Available online at www.sciencedirect.com
ScienceDirect
Electronic Notes in Theoretical Computer Science
ELSEVIER
Electronic Notes in Theoretical Computer Science 264 (2010) 55–71
www.elsevier.com/locate/entcs

Towards Context Independent Extra-functional Properties Descriptor for Components[1]

Kamil Ježek Přemek Brada

{kjezek|brada}@kiv.zcu.cz

Department of Computer Science and Engineering
University of West Bohemia
Pilsen, Czech Republic

Petr Štěpán

stepanp@cs.man.ac.uk

School of Computer Science
The University of Manchester
Manchester M139PL, United Kingdom

Abstract

Architectures based on composing target application functionality from pre-existing components have been successfully used in many projects, yet there are several aspects in which they fail to reach the desirable level of maturity. Since different vendors may provide components with the same functionality, extra-functional properties must be taken into account to help the developer to select the component which suits the final system. In addition, a selected component must conform to the target deployment environment. This paper addresses the problem of inadequate means to define extra-functional properties on components in a way that allows to express component's properties with respect to different computational environments. We provide a representative survey of the current state of the art for extra-functional properties and propose a formalism based on existing approaches which addresses this inadequacy. Our formalism further allows us to use the extra-functional properties for describing the dependency between a component and a deployment environment referred to as deployment contracts. Our approach uses a system of registries that has the advantage of a consistent interpretation of the extra-functional properties found on any given component regardless of its usage context and that separates the semantics and the syntax of each property.

Keywords: software component, extra-functional property, non-functional property, composition, specification, registry, deployment contract

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

1571-0661/$ – see front matter
doi:10.1016/j.entcs.2010.07.005

1 Introduction

Nowadays competitive software companies need to deliver software in a still shorter time and with a lower price. In an opposite trend, software products steadily increase in size and complexity. A lot of techniques for managing these aspects have been defined. Today software is often a composition of software libraries and newly written code. Newer technologies — component-based programming — compose a final product from a set of components so that little code (except for the creation of new components) needs to be written.

Although these technologies allow rapid software development, there are still barriers (see 1.1) which prevent us to fully benefit from them. A final component-based product must reach particular requirements in terms of functionality and extra-functional properties [2] (abbreviated as EFPs). These include characteristics such as dependability, maintainability, usability, security, portability, re-usability, etc. For that reason, components need to provide extra-functional properties that hold this additional information. A developer uses these properties to check which component best suits the final product.

1.1 Goal of the Paper

Extra-functional properties, as we define them, are a special kind of information expressing provided and required qualities of a piece of software. A lot of approaches to define and check EFPs have been proposed [20], [15] including languages for describing EFPs [6], [1].

One of the less well explored issues is how to target EFPs for a particular domain and context [3] (e.g. mobile phones, desktop computers, servers). Some characteristics are often important in one context but less relevant in another one, and concrete values of similar EFPs are usually different. For that reason, EFPs are nowadays used mainly in one context without any possibility to be re-used in another context. For instance, the context of real-time (R-T) applications usually successfully uses EFPs (e.g. response time) but these properties are not simply portable to, say, desktop applications.

This is evidently a discrepancy with component-based programming goals where a component's purpose is to be independent of a context. In this work, our aim is to introduce a way to define extra-functional properties which can be reliably and consistently interpreted in different contexts relevant to a domain. Additionally, we provide the possibility to express a relation of a component to its environment – called *deployment contract* – in terms of EFPs. The main technical contribution of the presented work is the system of registries that stores a group of EFPs relevant to a context and binds particular values to them.

The definitions of properties, deployment contract and registries introduced in this paper are expressed as a precise mathematical model. Although a practical

[2] Often also called non-functional properties or characteristics.

[3] By 'context' we mean specific computation environment — with particular hardware, software, performance or a user comfort needs.

usage of the proposal requires to also develop adequate technical means, the goal of this paper is to establish their fundamental model rather than suggesting a concrete technology. The definitions to be given in the paper aim at simple-to-use models and abstractions which will consequenly increase a chance of industrial adoption.

This paper is organized as follows. The next section contains a brief overview of technologies and research approaches relevant to extra-functional properties. Section 3 describes the general principles of our work, it shows how we define EFPs and their relation to deployment contracts, including the modifications we introduced to existing approaches. Then follows detailed information about our registry system. The last section of this paper provides a case study in which we show how a current system developed by a Czech company can be enhanced using our proposed method.

2 Related Work

This section contains information about approaches which address component-based programming and extra-functional properties.

2.1 Languages for Definition of Extra-functional Properties

One of the approaches to describe general EFPs is NoFun [6]. It distinguishes simple properties which are easily measurable (time, memory, speed, ...) and derived properties which are computed from simple or derived ones using logical expressions ($\wedge, \vee, <, \leq, >, \geq, =, \neg$, ...). The NoFun approach lacks semantics assigned to properties and values. One can assign values to properties and compose derived properties but once the values are defined, they are fixed. The relevance to a context and a domain is not addressed.

A newer approach proposed by Aagedal is the CQML [1] 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 the following basic types: Number, Enum or Set; no complex type is provided. CQML also provides derived properties, but they are meant only to extend an existing simple property or to compose a derived property from other ones without any further definition how this composition is treated. CQML allows to deal with properties before deployment and after a component is deployed. Although properties are usually treated differently in different phases, CQML does not distinguish it. 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.

The CQML has been extended by other authors. An extending language proposed by Röttger and Zschaler is called CQML+ [14]. They allow an explicit definition of resources needed by components. They consider not only resources between

components but also resources between a component and a system (framework or hardware). This is an important aspect of component-based development where a component has not only relations to other components but also relations to the deployment environment. CQML+ extends syntax of original CQML rather than providing a more generalised way.

There also exists another wide group of languages targeted at specific domains. Among these, TADL [11] is a language specialised for trustworthy systems in which *availability*, *reliability*, *security* and *safety* are selected as the most important EFPs. The language contains explicit constructs describing these four characteristics. Another one is HQML [7], which is an XML based language specialised for web applications with Quality of Service (QoS) capabilities. Another language called SLang [8] is oriented to the field of Service Level Agreement in which EFPs play also an important role. SLang uses XML to describe EFPs. It distinguishes between Vertical and Horizontal SLAs. Vertical SLAs concern different infrastructure and Horizontal SLAs concern different layers of abstraction. Each layer of abstraction covers different group of EFPs, allowing to use different EFPs for different layer of abstraction.The weakness of these approaches is their targeting at particular cases and developing a limited set of properties. We in opposite aim at providing a general solution.

2.2 Extra-Functional Properties on Components

One approach to integrate EFPs in component models using structured attributes is presented in [15] 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). Validity conditions specify in which contexts an attribute value is valid in terms of platform, usage profile or inter-attribute dependencies.

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.

Deployment Contracts [9] presented by V. Ukis are focused on detecting possible conflicts among components or a component and its execution environment. Deployment Contract (DC) defines a comprehensive set of meta-data describing (i) environmental dependencies of components and (ii) components' threading models.

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

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

DC might be considered as EFPs of a certain kind. Compared to our approach, there is no underlying formal definition of DCs and it is not specified how they should be compared, which stems from the overall focus on the conflict prevention rather than selecting the most suitable component candidate as in our case. We aim at using DC attributes in our work, but we create a general formalism for them which is consistent with our EFPs.

PECT [19] focuses on real-time quality attributes and their predictions. ROBOCOP [4] uses a set of various model with particular information about the system including an EFPs model that model i.e. reliability, cpu and memory usage. However, both of these models do not allow for EFP context-independent specification. For instance, it is not possible to parametrise an EFP with a particular platform.

Unlike the previous models, Palladio [3] allows EFPs to be parameterised by context. A component developer can annotate each provided service of a component (a method of one of provided interfaces) with an additional specification called Resource Demanding Service Effect Specification (RDSES). Using UML activity diagram 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). In further phases of system development RDSES is parameterised by a resource model, which binds the abstract resource types to concrete service's resource demands in a target resource container, and a usage model 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.

Compared with our approach, Palladio focuses only on performance-related EFPs for whose specification it provides a rich palette of models. Specifically, EFPs' values defined as random variables and taking usage profiles into account are strong concepts. On the other hand, the necessity to create a number of detailed models imposes a significant burden on system and component developers. Moreover, resource platform specification in the form of a resource model has to be created for each system from scratch since the resource repository contains only resource types, not particular instances with performance characteristics. In our approach, we try to encapsulate the values of EFPs from similar contexts (including hardware

[4] http://www.hitech-projects.com/euprojects/robocop/index.htm

platform) in local registries, where they can be reused by many systems, but EFPs definitions remains valid across systems.

In the field of Service Oriented Architectures where quality of service (QoS) characteristics are an important issue, [20] extend the Web Service Modeling Ontology to better support EFPs and propose a service comparison method using quality characteristics. Our work follows similar goals using more traditional means.

In the area of strong industrial frameworks, support for extra-functional properties is rather rare. The Enterprise JavaBeans [16] (EJB) component model works with several predefined properties that can be classified into this area: Locality (whether the services of a bean [5] can be accessed remotely or only by local clients), State (a bean can be defined as stateless or state-full), Transaction demarcation (defines the level of transaction support expected), Security (involves user roles and their privileges). EJB provides no mechanism for defining new or derived EFPs.

2.3 Quality Modeling

Since modeling techniques are of the same importance as the languages, it is necessary to mention the *UML Profile for Modeling Quality of Service and Fault Tolerance* developed by the OMG group [12]. It provides a UML profile extending "standard" UML diagrams with possibilities to add extra-functional properties. Unluckily this profile is inconsistent with the one proposed by Aagedal in his CQML.

A list of EFPs has been put together in the Component Quality Model [2] which may be used for defining a set of EFPs for a particular system. A different approach, Performance Trees [17], aim at an easy-to-use graphical representation of stochastic systems where states, transitions between states and transition conditions together with probabilities are modelled. It allows to express performance queries as graphical trees which is useful to model EFPs of performance demanding systems.

2.4 Summary

Figure 1 summarises our survey of the state-of-the-art. A desired approach should (i) allow general definition of EFPs, (ii) deal with context and domain dependency of components (iii) be easy to use, and (iv) allow to express a dependency both on other components and on the environment. The table shows how current works fulfil our needs and which requirements are missing.

3 Our Approach

When a system developer has a set of components with the same functionality he or she needs to know "more" to select the most suitable one for a particular purpose. That "more" is an extra-functional property assigned to the component or to its service. The developer may decide which component is better (more suitable) by comparing EFPs.

[5] a component in the EJB model

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

Fig. 1. Important attributes of existing works

As can be seen from the above analysis of related work, there is no consistent understanding how extra-functional properties should look like, where they should be stored and how they should be defined. In our point of view, EFPs are additional information about components and their features. Their role is to enhance component specification and extend the possibilities of component verification.

This section describes a system we propose for defining and using extra-functional properties which supports their reuse and reliable comparison.

3.1 General Principles

To obtain comparable properties, there must be a shared understanding between different component vendors and users about the available properties and their characteristics. Apart from the use of standards such as the CQM [2], such understanding can be helped by a technical infrastructure which comprises a general repository containing all available properties in the domain (the field of usage). It allows to assume that properties are defined before a component is created and vendors can therefore use it to attach properties with the same meaning to components.

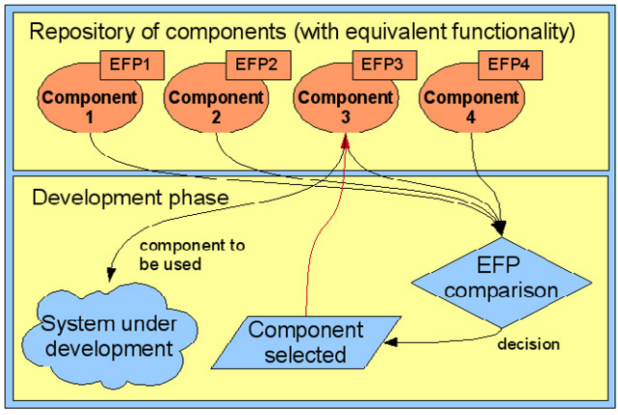

Fig. 2. Component selection

In Figure 2 it is assumed that vendors enrich components by valid EFPs (which

are obtained from the repository and are thus comparable). Then the comparison is made and the best component is selected.

Note that techniques for checking whether the claimed EFP values are valid are needed in this process. They may include any kind of static analysis, simulation, testing, etc., however, the concrete techniques are out of the scope of this paper.

3.2 Registries of Extra-Functional Properties

In our work the EFP repository is implemented by an innovative system of registries. This part of the paper provides their description and explains how they are related to components. The following subsection then formalizes the whole system.

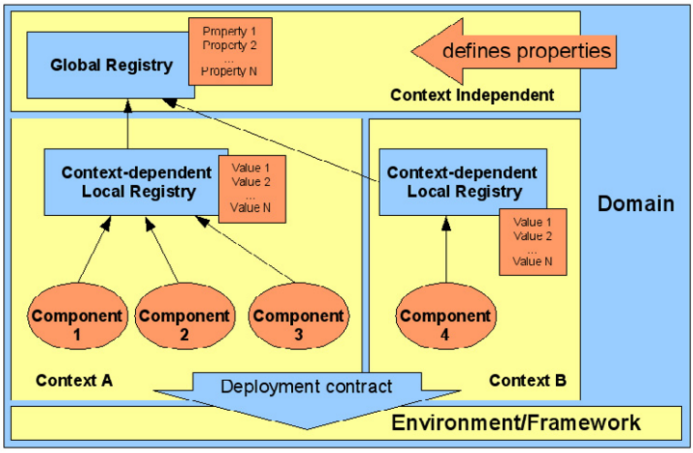

Fig. 3. A relation of registries and components adjusted by contexts and a domain

Figure 3 shows the core idea of our system, demonstrating two dimensions: one concerns **context** dependencies and the other one concerns a **domain** of usage. By a context we mean a different computational environment (e.g. context of mobile phones, desktop machines, servers) and a domain is an area which a final system is developed for (e.g. systems for libraries, hospitals, schools, automotive industry).

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

Global registry (GR) is a store with definitions of EFP types. The GR contains records with the name and the type of each property. It only defines the properties themselves but does not contain their values. Global registry is valid for all contexts specified by the domain of the registry. It contains the definition of all meaningful EFPs defined by a domain expert. The domain expert may be a program working with a domain specification or a human knowing the domain.

Local registry (LR) is concerned with a contextual meaning of EFPs. Each context has one local registry (with a link to the domain GR) which stores values valid for the context. They are linked to the definitions provided by the GR. A component to be deployed in a particular context which links to a LR. It thus binds context-dependent values to its properties (taken from the GR).

This mechanism creates symbolic names for values, prepared by an expert who knows the context. The EFPs themselves are usable in other contexts with different values, because the value names can remain the same while the underlying values are changed.

The rationale behind this design is that one EFP is usually defined for only one purpose and holds a value valid in only one context. For instance, for a "memory consumption" EFP the value 20MBytes would be considered "small" on desktop systems but "high" on mobile devices. Because components can be used in very different environments, this technique is unusable due to its lack of generality.

Note: The deployment contract shows a dependency of components on execution environment or framework (e.g. a resource as a file in operating system, access to hardware, an execution of other processes/binaries). The system of registries does not distinguish between extra-functional and deployment contract properties. They are defined equivalently and they are distinguished when they are used on components.

3.3 Attaching Properties to a Component

Once the properties have been defined both in terms of their type (in global registry) and named values (for the given context), they can be used as a part of the specification of a particular component. We assume that the particular component model using the proposed approach will allow to assign EFPs to each service individually as well as to the whole component. The component's *extra-functional properties descriptor* has a link to a concrete LR. The descriptor is thus able to work with names of EFP values (if names were assigned for a property) or with direct values (if names were not assigned for a property).

The EFP descriptor contains the declarations of two kinds of EFPs — provided and required ones. Provided properties represent the characteristics which the component possesses, as determined by its developers. Required properties declare what characteristics the component's implementation expects on the features to which it will be bound during assembly/deployment stage and at run-time. Services of a component usually contain both provided and required properties. The whole component can only provide some extra-functional properties towards other components. Component's required properties may be explicitly determined as deployment contracts.

Note that for every property it is optional to have a record in local registry. Each property which is not, in essence, context dependent can have values assigned directly in a component descriptor. It is usually every deployment contract or other EFPs (e.g. physical values like $t = 0^oC$, $gravity = 6,67$ or context independent ones like *price*, *marketability*).

4 Formalisation of Extra-Functional Properties and Deployment Contracts

In our approach, we distinguish between two types of extra-functional properties: simple properties and derived properties. A simple property is any measurable property which often has a measuring unit. A derived property is based on a set of simple or derived properties using logical expressions.

We use the CQML model for simple properties with its types *numeric*, *set* and *enum*, but we have added a *string* and a *ratio* type and a *complex* type. The complex type is a composition of simple or complex types.

We have also generalised deployment contract properties [9] to be consistent with our (simple or derived) properties. The provided generalisation defines a deployment contract the same way as extra-functional properties and they differ later when they are used on components. EFPs bound to a component express strictly a relation to other components while DCs bound to a component express strictly a relation to a runtime environment even though the definitions of both do not differ.

4.1 Definition of EFP and Deployment Contracts

Definition 4.1 Definition of simple and derived properties

$$e_{def}^{simple} = (n, \gamma, t, META) \tag{1}$$

$$e_{def}^{derived} = (n, E, \gamma, t, META) \tag{2}$$

$$e^{deployment\_contract} \equiv e^{simple} \vee e^{deployment\_contract} \equiv e^{derived} \tag{3}$$

where the meaning of the formula is:

n is the name of a property

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

$\gamma : x \times y \rightarrow z; z \in \{-1, 0, 1, \text{“}n/d\text{”}\}$ is a function which compares two instances x, y of the property type t, stating which of the two values is better. We work with several predefined *gamma* functions such as Increasing (more is better), Decreasing (less is better), and assume the possibility to define new ones. The meaning of the return values is: -1: x is worse than y, 0: x is equal to y, +1: x is better than y, “n/d”: not-defined.

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

the implicit rule, the value "n/d" is resulted.

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

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

$unit$: String is a measuring unit of the property

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

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

4.2 Registry Formalism

This section provides a formalism for our system of registries which was mentioned in Section 3.2.

Definition 4.2 Global registry is a simple list of definitions of properties.

$$GR = (loc, \{e_i\}) \tag{4}$$

where:

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

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

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

Definition 4.3 Assume there exists a global registry GR. Then a local registry for a context contains records defining values valid in the context. In effect, this assigns a semantics to the properties.

$$LR = (loc, loc_{gr}, S, D) \tag{5}$$

where:

loc is the URI of the registry associating it with the context

loc_{gr} is a link to the global registry

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

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

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

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

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

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

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

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

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

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

4.3 Property Comparison

Two components C_1 and C_2 can be marked as compatible when (i) extra-functional properties on the provided side guarantee at least the same level of quality, (ii) extra-functional or deployment contract properties on the required side declare the need for the same or a lower level of quality, (iii) properties with the same names match [6]. An algorithm for comparing components (usually two versions of the same component or a component in different environments) works in two steps. Firstly, it matches provided and required properties comparing their names and then it checks whether no property is missing on the provided side and no property has been added on the required side.

Secondly, a comparison function is applied on all equivalent properties. Using a sequence $E_C(C_1, C_2) = ((x_i, y_i)_k), x_i \in efp(C_1), y_i \in efp(C_2)$ of equally named properties from the two components, the function $m : C \times C \rightarrow (z_k)$, $z_k \in \{-1, 0, 1, n/d\}$ compares the components by evaluating the function $\gamma(x, y)$ which is defined by formulas 1 and 2 above.

The informal definition of the comparison function is:

$$m(C_1, C_2) : z_k = \gamma_k(E_C(C_1, C_2)_k) \tag{6}$$

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

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

The DCs comparison require the runtime enviroment to be enriched by properties on which the comparision is performed. Althought a detailed model of the environment is not shown here, attaching of properties to environment works essentially the same way as attaching to components. The enviroment provides some properties which are compared on components reguired deployment contracts.

[6] Note that we deal only with an extra-functionality here. There must, of course, be a match with respect to versions, provided and required services, etc.

5 Case Study and Example

In this section we show a re-design of an existing system using EFPs and a deployment contract. This case study explains how our solution can be used in a real application and also shows how concrete values are set in the registry and assigned in a component's manifest file.

5.1 Problem Explanation

Ministry of the Interior of the Czech republic provides information about citizens for each city. These data are delivered as a text file exported from the database of the Ministry which lists address data of persons. The data are imported into databases of cities. The system is shown in Figure 4 and it is a part of a complex software product. There is a component which reads data from the text files and stores the information into the database.

There is another component which is a database of addresses. It stores all addresses in the Czech republic and can provide them through web services. Data about citizens provided by the Ministry may contain mistakes. For that reason, data about addresses are synchronised through the process of the import and mistakes are corrected. When the correctly working component of addresses is present, it must provide valid addresses all the time.

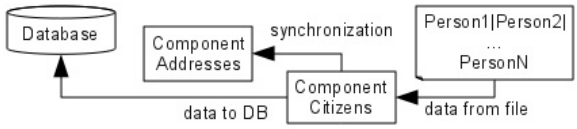

Fig. 4. System for importing information about citizens

We have defined two simple properties *data transferred* and *time to process* concerning the speed and amount of transferred data for the database import. Both properties are referenced by a derived property *performance*. The system depends on an installed database engine, which we define as a deployment contract because it is an external resource. The assumed correctness of addresses is expressed by a *data correct* property.

Example 5.1 The Global Registry for this system contains definition of extra-functional properties. The example shows a part of the GR relevant for the case study application.

```
#Simple properties
data_transferred : increasing integer {unit:‘‘ms’’, names: {low, average, high}}
time_to_process : decreasing integer {unit:‘‘KB’’, names: {low, average, high}}
data_correct : boolean
update_period : decreasing real {unit: "Month" }

#DC property
db_engine : complex { db: enum {Oracle, MSSQL, MySQL}, transactional : boolean}

# derived properties
performance : derived (data_transferred, time_to_process) enum {sufficient, insufficient}
```

Example 5.2 The definition of specific values with assigned values in local registry. The registry assigns specific values to the names. The following values are prepared

for a "smaller cities" context (they have smaller databases and the performance may be lower).

```
URI: http://services.kiv.zcu.cz/citizens/extrafunc/smaller/v1/

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

# 'data_transferred' would be defined the same way
# 'update_period' has only a global meaning and thus is not defined in LR

# the derived property
performance : sufficient = data_transferred >= high AND time_to_process <= low,
      insufficient = data_transferred <= low OR time_to_process > high
```

The LR for a "big cities" context (a big database with a lot of citizens with higher performance demands) may look like:

```
URI: http://services.kiv.zcu.cz/citizens/extrafunc/big/v1/

time_to_process : high = (100; +INFINITY)
time_to_process : average = (50; 100]
time_to_process : low = (0; 50]
# ...
```

Example 5.3 These EFPs are then assigned to the Citizens component in the component's descriptor as follows:

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

# EFP valid for the whole component
Bundle-ExtraFunc:
   performance = sufficient,
   update_period = 3.0
Bundle-DeplContr: db_engine = {db = Oracle, transactional = true }

Provided-Services : cz.zcu.kiv.services.DataReader;
  extrafunc=(data_transferred = average, time_to_process = low)

Required-Services : cz.zcu.kiv.services.DataWriter; extrafunc=(time_to_process = high)
Required-Services : cz.zcu.kiv.services.AddressSynch; extrafunc=(data_correct = true)
```

This last example shows how registry allows to deploy components to a different context without the danger of misinterpreting their extra-functional properties. The interpretation of the symbolic name "*low*" for the time-to-process EFP would vary in different contexts, let's say it might mean "100ms" for server applications, "1 second" on a desktop computer and "6 seconds" for a PDA. However, as long as the component references the same EFP registry (which holds unless its manifest file is tampered with) its *time_to_process* = *low* EFP declaration will evaluate to the same value regardless of the environment where it is deployed.

Another advantage of this approach is that the application designer can reason about the EFPs of services and component in abstract, domain-specific and thus easier to handle terms – for example, both a real-time system's designer and a desktop application programmer can state their system requires a connection to a service of "average or high" processing speed. The links between registries ensure that the values behind these symbolic names are unambiguous in each context, and the designer is always able to obtain the precise value hidden behind the symbolic name.

5.2 Implementation

For the purposes of testing and research we have extended the CoSi component framework [5]. This framework is inspired by OSGi [13], but CoSi is more strict in component's interchangeability checks. Every component in CoSi is treated strictly as a black-box (along the rules described in Szyperski's book [18]) and can be fully versioned. We are able to compare two components by their versions as well as check whether provided and required services match and decide whether they are compatible (for further description of the framework abilities please see [4]). This work extends our framework to allow attaching EFPs to components and later to compare them.

In our particular case, the CoSi framework treats components as Java JAR files with extended manifest files. The manifest file contains information about the public API of the component. It includes a version of components, provided and required services with their versions, etc. Since this constitutes the component's specification, it is suited for enhancement by required and provided EFPs as well as deployment contracts. A property can be in CoSi assigned to the whole component or to service only. A deployment contract is always attached to the whole component. A link to the registry is included as well.

6 Evaluation and Further Research

Since this is still a work in progress, there are several issues that need further research. We need to find a good solution concerning the actual location of context registries. One option is to store this information as a part of the global registry file, another one is to provide other (physically separated) registries per individual contexts and link them to the global file.

The system of registries, however, raises a new group of problems. Once the names for values are defined, their perception will change in time — what is defined as quick may be slow in a few years. A versioning system for registries would help to invalidate old values and keep backward compatibility with legacy components.

The presented model works only with statically defined values of properties. A user compares only properties with values defined in registry. In the real world, the components are connected in a chain of components where a property on provided side may be impacted by properties connected on a required side. The same idea holds also between a component and a runtime environment. Our model is simplified in this respect because it does not reflect this impact. In our future work we would like to introduce functions which describe how a property is influenced by the properties of connected components (or a runtime environment). The aim is to obtain a specification of a function: $e^{prov} = f(\{e^{req}\})$ expressing how a provided property is impacted by a set of required properties.

Local registries define values for the context, but do not address the problem of rescaling values for other or new contexts. Currently, each local registry must be created manually. We would like to provide a model for a context which would, together with functions defining properties, allow to automaticly define values for

other contexts. Additionally, if component's behaviour is different than the descriptor claims, the descriptor must be manually changed which would not be needed when provided properties would be expressed as functions of reguired properties.

We have formalised the deployment contract attributes with the original algorithm presented in [10], but we would like to simplify the algorithm. We allow to put properties only on components and services though the original work allows to do so even for methods, parameters and return types of methods. This will not pose problems, because a resource demand is moved to a component or a service but it is still present. We also assume not to use each attribute of the original work because some are weakly related to deployment (e.g. reentrance and state of methods) and some are difficult to manage (e.g. an ability to store a browser cookie).

Whereas this paper has developed a fundamental mathematical model for a static definitions of EFPs and DCs, the technical means allowing the practical usage of the model have yet to be created. We are currently working on a tool which uses XML to form registry and a Java GUI which serves as an editor of registry. Our next goal is to finish the tool which will verify the model.

7 Conclusion

This paper has presented an approach to the definition of extra-functional properties and deployment contracts targeted to the field of reusable software components. The study of existing approaches has shown that some of the languages describing extra-functional properties developed earlier are a useful base for further developments. Our proposed system builds on CQML and NoFun but simplifies their syntax and extends them in a few directions to better fit our needs. We have further used the approach to introduce a formalisation of the existing, informally defined deployment contracts.

The main focus of our research is to allow interpreting extra-functional properties bound to a component consistently when the component is deployed in various environments or usage contexts. The presented approach preserves the measurable scale of values i.e. their semantics.

We have proposed a system of registries as a repository of existing extra-functional properties stored in registry records which define the property values for each computational environment and assign them names expressing their meaning. The core idea has been implemented in our experimental framework CoSi.

References

[1] Aagedal, J. Ø., "Quality of Service Support in Development of Distributed Systems," Ph.D. thesis, University of Oslo (2001).

[2] Alvaro, A., 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* (2006), pp. 28–37.

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

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

[5] Brada, P., *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 (2008).

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

[7] Gu, X., 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** (2001), pp. 61–95.

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

[9] Lau, K.-K. and V. Ukis, *Defining and checking deployment contracts for software components*, Proc. 9th Int. Symp. on Component-based Software Engineering, LNCS 4063 (2006), pp. 1–16.

[10] Lau, K.-K. and V. Ukis, *Preprint series cspp-37: A reasoning framework for deployment contracts analysis*, University of Manchester (2006).

[11] Mohammad, M. 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* (2008), pp. 290–297.

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

[13] The OSGi Alliance, "OSGi Service Platform Core Specification, Release 4.1," (2007), available at http://www.osgi.org/.

[14] Röttger, S. 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* (2003), pp. 43–56.

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

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

[17] Suto, T., *Performance trees: A new approach to quantitative performance specification*, in: *in Proc. 14th IEEE/ACM Intl. Symposium on Modeling, Analysis and Simulation of Computer and Telecommunications Systems (MASCOTS 2006* (2006), pp. 303–313.

[18] Szyperski, C., (with Dominik Gruntz and S. Murer), "Component Software - Beyond Object-Oriented Programming: Second Edition," Addison-Wesley / ACM Press, 2002, 608 pp.

[19] Wallnau, K. C., *Volume III: A technology for predictable assembly from certifiable components*, TECHNICAL REPORT, CMU/SEI-2003-TR-009 (2003).

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