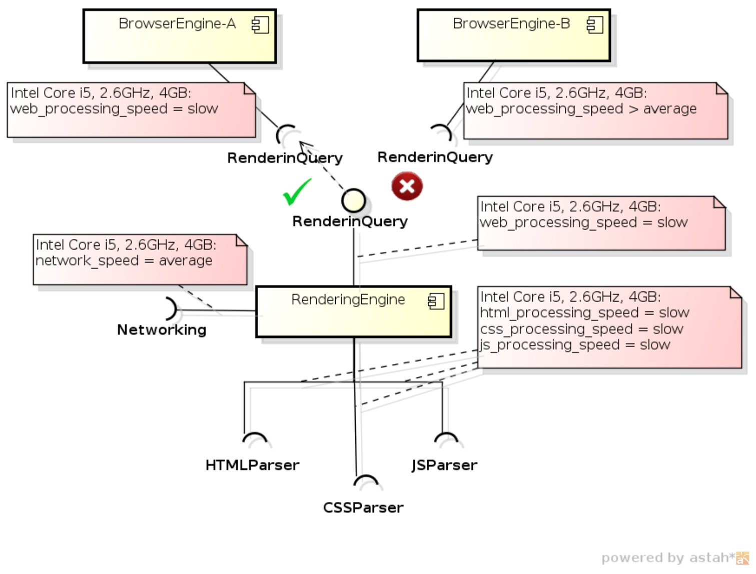

BrowserEngine-A
BrowserEngine-B
Intel Core i5, 2.6GHz, 4GB:
web_processing_speed = slow
Intel Core i5, 2.6GHz, 4GB:
web_processing_speed > average
RenderinQuery
RenderinQuery
RenderinQuery
Intel Core i5, 2.6GHz, 4GB:
web_processing_speed = slow
Intel Core i5, 2.6GHz, 4GB:
network_speed = average
Networking
RenderingEngine
Intel Core i5, 2.6GHz, 4GB:
html_processing_speed = slow
css_processing_speed = slow
js_processing_speed = slow
HTMLParser
JSParser
CSSParser
powered by astah*

Figure 6.7: Example Binding with Web Processing Speed

EFPs before they run in the runtime and before real web pages are loaded. For that reason, a set of test pages concerning different complexity may be defined for measuring the components first. Then the component claimed speeds must be related to the set of the testing data.

Defining speed values for different web pages, a total number N of testing pages is required where each page has a different complexity. Applying any simulation and measurement for these N pages, a matrix $A = [a_1, a_2, \cdots, a_N]$ contains different speeds measured for different pages. Since a model situation may contain permutations of all page counts processed with their respective speeds, the $m \times n$ matrix B is defined. The dimension m is the same as the number N and n is equal to $N!$. The first row of the B matrix has a form $B_1 = [1, 2, 3, \cdots, b_N]$ while other rows have column permutations of the first one $B_2 = [2, 1, 3, \cdots, b_N], B_3 = [3, 1, 2, \cdots, b_N]$ etc.

The multiplication of both matrices creates a $1 \times n$ sized matrix $X' = BA$. Each item of the matrix contains a (denormalised) speed combining all page counts with their respective speeds. E.g. an item X'_{11} denotes a speed to process a page containing one element with the speed a_1, two elements with the speed a_2, three elements with the speed a_3 etc. The next element contains the speed of different permutation of page counts. Having a sum $s = \sum_{i=1}^{N} i$ a normalised speed matrix is computed:

$$X = \frac{1}{s}BA = \frac{1}{s}\begin{bmatrix} 1 & 2 & 3 & \cdots & b_{1,N} \\ 2 & 1 & 3 & \cdots & b_{2,N} \\ 3 & 1 & 2 & \cdots & b_{3,N} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ b_{N!,1} & b_{N!,2} & b_{N!,3} & \cdots & b_{N!,N} \end{bmatrix} \begin{bmatrix} a_1 & a_2 & a_3 & \cdots & a_N \end{bmatrix}$$

Considering also a network speed taking a role in the evaluation, it may be again assumed the speed depends on a page loading and its processing (parsing). For that reason, a matrix $A_N = [n_1, n_2, \cdots, n_N]$ is created containing speeds measured for the same pages and the same order as it is in the matrix A. A new (denormalised) matrix is computed considering two operations to process a page: $X' = B(A + A_N)$. Since two operations are needed, a new normalising sum must be computed: $s = \sum_{i=1}^{N} 2i$ and a normalised matrix is $X = \frac{1}{s}B(A + A_N)$.

Using the same rationale as in the simple example above, the processing of a page consists of processing HTML, CSS and Java Script and other resources. For that reason, the presented matrix computation may be separately applied to each component of the web page. Since all the page elements come through the network in a form of a plain HTTP text, the matrix A_N with the network speed values may be considered still the same.

A new set of matrices A_c and A_j is created the same way as the matrix A holding the speed for CSS and Java Scripts respectively. Other resources on the page are expected to have the same speed equivalent to the speed of the network all the time. It is practical to have a number of test CSS, JavaScripts and other resources the same as the number of test HTML pages. The benefit is that all matrices have the same sizes. Taking it together, the speed matrix is computed $X = \frac{1}{s}(B(A + A_N) + B(A_c + A_N) + B(A_j + A_N) + BA_N) = \frac{1}{s}B(4A_N + A + A_c + A_j)$ with the normalisation sum computed as $s = \sum_{i=1}^{N} 7i$.

In the binding process, the *HTML*, *CSS* and *JS* components provide matrices A, A_c and A_j with the typical speed according to a page complexity first. The component *RenderingEngine* then computes the matrix X using these matrices together with the A_N matrix provided via the *Networking* interface. Furthermore, the matrix X must be compared with a requirement on the interface *RenderingQuery*. It would be probably impossible to find an exact match of the required and provided matrices X in terms of comparing each value. Instead, a difference of these two matrices may be computed first. Then a result with a difference under a certain threshold would be successfully matched. A similar approach concerning matrix differences computation has been used in [101] where the lowest difference matches, however, without considering a threshold.

To sum up, this section has introduced a case-study from the area of web browsers. The overview of the practical application of registries has been demonstrated. The case-study has shown that registries may be beneficial in the product lines where several related products are created from one group of components. This means that each registry may describe a sub-domain of each product or application. It has been demonstrated using registries following device, operating system and performance dimensions. One of the limitations is that a considerable amount of registries may have to be created. This issue is, however, addressed by implementing a rich tool support. In addition, the case-study has shown the application of EFPs concerning both simple and complex evaluation. Whereas the simple evaluation concerning constant value evaluating has been well developed, the complex one concerning comprehensive formulas has been only partly developed as an example. A question that needs to be asked in the future is whether the complex evaluation could be generalised to a more unified process.

Chapter 7

Evaluation and Future Work

7.1 Evaluation

The approach presented in this work is divided into three main modules where each of them is innovative in a certain way.

First of all, let us mention the extra-functional properties repository. Obviously, other approaches targeting extra-functional properties must also include means to store extra-functional properties. For instance, Palladio [13] or ProCom [92] store extra-functional property definitions that are later instantiated and applied to concrete components. Comparing it with our work, they store properties usable among different instances of components, but only for their own component models. The repository we have developed goes farther in its versatility and it provides the unified form of extra-functional properties available to a broad range of existing systems. Moreover, the repository does not target exclusively components. Its extra-functional properties may be used in other systems, e.g. services in SOA. We assume that the repository provides a faster adoption of extra-functional properties since a lot of existing approaches may instantly start to use them.

Moreover, practical applicability of the repository has been proved by its prototype implementation. Since the implementation uses the state-of-the-art technologies based on JavaEE providing the repository data via web services, it is actually usable by a lot of modern applications that are able to make up a client according to WSDL files. In addition, a detailed structure of the repository in a form of formalisation and models allows to re-implement the repository to other technologies.

The repository design and the structure of extra-functional properties have been inspired by other approaches. In a nutshell, the named values from HQML [44], definitions of different values for different context-of-usage from SLang [63], simple and derived properties from NoFun [36], the extra-functional properties structure and basic data types from CQML [3], value intervals from the CQML extension [104], dependencies of a component system and its runtime from Deployment Contracts [68] and a lot of other approaches mentioned in Section 3 have been used. From this point of view, the repository and the extra-functional property structure is not a brand new concept, though it is innovative in its consolidation of a lot of other approaches with the new concept of sharing extra-functional properties apart from components and concrete component models. To the best of our knowledge, there is no other approach of such a repository.

One of the limitations of this work may be seen in the simple structure of extra-functional properties. This simplification probably cannot model the properties in a comprehensive manner such in QML/CS [103]. On the other hand, we believe our approach is sufficient for a lot of practical applications as it has been demonstrated in the case-study.

This work has also the addressed applicability of extra-functional properties to different domains. To deal with the domains, the system of layers in the repository has been proposed. The layers hold domain dependent properties and their values in tree hierarchies. Although the layers allow to define different values for different environments, the scalability of the approach has been only partially targeted. As a partial solution, the layers may be aggregated with one another. Although this aggregation allows to re-use definitions from existing layers, it probably cannot prevent the creation of considerably complex structures for components used in a lot of domains.

Another problem addressed in this work is quite a slow adoption of existing extra-functional property approaches to industry. Although the existing approaches are often quite complex with the broad range of functionality, they are not adapted by the industry. For instance, in the world of Java, the widely used industrial frameworks such as OSGi or Spring have no support of extra-functional properties. They use only a few specialised means to express extra-functionality. For instance, *capabilities* in OSGi or JSR 305 [46] implemented in Spring. They are still very simple comparing to the existing research approaches.

Dealing with the slow adoption, we have created a system transforming extra-functional properties between the repository, a concrete component model and sending the properties to their processing. The system includes a customisable EFP Assignment module.

The innovative part of the EFP Assignment module is its distribution to component model dependent and independent sub-modules providing applicability of extra-functional properties to a variety of existing component models. We assume it helps a better applicability of extra-functional properties to industry thanks to its transparent and easy-to-use mechanism. Its benefit is instant applicability. To our best knowledge, there is the only similar solution Q-ImPrESS[1] which targets research component models while we target industrial ones.

Although the EFP Assignment module provides applicability of the approach to a wide range of existing systems, it still has some limitations. It is assumed that this approach is used for component models with components bound via required and provided elements and components are independently deployable units. The approach is not targeted at systems where components serve as units of architecture and the final application is monolithic.

Reaching practical applicability, the approach uses several simplifications. It is assumed that extra-functional property semantics is implicitly known from a domain without formal definitions, type-based binding is preferred in the evaluation process, approximated EFP values are used. These simplifications may cause the weak usability of the approach in some applications. Mainly the applications where a precise system behaviour or detail system models are needed should use specialised approaches. On the other hand, the simplifications were introduced to have as easy-to-use approach as possible, still keeping benefits of acquiring extra-functional properties in applications.

The structure of the data provided by the EFP Assignment module includes basic values, context-dependent values as well as functions allowing to compose extra-functional properties. The work presented here has invested only little attention to complex compositions of the properties that may cause problems when complex properties are needed. Since Crnkovic [29] mentioned in his work the complexity of the extra-functional property composition, this work proposes a generic mechanism. This mechanism allows to implement wide varieties of the compositions. However, an approach to implement particular functions in an easy manner should be also provided. This work have proposed prototype implementations of mathematical formulas such as those used in CQML+ [90] or by Defour [32]. Since these formulas suit only simpler cases, complex ones must be additionally implemented in the future.

The EFP Assignment module provides the extra-functional properties in one unified form. The processing of the properties is easier and implementation of evaluators is straightforward due to the same understanding of the properties among different systems.

This work contains the implementation of the EFP Assignment module to existing frameworks to prove its overall goal – its applicability to a variety of other systems. The application of the implementation is, however, limited

[1]Available at: http://www.q-impress.eu/wordpress/ (2012)

only to Java based programs. This work provides the rich formal and model approach that should allow re-implementation to other systems.

The last module of the approach is the EFP Evaluator. The evaluator tries to minimise computational and model complexity that may be found in other approaches. For instance, in QML/CS [103] a developer is required to create a considerable wide set of models to define each extra-functional property. These models provide the advantage of quite accurate results in the process of the binding and evaluation. In our work, we propose the type-based approach that compares only extra-functional properties attached to communicating interfaces. The benefit of this solution is a fast computation without the need of complex models. This solution should provide a simple approach that developers may easily acquire.

A question raised with the type-based approach is where to obtain extra-functional property values that are attached to component interfaces. This work assumes the use of simulations (e.g. the simulation system proposed by Potužák [87]) for obtaining approximated values. On the one hand, the weakness of this solution is that it provides only approximated results. On the other hand, the benefit of the fast computation and no complex model needs is considerable. Avoiding this weakness, the extra-functional property repository holds intervals of values divided to several sub-intervals. The evaluation of the values consist of fitting a computed result to an interval. This solution is probably impractical in specialised areas such as hard real-time applications where specialised approaches would suit better. On the other hand, using this solution, a user may check if a value overcomes a certain threshold. It should be helpful in a lot of practical applications.

This idea has been also mentioned in Hamlet's work [45]. In his work he recommends to rely on measured and simulated values rather than describing component's behaviour using complex models. The citation reads:

> Mathematical, analytical methods are in principle entirely correct, but difficult to apply in practice, where testing and measurement [simulation] are the analysis methods of choice.

This work allows to model graph dependencies of extra-functional properties. The dependencies are modelled by already mentioned functions. The problem of the graph dependencies is that one incompatibility may require to change several components among the graph. It is a considerable drawback that has been only partly solved in this work. This work has proposed a heuristic method allowing to create a graph of compatible component in the polynomial time. This method is inspired in Lau's incrementally composed component model [67].

The evaluator mechanism can participate in the component binding of an underlying component framework. The advantage is that the component graph used by the evaluator is the same as the component graph generated by the component framework. This approach is, however, not usable for component frameworks that do not allow to observer their component binding process. In such a case, the evaluator may use its default binding to generate the graph. As a consequence, the produced results are limited in their reliability. The reason is that the graph may differ from the real component binding.

7.2 Future Work

This work has introduced a complete approach to deal with extra-functional properties among a variety of industrial as well as research component models. However, several improvements of the approach are left for the future work. The possible improvements touch both technical (implementing) and research topics.

From the technical point of view, the approach has been implemented in Java and applied to three component models: OSGi, Spring IoC Container and CoSi. Currently, the application to Component Repository supporting Compatibility Evaluation (CRCE)[2] is in progress. CRCE is an advanced repository of OSGi bundles allowing to verify the components in the repository so that all components obtained from the repository are verified and compatible. The idea behind the CRCE approach is that resource constrained devices do not have to run resource consuming verification. For that reason, the verification is pre-computed on the repository level. It is the role of the mechanism presented in this work to enrich CRCE with extra-functional property verifications and it should be finished in the near future.

In addition, the application of the mechanism to other component models is also desired. Despite the fact that the implemented prototypes have already shown directions of how to use the mechanism, a routine usage should be supported by implementations for a wide set of other component models. It consequently verifies the approach is generic enough. For instance, support for PicoContainer[3], ArchJava[4] and even pure Java .jar files is worth implementing in the future. Moreover, re-implementation for other programming languages is also worth considering.

Since the registry implementation is still a prototype proof-of-the-concept, it does not deal with aspects such as user privileges, history or versioning.

[2]Available at: http://www.assembla.com/spaces/crce/ (2012)
[3]Available at: http://picocontainer.org/ (2012)
[4]Available at: http://archjava.fluid.cs.cmu.edu/ (2012)

Once registries are applied in practice, it will most probably require setting up an organisation structure determining persons and their roles for the registries. It is our current ongoing implementation attempt that will be finished in the near future.

Considering a long term usage of registries, they should be able to capture any change requests. There will probably be situations when the values of registries are to be changed. In these situations, a value cannot be simply re-written because it may be used elsewhere. For that reason, versioned values would allow to go back in the history.

Apart from these technical requirements, there are several research issues that have not been finished yet.

The first issue is related with a way the EFP values are put into registries. Currently, users must manually input values for every named value and every registry. Despite the strong tool support, it is still a slow process. However, a lot of kinds of values can be computed according to formulas. The rationale is that users often need to put values in scales or ranges rather than exact values.

For instance, a *network speed* property may be considered *slow*, *average* and *fast* for respective values 10, 100, 1000 "Mb/s". It most probably has no sense to put not rounded values such as 9, 101, 999 respectively in registry. As a result, a user may have to be asked to input only a formula (e.g. $x_i = 10x_{i-1}$ starting from $x_{i-1} = 1$ in this example). Assuming these values are valid for a registry with *LAN Ethernet*, it may be required to put values for *GPRS* registry with values 1, 10, 100 "Mb/s". Therefore, formulas re-computing values among registries could also be used with the same benefit. As a result, user would be asked only to input several functions with initial values and all other values for all registries would be automatically computed leading to better user comfort. Application, evaluation and testing of such features are, however, left for future improvement.

The EFP mechanism allows the usage of formulas for EFPs. However, the formulas have been presented on an abstract level with some practical implementation presented in the case-study. The abstract level provides a fairly unlimited spectrum of formulas, though it would be practical to prepare a limited set meeting most frequent user needs. For future research, more concern will be taken to precise the formulas and implement most typical ones.

The case-study has also shown one possible distribution of EFP values into registries. It has essentially introduced its possible practical application, though the problem may be seen in that the registries have been made up ad-hoc. The distribution to the operating system, the device and the performance sub-domains has been done by empirical observation to fit current needs. Such an approach is, however, not systematic and may prevent understandability among users. A solution may be in introducing typed registries where it is explicitly said which kind of values each registry holds.

Section 4.6.3 has discussed possible solutions to be taken once the presented mechanism discover an incompatibility. Due to the time complexity of existing general solutions, a heuristic method has been suggested. However, it has been only empirically observed that such a method may be useful in a lot of typical cases. For that reason, in our future work, a percentile number of cases in which the approach is capable of resolving an incompatibility is worth measuring.

Chapter 8

Conclusion

This work has pointed out a need for extra-functional properties to improve current component based development. It has started by a rich survey in the state-of-the-art of extra-functional properties. Despite the wide number of existing approaches, this work has secondly realised that they have been only scarcely applied in practice. It has been highlighted that such an insufficient usage of extra-functional properties prevents successful application of component based programming. As a result, a solution to overcome the discrepancy between industrial and research approaches has been suggested. The proposed solution attempts to improve the adoption of extra-functional properties in component based programming leading to better adoption of component based programming itself.

Therefore, the main contribution of this work is the introduction of an independent mechanism for working with EFPs in a comprehensive manner. To reach the independence, the mechanism consists of a few separate modules. The first module includes a repository of general extra-functional properties. Its layered approach deals with extra-functional property definitions apart from a concrete application in one layer and a set of application specific definitions in inherited layers. The second module covers an application of the general extra-functional properties to components. This module allows the application to a variety of existing component systems. It splits component model dependent and independent parts into two sub-modules. The last module allows to evaluate extra-functional properties. In addition to a generic evaluation of extra-functional properties, the evaluator is able to integrate a binding mechanism of an underlying component model.

As a result, this approach enriches, but does not limit, current industrial component frameworks. Instead, it aims at filling the gap between the application of extra-functional properties and industrially used component frameworks.

Proving its practical applicability, the mechanism has been implemented in Java, and applied to industrial Spring and OSGi component models. Furthermore, its sample application in a form of a case-study has been presented.

The Java-based implementation verifies integrity an practical usability of the presented formalisations. Together with the prototype implementation to OSGi and Spring, the goal of practical applicability has been reached.

The case-study focuses on a product line of web browsers, which provides an example of a practical application of the approach. It has also shown one of the suitable applications of the system of registries. A software vendor may produce a variety of applications based on one set of components. The vendor then prepares registries for all supported platforms and enriches the components with respective properties and values.

Bibliography

[1] IEEE 610.3-1989 standard glossary of modeling and simulation terminology. Institute of Electrical and Electronics Engineers, IEEE Computer Society, 1989.

[2] IEEE 830-1998 recommended practice for software requirements specifications. Software Engineering Standards Committee of the IEEE Computer Society, 1998.

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

[4] M. Abadi and L. Lamport. Composing specifications. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 15:73–132, January 1993.

[5] R. Aigner, M. Pohlack, S. Röttger, and S. Zschaler. Towards pervasive treatment of non-functional properties at design and run-time. In *16th International Conference on Software and Systems Engineering and their Applications (ICSSEA'03)*. CNAMCMSL, 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 *10th International Workshop on Component-Oriented Programming (WCOP) in Conjunction with the 19th European Conference on Object Oriented Programming (ECOOP)*, 2005.

[8] A. Anton. Goal identification and refinement in the specification of information systems. PhD Thesis, Georgia Institute of Technology., 1997.

[9] C. Atkinson, P. Bostan, D. Brenner, G. Falcone, M. Gutheil, O. Hummel, M. Juhasz, and D. Stoll. Modeling components and component-based systems in KobrA. In A. Rausch, R. Reussner, R. Mirandola, and F. Plasil, editors, *CoCoME*, volume 5153 of *Lecture Notes in Computer Science*, pages 54–84. Springer-Verlag Berlin, Heidelberg, 2007.

[10] C. Atkinson and D. Muthig. Component-based product-line engineering with the UML. In C. Gacek, editor, *International Conference on Software Reuse*, volume 2319 of *Lecture Notes in Computer Science*, pages 343–344. Springer-Verlag Berlin, Heidelberg, 2002.

[11] 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, Software Engineering Institute, Carnegie Mellon, 2000.

[12] J. Bauml and P. Brada. Automated versioning in OSGi: A mechanism for component software consistency guarantee. In *Proceedings of 35th Euromicro conference on Software Engineering and Advanced Applications*, pages 428–435. IEEE Computer Society, 2009.

[13] 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.

[14] B. W. Boehm, J. R. Brown, and M. Lipow. Quantitative evaluation of software quality. In *ICSE '76: Proceedings of the 2nd international conference on Software engineering*, pages 592–605, Los Alamitos, CA, USA, 1976. IEEE Computer Society.

[15] 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.

[16] P. Brada. The CoSi component model. Technical Report DCSE/TR-2008-07, Department of Computer Science and Engineering, University of West Bohemia, July 2008.

[17] 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 Berlin, Heidelberg.

[18] P. Brada. Enhanced OSGi bundle updates to prevent runtime exceptions. In *Proceedings of the 34th Euromicro conference on Software Engineering and Advanced Applications*, Parma, Italy, September 2008. IEEE Computer Society.

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

[20] E. Bruneton, T. Coupaye, M. Leclercq, V. Quéma, and J.-B. Stefani. An open component model and its support in Java. In *Component-Based Software Engineering*, pages 7–22. Springer-Verlag Berlin, Heidelberg, 2007, 2004.

[21] E. Bruneton, T. Coupaye, M. Leclercq, V. Quéma, and J.-B. Stefani. The FRACTAL component model and its support in java: Experiences with auto-adaptive and reconfigurable systems. *Software Practise Experience*, 36(11-12):1257–1284, 2006.

[22] L. Bulej, T. Bures, T. Coupaye, M. Decký, P. Jezek, P. Parizek, F. Plasil, T. Poch, N. Rivierre, O. Sery, and P. Tuma. CoCoME in Fractal. In A. Rausch, R. Reussner, R. Mirandola, and F. Plasil, editors, *CoCoME*, volume 5153 of *Lecture Notes in Computer Science*, pages 357–387. Springer-Verlag Berlin, Heidelberg, 2007.

[23] T. Bures, P. Hnetynka, and F. Plasil. SOFA 2.0: Balancing advanced features in a hierarchical component model. In *Software Engineering Research, Management and Applications*, pages 40–48. IEEE Computer Society, 2006.

[24] M. Butkiewicz, H. Madhyastha, and V. Sekar. Understanding website complexity: Measurement, metrics, and implications. In *Internet Measurement Conference, ACM/USENIX IMC*, 2011. [to be printed].

[25] J. Cheesman and J. Daniels. *UML Components - A Simple Process for Specifying Component-Based Software.* Component Software Series. Addison-Wesley, 2001.

[26] L. Chung, B. A. Nixon, E. Yu, and J. Mylopoulos. *Non-Functional Requirements in Software Engineering.* Series: International Series in Software Engineering, Vol. 5, Springer-Verlag Berlin, Heidelberg, 476 p, ISBN: 978-0-7923-8666-7, October 1999.

[27] L. Chung and J. C. Prado Leite. Conceptual modeling: Foundations and applications. chapter On Non-Functional Requirements in Software Engineering, pages 363–379. Springer-Verlag, Berlin, Heidelberg, 2009.

[28] I. Crnkovic. *Building Reliable Component-Based Software Systems.* Artech House, Inc., Norwood, MA, USA, 2002.

[29] I. Crnković, M. Chaudron, S. Sentilles, and A. Vulgarakis. A classification framework for component models. In *Proceedings of the 7th Conference on Software Engineering and Practice in Sweden.* IEEE Computer Society, October 2007.

[30] I. Crnković, S. Sentilles, A. Vulgarakis, and M. R. Chaudron. A classification framework for software component models. *IEEE Transactions on Software Engineering*, 37:593–615, 2011.

[31] A. Davis. *Software requirements: objects, functions, and states.* Prentice-Hall, Inc. Upper Saddle River, NJ, USA, 1993.

[32] O. Defour, J. Jézéquel, and N. Plouzeau. Extra-functional contract support in components. In *In Proceedings of 7th International Symposium on Component-Based Software Engineering (CBSE 7).* Springer-Verlag Berlin, Heidelberg, 2004.

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

[34] J. Feljan, L. Lednicki, J. Maras, A. Petričić, and I. Crnković. Classification and survey of component models. Technical report, DICES, 2007.

[35] A. V. Fioukov, E. M. Eskenazi, D. K. Hammer, and M. R. V. Chaudron. Evaluation of static properties for component-based architectures. In *EUROMICRO*, pages 33–39. IEEE Computer Society, 2002.

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

[37] S. Frølund, S. F. Lund, J. Koistinen, and J. Koistinen. Quality of service specification in distributed object systems design. Tech. Report, HP lab., 1998.

[38] 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, ISBN: 978-3-540-74973-8*, pages 69–80. Springer-Verlag Berlin, Heidelberg, 2007.

[39] J. M. García, I. Toma, D. Ruiz, and A. Ruiz-Cortés. A service ranker based on logic rules evaluation and constraint programming. In *2nd Non-Functional Properties and Service Level Agreements in*

Service Oriented Computing Workshop (NFPSLA-SOC'08) co-located with the 6th IEEE European Conference on Web Services (ECOWS 2008), Dublin. CEUR-WS.org, November 2008.

[40] T. Genssler, A. Christoph, B. Schulz, M. Winter, C. M. Stich, C. Zeidler, P. Müller, A. Stelter, O. Nierstrasz, S. Ducasse, G. Arevalo, Roel, R. Wuyts, P. Liang, B. Schönhage, and R. V. D. Born. PECOS in a nutshell. Pecos Handbook, Tech. report, September 2002.

[41] M. Glinz. Rethinking the notion of non-functional requirements. In *Proceedings of the Third World Congress for Software Quality (3WCSQ'05*, pages 55–64, 2005.

[42] M. Glinz. On non-functional requirements. In *Requirements Engineering Conference*, pages 21–26, Los Alamitos, CA, USA, 2007. IEEE Computer Society.

[43] A. Grosskurth and M. W. Godfrey. A reference architecture for web browsers. In *Proceedings of the 21st IEEE International Conference on Software Maintenance*, pages 661–664, Washington, DC, USA, 2005. IEEE Computer Society.

[44] X. Gu, K. Nahrstedt, W. Yuan, D. Wichadakul, and D. Xu. An XML-based quality of service enabling language for the web. *Journal of Visual Language and Computing, Special Issue on Multimedia Language for the Web*, 13:61–95, 2001.

[45] D. Hamlet, D. Mason, and D. Woit. *Component-Based Software Development: Case Studies*, chapter Properties of Software Systems Synthesized from Components, pages 129–158. World Scientific Publishing Co. Pte. Ltd., 2004.

[46] D. Hovemeyer and W. Pugh. Status report on jsr-305: annotations for software defect detection. In *Companion to the 22nd ACM SIGPLAN conference on Object-oriented programming systems and applications companion*, OOPSLA '07, pages 799–800, New York, NY, USA, 2007. ACM.

[47] 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 Berlin, Heidelberg.

[48] IEEE. IEEE standard glossary of software engineering terminology. *IEEE Std 610.12-1990*, Dec. 1990.

[49] IEEE. IEEE recommended practice for software requirements specifications. Technical report, 1998.

[50] International Standards Organisation (ISO). ISO/IEC 9126: Informational technology - product quality - part1: Quality Model, June 2001.

[51] A. Jaaksi. Developing mobile browsers in a product line. *IEEE Software*, 19:73–80, July 2002.

[52] I. Jacobson, G. Booch, and J. Rumbaugh. *The unified software development process*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1999.

[53] K. Ježek. A complex meta-model for extra-functional properties concerning common data types their comparing and binding. In *2nd World Congress on Software Engineering (WCSE 2010), Volume 2, pages: 71-74,ISBN:978-0-7695-4303-1*. IEEE Computer Society, 2010.

[54] K. Ježek. Universal extra-functional properties repository, model overview and implementation. In *Proceedings of the International Conference on Knowledge Management and Information Sharing (KMIS 2010), pages: 382-385,ISBN: 978-989-8425-30-0*. SciTePress, 2010.

[55] K. Ježek and P. Brada. Implementation of language for extra-functional properties for reusable software components. In *Proceedings of 35th Euromicro conference Work in Progress session*, Patras, Greece, September 2009.

[56] K. Ježek and P. Brada. Compatibility verification of components in terms of functional and extra-functional properties, tool support. In *Proceedings of the 12th International Conference on Enterprise Information Systems (ICEIS 2010), Volume 3, pages: 510-514, ISBN: 978-989-8425-06-5*. SciTePress, 2010.

[57] K. Ježek and P. Brada. Correct matching of components with extra-functional properties - a framework applicable to a variety of component models. In *Evaluation of Novel Approaches to Software Engineering (ENASE)*. SciTePress, 2011. ISBN: 978-989-8425-65-2.

[58] K. Ježek and P. Brada. Extra-functional properties framework with configuration based on deployment environment, tool demonstration and case-study. In *International Joint Conference on Knowledge Discovery, Knowledge Engineering and Knowledge Management*. SciTePress, 2011.

[59] K. Ježek and P. Brada. *Evaluation of Novel Approaches to Software Engineering*, chapter Formalisation of a Generic Extra-functional Properties Framework. Communications in Computer and Information Science (CCIS). Springer-Verlag Berlin, Heidelberg, 2012. [in print].

[60] K. Ježek, P. Brada, and L. Holý. Enhancing OSGi with explicit, vendor independent extra-functional properties. In *50th International Conference on Objects, Models, Components, Patterns. Lecture Notes in Computer Science*. Springer-Verlag Berlin, Heidelberg, 2012. [accepted to publication].

[61] 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 approaches to Software Components and Architectures (FESCA 2010), Electronic Notes in Theoretical Computer Science (ENTCS) Volume 264, page 55-71, ISSN: 1571-0661*, pages 55–71. Elsevier Science Publishers B. V., 10th August 2010.

[62] G. Kotonya and I. Sommerville. *Requirements Engineering - Processes and Techniques*. John Wiley & Sons, ISBN: 978-0-471-97208-2, 1998.

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

[64] L. Lamport. *Specifying Systems, The TLA+ Language and Tools for Hardware and Software Engineers*. Addison-Wesley, 2002.

[65] K.-K. Lau, P. V. Elizondo, and Z. Wang. Exogenous connectors for software components. In G. T. Heineman, I. Crnkovic, H. W. Schmidt, J. A. Stafford, C. A. Szyperski, and K. C. Wallnau, editors, *CBSE*, volume 3489 of *Lecture Notes in Computer Science*, pages 90–106. Springer-Verlag Berlin, Heidelberg, 2005.

[66] K.-K. Lau, A. Nordin, T. Rana, and F. Taweel. Constructing component-based systems directly from requirements using incremental composition. *Software Engineering and Advanced Applications, Euromicro Conference*, 0:85–93, 2010.

[67] K.-K. Lau, M. Ornaghi, and Z. Wang. A software component model and its preliminary formalisation. In F. S. de Boer, M. M. Bonsangue, S. Graf, and W. P. de Roever, editors, *FMCO*, volume 4111 of *Lecture Notes in Computer Science*, pages 1–21. Springer-Verlag Berlin, Heidelberg, 2005.

[68] K.-K. 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 Lecture Notes in Computer Science*, pages 1–16. Springer-Verlag Berlin, Heidelberg, 2006.

[69] K.-K. Lau and V. Ukis. Deployment contracts for software components. Preprint CSPP-36, School of Computer Science, The University of Manchester, February 2006.

[70] K.-K. Lau and Z. Wang. Software component models. *IEEE Transitions Software Engineering*, 33(10):709–724, 2007.

[71] B. Meyer. The grand challenge of trusted components. In *Proceedings of the 25th International Conference on Software Engineering*, pages 660–667. IEEE Computer Society, 2003.

[72] 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-Verlag Berlin, Heidelberg, 2008.

[73] M. Mohammad and V. S. Alagar. TADL - an architecture description language for trustworthy component-based systems. Technical report, Department of Computer Science and Software Engineering, Concordia University, 2008.

[74] R. E. Moore. Automatic error analysis in digital computation. Technical Report Space Div. Report LMSD84821, Lockheed Missiles and Space Co., Sunnyvale, CA, USA, 1959.

[75] M. Mulugeta and A. Schill. *A Framework for QoS Contract Negotiation in Component-Based Applications.* Springer-Verlag, Berlin, Heidelberg, 2008.

[76] 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 Berlin, Heidelberg, 2005.

[77] J. Mylopoulos, L. Chung, and B. Nixon. Representing and using nonfunctional requirements: A process-oriented approach. *IEEE Transactions on Software Engineering*, 18(6):483–497, 1992.

[78] C. Ncube. A requirements engineering method for cots-based systems development. PhD Thesis, City University London, 2000.

[79] O. Nierstrasz, G. Arévalo, S. Ducasse, R. Wuyts, A. P. Black, P. O. Müller, C. Zeidler, T. Genssler, and R. van den Born. A component model for field devices. In J. M. Bishop, editor, *Component Deployment*, volume 2370 of *Lecture Notes in Computer Science*, pages 200–209. Springer-Verlag Berlin, Heidelberg, 2002.

[80] OMG. MOF 2.0 core. OMG Document ptc/06-01-01, Januar 2006.

[81] OMG. UML profile for modeling quality of service and fault tolerance characteristics and mechanism specification 1.1. Technical report, OMG - Object Management Group, 2008. formal/2008-04-05.

[82] OMG. *UML Profile for MARTE: Modeling and Analysis of Real-Time Embedded Systems*. OMG, 2009. formal/2009-11-02, available at: http://www.omg.org/spec/MARTE/1.0/PDF (2010).

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

[84] OSGi. *OSGi Service Platform Service Compendium 4.2*. The OSGi Alliance, 2009. Available at http://www.osgi.org/Download/Release4V42 (2011).

[85] OSGi. *OSGi Service Platform Core Specification 4.3*. OSGi Aliance, 2011. Available at http://www.osgi.org/.

[86] The OSGi Alliance. *OSGi Service Platform, Release 4*, August 2005. Available at http://www.osgi.org/.

[87] T. Potužák, R. Lipka, J. Šnajberk, P. Brada, and P. Herout. Design of a component-based simulation framework for component testing using SpringDM. In *Second Eastern European Regional Conference on the Engineering of Computer Based Systems, pages 167-168, ISBN 978-0-7695-4418-2*. IEEE Computer Society, 2011.

[88] S. Robertson and J. Robertson. *Mastering the Requirements Process*. ACM Press, 1999.

[89] W. P. d. Roever. The need for compositional proof systems: A survey. In *Revised Lectures from the International Symposium on Compositionality: The Significant Difference*, COMPOS'97, pages 1–22, London, UK, 1998. Springer-Verlag Berlin, Heidelberg.

[90] 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.

[91] G. Salazar-zárate and P. Botella. Use of UML for modeling non-functional aspects, 2007.

[92] S. Sentilles, P. Stepan, J. Carlson, and I. Crnkovic. Integration of extra-functional properties in component models. In I. P. Christine Hofmeister, Grace A. Lewis, editor, *12th International Symposium on Component Based Software Engineering (CBSE 2009), LNCS 5582*. Springer-Verlag Berlin, Heidelberg, June 2009.

[93] I. Sommerville. *Software Engineering, Seventh Edition*. Pearson, 2004.

[94] Spring Comunity. *Spring Framework, ver.3, Reference Documentation*. SpringSource, ver. 3 edition, 2010. Available at http://static.springsource.org/spring/docs/3.0.x/spring-framework-reference/html/.

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

[96] C. Szyperski, D. Gruntz, and S. Murer. *Component Software - Beyond Object-Oriented Programming: Second Edition*. Addison-Wesley / ACM Press, 624 pages, ISBN-13: 978-0201745726, 2002.

[97] R. C. van Ommering, F. van der Linden, J. Kramer, and J. Magee. The Koala component model for consumer electronics software. *IEEE Computer Society*, 33(3):78–85, 2000.

[98] X. Wang, T. Vitvar, M. Kerrigan, and I. Toma. A QoS-aware selection model for semantic web services. *Lecture Notes in Computer Science*, 4294:390–401, 2006.

[99] E. J. Weyuker. *Component-Based Software Engineering: Putting the Pieces Together*, chapter The Trouble with Testing Components, pages 499–512. Addison-Wesley, 2001.

[100] K. E. Wiegers. *Software Requirements, Second Edition (Pro-Best Practices)*. Microsoft Press, 2 sub edition, 2003.

[101] J. Yan and J. Piao. Towards QoS-based web services discovery. *Service-Oriented Computing – ICSOC 2006, Lecture Notes in Computer Science*, 5472:200–210, 2009.

[102] N. K. Yap, A. Azim, A. Ghani, A. Mamat, and H. Zulzalil. The robust software metric data model defined in XML. *IJCSNS International Journal of Computer Science and Network Security*, 8(2), February 2008.

[103] S. Zschaler. Formal specification of non-functional properties of component-based software systems: A semantic framework and some applications thereof. *Software and Systems Modelling (SoSyM)*, 2009.

[104] S. Zschaler and M. Meyerhöfer. Explicit modelling of QoS-dependencies. *In Proceedings of the 1st International Workshop on Quality of Service in Component-Based Software Engineering, Cépadués-Éditions, Toulouse, France*, pages 57–66, 2003.

Appendix A

Author's Publications

The following papers were published in conference proceedings:

1. K. Ježek and P. Brada. Implementation of language for extra-functional properties for reusable software components. In *Proceedings of 35th Euromicro conference Work in Progress session*, Patras, Greece, September 2009
2. 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 approaches to Software Components and Architectures (FESCA 2010), Electronic Notes in Theoretical Computer Science (ENTCS) Volume 264, page 55-71, ISSN: 1571-0661*, pages 55–71. Elsevier Science Publishers B. V., 10th August 2010
3. K. Ježek and P. Brada. Compatibility verification of components in terms of functional and extra-functional properties, tool support. In *Proceedings of the 12th International Conference on Enterprise Information Systems (ICEIS 2010), Volume 3, pages: 510-514, ISBN: 978-989-8425-06-5*. SciTePress, 2010
4. K. Ježek. Universal extra-functional properties repository, model overview and implementation. In *Proceedings of the International Conference on Knowledge Management and Information Sharing (KMIS 2010), pages: 382-385,ISBN: 978-989-8425-30-0*. SciTePress, 2010
5. K. Ježek. A complex meta-model for extra-functional properties concerning common data types their comparing and binding. In *2nd World Congress on Software Engineering (WCSE 2010), Volume 2, pages: 71-74,ISBN:978-0-7695-4303-1*. IEEE Computer Society, 2010

6. K. Ježek and P. Brada. Correct matching of components with extra-functional properties - a framework applicable to a variety of component models. In *Evaluation of Novel Approaches to Software Engineering (ENASE).* SciTePress, 2011. ISBN: 978-989-8425-65-2

7. K. Ježek and P. Brada. Extra-functional properties framework with configuration based on deployment environment, tool demonstration and case-study. In *International Joint Conference on Knowledge Discovery, Knowledge Engineering and Knowledge Management.* SciTePress, 2011

8. K. Ježek and P. Brada. *Evaluation of Novel Approaches to Software Engineering*, chapter Formalisation of a Generic Extra-functional Properties Framework. Communications in Computer and Information Science (CCIS). Springer-Verlag Berlin, Heidelberg, 2012. [in print]

9. K. Ježek, P. Brada, and L. Holý. Enhancing OSGi with explicit, vendor independent extra-functional properties. In *50th International Conference on Objects, Models, Components, Patterns. Lecture Notes in Computer Science.* Springer-Verlag Berlin, Heidelberg, 2012. [accepted to publication]

This work contributed to the following grants:

1. The Grant Agency of the Czech Republic, grant number 201/08/0266 "Methods and models for consistency verification of advanced component-based applications"

2. The Grant Agency of the Czech Republic, grant number 103/11/1489 "Methods of development and verification of component-based applications using natural language specifications"