CORRECT MATCHING OF COMPONENTS WITH EXTRA-FUNCTIONAL PROPERTIES
A Framework Applicable to a Variety of Component Models

Kamil Ježek and Přemek Brada
Department of Computer Science and Engineering, University of West Bohemia, Univerzitni 8, 30614 Pilsen, Czech Republic

Keywords: Software components, Extra-functional properties, Compatibility, Inter-component binding, Framework.

Abstract: A lot of current approaches attempt to enrich software systems with extra-functional properties. These attempts become remarkably important with the gradual adoption of component-based programming. Typically, extra-functional properties need to be taken into account in the phase of component binding and therefore included in the process of verifying component compatibility. Although a lot of research has been done, practical usage of extra-functional properties is still rather scarce. The main problem could be in a slow adaptability of specialized research models to rapidly changing industrial needs. We have designed a solution to this issue in the form of a modular framework which provides a formally sound yet practical means to declare, assign and evaluate extra-functional properties in the context of component-based applications. One of its strengths is applicability to a variety of industrial as well as research component models. This paper describes the models and algorithms of the framework and introduces a prototype implementation proving the concept.

1 INTRODUCTION

With today's need for software systems, techniques which help improve their production are becoming more important. Both industry and the research community therefore invest considerable effort into improving such approaches as component programming or service oriented architecture. Despite noticeable benefits which they bring, new issues keep to arise. One of the important ones is enriching components or services with extra-functional properties (EFPs).

For instance, (1) qualitative properties such as *speed*, *response time*, *memory consumption* or (2) user requirements such as *marketability*, *price*, *regular updates*, *technical support* or (3) behavior properties such as *synchronisation*, *concurrent access*, *deadlock free computation* should be taken into account once particular components are selected to compose a final software. Hence, EFPs provide developers with strong means to assess the applicability and compatibility of components with the task and architecture at hand.

There have been several attempts at providing EFP support for software systems. They start from describing EFPs (Chung et al., 1999; ISO/IEC, 2001) through they applicability to other systems to the development of complex systems embedding as their part either EFPs (Muskens et al., 2005; Bondarev et al., 2006) or quality of service specifications (Yan and Piao, 2009; García et al., 2007). Although these research frameworks have already shown directions leading to the successful implementation of EFPs, their practical applications are still rare and industrial component frameworks with no EFP support such as Spring or OSGi are widely used.

In this paper we present a different approach to this problem. Rather than creating a complex component model which natively supports EFPs, we propose an independent framework which enables to enrich existing industrial systems with EFPs through a set of extension points. The rationale is that even a basic EFPs support is beneficial for industrial component frameworks. In addition, evolution of these frameworks is then not limited to the adaptation of EFPs.

1.1 Structure of the Paper

We first introduce the proposed extra-functional framework with its modules in Section 2. A brief introduction of the concept is followed by formalisations of each part of the framework in Section 3. It

Ježek K. and Brada P..
CORRECT MATCHING OF COMPONENTS WITH EXTRA-FUNCTIONAL PROPERTIES - A Framework Applicable to a Variety of Component Models.
DOI: 10.5220/0003468401550166
In *Proceedings of the 6th International Conference on Evaluation of Novel Approaches to Software Engineering* (ENASE-2011), pages 155-166
ISBN: 978-989-8425-57-7
Copyright © 2011 SCITEPRESS (Science and Technology Publications, Lda.)

includes an algorithm of EFP evaluation and matching. In Section 4 we provide some details on a prototype implementation validating the introduced concept and Section 5 presents examples of the applicability of the approach to selected industrial frameworks.

2 EXTRA-FUNCTIONAL FRAMEWORK MODULES

The proposed framework aims to cover the typical user roles and activities in component-based development. A domain expert or architect first designs the extra-functional properties to be used across a range of components and applications, then a developer states the concrete properties of his components and their values. Finally, when the component software is being composed, an application assembler needs to verify the compatibility of components which should form the application. The framework proposed here consequently allows to declare EFPs, store their definitions and values in a repository, assign them to particular components and evaluate their compatibility.

The conceptual structure of the framework consists of four modules as depicted in Figure 1. The Repository stores EFP definitions and is accessed by other modules to obtain, create or modify the properties. The EFP Assignment part uses the Repository so it can attach the declared EFPs to each component. Once components are enriched by EFPs the Evaluator takes care of comparing attached EFPs when verifying component compatibility during their binding process. While the Assignment works with separate components, the Evaluator covers a set of components which compose a final component application.

All these modules are tied together by EFP Types which defines the structure (type and values) of individual extra-functional properties (Ježek, 2010a).

What we aim at is a loosely coupled framework which may be easily extended and applied to a wide set of component models. The only assumption is that the targeted component models recognize required and provided counterpart elements used when creating inter-component bindings (Szyperski et al., 2002).

In the following subsections we respectively describe the details of the EFP Types, the Repository, how the Assignment module achieves component framework linking, and EFP evaluation.

2.1 Extra-functional Properties Types

The interchange of extra-functional properties between the modules of the framework requires a shared

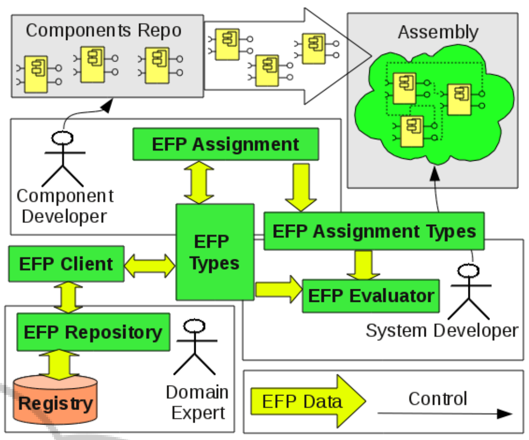


Figure 1: Framework Overview.

understanding of EFP data. This is realized by the module called EFP Types which is the implementation of the model of extra-functional properties presented in (Ježek, 2010a) and formalized in (Jezek et al., 2010). It defines individual EFPs, their structure and relations to a system of registries (Jezek et al., 2010; Ježek, 2010b).

Getting some inspiration in NoFun (Franch, 1998), we distinguish between simple and derived extra-functional properties. The semantics is that a derived property is based on a set of other (simple or derived) properties and its value is assigned according to a logical formula expressing the conditions when the combinations of values of other properties is valid.

The form of the properties allows also to define deloyment contracts (Lau and Ukis, 2006), which express relations between components and a runtime environment.

We formally define a collection of extra-functional properties as a set

$$E = \{e \mid e = (n, E_d, \gamma, T, META)\} \qquad (1)$$

where

n is the name of a property,

$T \in T_{types} = T_c \cup T_s$ is the type of a property,

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

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

$\gamma : T \times T \to Z$; $Z = integer \cup \{\text{“}n/d\text{”}\}$ is a function which compares two instances $x, y \in T$ of the property with type T, stating which of the two values is better. The meaning of the return values is: *negative integer*: x is worse than y, 0: x is equal to y, *positive integer*: 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 comparison can not be determined by the implicit rule, the value "n/d" is returned.

$E_d \subset E, e \notin E_d$ is an optional set containing other properties composing this EFP,

$META$ 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 be used in local registries from Section 2.2.

For instance, two simple properties are defined as follows:

```
(time_to_process, no-gamma, integer,
  META {unit:``ms'',
       names: {low, average, high}}
)
(data_transferred, no-gamma, integer,
  META {unit:``MB'',
        names: {low, average, high}}
)
```

A derived property definition then looks like this:

```
(performance,
  {time_to_process, data_transferred},
  no-gamma,
  enum {sufficient, insufficient}, META {}
)
```

2.2 Universal EFP Repository

A stand-alone repository provides one commonly accessible place to store the EFPs. The reason for this centralization is that in component development, the components are typically developed and used by world-wide organizations. If two vendors attach an EFP with the same name to their components, the same understanding of its structure and semantics is needed (Ježek, 2010b). This is the role of the repository and therefore other modules load data from this common repository.

Since a given component or service can be run in different environments, an EFPs mechanism must take this heterogeneity into account. We therefore use a layered design of the repository (Jezek et al., 2010).

The upper layer called Global Registry (GR) is a storage of EFP definitions. It collects definitions of EFP types from Section 2.1 and thus ensures all its EFPs are valid and meaningful in a particular domain (e.g. education, healthcare, automotive). Let us furthermore highlight that Global Registry does not contain concrete values of EFPs because they may differ widely among runtime and application environments.

Formally, the Global Registry is a triple

$$GR = (id, name, E) \tag{2}$$

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 2.1).

The lower layer of the repository is called Local Registry (LR). It stores EFP values pertinent to a particular computational environment, i.e. each environment has its own Local Registry with concrete values of the properties defined in the Global Registry.

Local Registry is formally defined as

$$LR = (id, GR, name, id_{parent}, S, D) \tag{3}$$

where:

id : Integer is the registry's unique identifier,

GR is the Global Registry this LR is linked to,

name : String is a human readable repository name,

id_{parent} : Integer is the (optional) identifier of a parent LR, which allows to build LRs in tree hierachies. The semantic is that a value from a parent is inherited unless this LR overrides the value.

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

$D = \{(e, value\_name, v, r) \mid e \in E \wedge value\_name \in String \wedge r \in R \wedge v \in V_{LR}\}$ is a set of derived property definitions, where each derived property e is governed by a logical rule r for which

$V_{LR} = \{v_i\}_{i \in I}$ set holds all values for the given EFP assigned in this LR or its parents. I is the set indexing values assigned in LR or its parents,

v is an assigned value,

e is a property from GR,

value_name 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.

$r \in R = E \times LF \times V_{LR}$ is a set of derivation rules. E is a set of EFPs, LF is a set of logical expressions $LF = \{f(x) \mid f : String \rightarrow Boolean\}$. A value v is assigned to the EFP when the evaluation of f is *true*.

One advantage of using a Local Registry is that it holds context-dependent values with assigned names. The names remain the same while concrete values differ, and a developer using the Local Registry mechanism may think of the semantics of the EFP value as denoted by the name rather than about a concrete number.

For example, a Local Registry for smartphones with GPRS-only connection may contain the following value definitions:

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

whereas a LR for wifi-connected tablets could define

```
time_to_process: low = 1, high=1000, ...
data_transferred: low = 10, high=500, ...
```

Another advantage of this solution is that continuous intervals of values can be partitioned into a disjunctive set of named intervals, values or subsets. Hence all values in one partition may be treated as equivalent in the EFP evaluation process. For example, memory consumption in an interval $\{1, +\infty\}GB$ may be considered as *too high* for resource constrained devices. It does not matter whether the real value is $1GB$, $1.2GB$ or $5GB$ – they all belong to one named group.

2.3 Applicability of EFPs to Variety of Component Models

The assignment of EFPs to components actually consists of two phases of EFPs manipulation. In the first phase a developer attaches EFPs to components, loading the properties from the repository. In the second phase the Assignment module provides the previously attached EFPs to other systems (see Figure 2). The form of transfered data are so called EFP Assignment Types.

When designing the module, special care has been taken not to depend on a concrete component model. The EFP Assignment module solves the independence using two separate sub-modules.

The EFP Mirror sub-module shown in Figure 2 represents the independent part of the EFP Assignment module. In the phase of attaching EFPs to a component, a developer loads EFPs from the remote EFP repository and applies them to the component (e.g. attaches a *time_to_process* property to a service

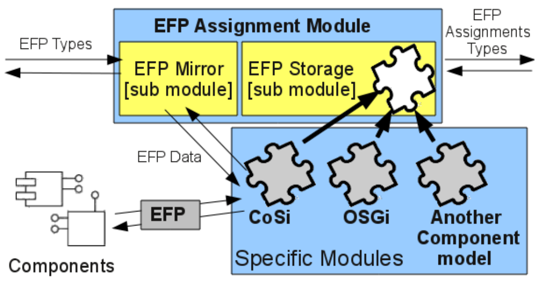


Figure 2: EFP Assignment Module.

interface). Since it would be impractical to call the EFP repository every time the data are needed later on, the EFP Assignment module stores complete information about 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 considerable amount of extra information may potentially need to be stored together with the component in case the EFPs are many or deeply structured.

The Data Storage Sub-module from Figure 2 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 differently in different component models. For that reason each implementation of this sub-module decides (1) where and how to store EFP data, (2) how to link the data with concrete features of the component.

In Section 4 we describe an implementation of this framework for the CoSi component model (Brada, 2008) together with technical details on the modules.

2.4 EFP Evaluation and Binding

The most innovative part of the framework is the Evaluator. Its main purpose is to load a set of components and verify their compatibility in terms of extra-functional properties.

The module first obtains EFPs of components calling the EFP Assignment module for each component. 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 3.1 and 3.2 provide more details.

Unlike other modules the Evaluator is not customizable because 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.

2.5 EFP Assignment Types

Since the evaluator aims at generality, data coming in must cover a wide spectrum of component models. For that reason EFP Assignment Types is a 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 the framework is able to serve this data to its other parts; in particular, the Evaluator receives data from the Assignment module via EFP Assignment Types.

The model used by EFP Assignment Types is formally defined as a set $AT = F \times E \times V$ where F is a set of all generic representations of component features:

$$F = \{f \mid f = (name, type, role, mandatory, \mu)\} \quad (4)$$

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 must be bound to another feature in the matching proccess.

type is the meta-type of the feature including a name (for instance “interface”, “package”, “component”), parameters (for instance inputs and outputs of methods). Very often, the type is extended with a version of the feature where $version = (s_1 ... s_N), N > 1$ and s is a tuple expressing a part of a full feature version information. Typical form of the version is a triplet concerning major, minor and micro change in interfaces together with a release note. E.g. a version may read $1.0.1.RC1$.

μ is a matching function on F.

Implementing a default behavior, the μ function matches 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,
- mandatory required feature must have a provided counterpart,
- features are compatible in terms of their types (e.g. parameters of interfaces are of the same types). A required feature must be a sub-type of the provided feature. If the type compatibility is explicitly expressed as versions, then a version on the provided side is equal or greater than a 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 (Bauml and Brada, 2009) would reach more accurate results. Since instances of the features come from the EFP Assignment module, the extension is straightforward: the reimplementation of a component-specific sub-module in the assignment module provides different μ function while the algorithm of the evaluator remains unchanged.

Continuing with the definition of EFP Assignment, E is a set of extra-functional properties from Section 2.1 and V is a set of values (Ježek, 2010a) assigned to the properties which has three forms:

$$V = V_{direct} \cup V_{LR} \cup V_{formula} \quad (5)$$

$v \in V_{direct}$ is a directly assigned value typically independent on a context of usage. In other words, these values remain constants independently on a runtime environment.

$v \in V_{LR}$ is a value valid for a particular context of usage as defined by a local EFP repository (LR) typically holding values dependent on a context of usage and varying among contexts. A component can thus contain multiple values of a given property for different contexts. Evaluating components, one must select which context a result should be computed for and the evaluator then uses only values valid for the selected context.

$v \in V_{formula}$ is a mathematical 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.

For instance, a component may declare its speed-up by the Amdahl’s law $\frac{1}{(1-P)+\frac{P}{S}}$. P expresses the amount of a code which may be parallelized and for a particular component it is constant (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 from the runtime.

Following example concludes thos section. It shows the EFP from Section 2.1 attached two 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) )
)
```

Let us note that `matched-by-name` denotes a function which matches two features with the same names. `LR.1` and `LR.2` are two local registries identified by their IDs, `direct` is a direct value – 20ms in this example, `math` defines a math formula. There must be also assignment for the EFP `data_transferred` which we omit here for space constraints. Typical cases have only a one type of a value (LR, direct or math-formula one) defined in assignment. It is only a purpose of this example to show all possibilities.

3 EFP EVALUATOR ALGORITHMS AND FORMALIZATIONS

The following sections detail the process of the evaluation using more formal means. Formal definitions of data used by the evaluator and the algorithm verifying compatibility of components are also introduced.

3.1 Structure of EFPs 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 which is created by the EFP Evaluator is an oriented graph $\overrightarrow{G} = (V, E)$ where V is a set of vertexes and E is a set of edges, with specialized types of vertexes and edges:

$$\begin{aligned} V(\overrightarrow{G}) &= V_{component}(\overrightarrow{G}) \cup V_{feature}(\overrightarrow{G}) \cup V_{efp}(\overrightarrow{G}) \\ E(\overrightarrow{G}) &= E_{belong}(\overrightarrow{G}) \cup E_{match}(\overrightarrow{G}) \end{aligned} \tag{6}$$

The following rules hold for each vertex v:

$v \in V_{component}(\overrightarrow{G})$ if v represents a component. It is a root meta-vertex which purpose is to simply aggregate all features of a component,

$v \in V_{feature}(\overrightarrow{G})$ if the vertex represents a feature. It expresses one type of feature depending on concrete implementation for a particular component model. It may express e.g. "interface", "service" or whole "component",

$v \in V_{efp}(\overrightarrow{G})$ if the vertex represents an EFP. These vertexes are connected with $V_{feature}(\overrightarrow{G})$ vertexes to express EFPs on concrete features of a component.

Furthermore, the following rules hold for each edge e:

$e \in E_{belong}(\overrightarrow{G})$:

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

This kind of edge expresses how components, features and EFPs are connected.

$e \in E_{match}(\overrightarrow{G})$:

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

While features are bound by the mentioned function μ, 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.

Using this model, the 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 added for each component and vertexes for features ($V_{feature}(\overrightarrow{G})$) and EFPs ($V_{efp}(\overrightarrow{G})$) are created. Furthermore, the vertexes are connected using the "belong" edges ($E_{belong}(\overrightarrow{G})$) to express which features and EFPs are attached to the components.

Finally, isolated graphs, representing individual components, produced by the previous steps are connected by matching all pairs of corresponding provided-required features as well as EFPs. Hence, "match" edges of the type ($E_{match}(\overrightarrow{G})$) complete the graph. These edges denote the connections of features among components and pairs of EFPs attached to the features.

The final graph completely represents components and their binding together with their EFPs.

3.2 Evaluation of EFPs

Having a 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.

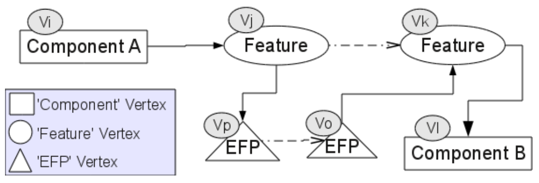


Figure 3: Example Graph.

3.2.1 Construction of EFP Graph

The algorithm which computes the values of attached EFPs as well as checks the connection of components with each other uses a modified depth first search 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):

1. Input sets of vertexes $V_{component}(\overrightarrow{G})$, $V_{feature}(\overrightarrow{G})$ and $V_{efp}(\overrightarrow{G})$ are established and the first vertex $v_i \in V_{component}(\overrightarrow{G})$ is selected, temporary vertexes $v_j, v_k, v_l = null$, previous vertex $v_{i-1} = null$.
2. Find a first feature vertex finding the first edge $e_{ij} \in E(\overrightarrow{G})$ where $v_j \in V_{feature}(\overrightarrow{G})$. The direction of the edge is from a component to the feature which symbols a required feature. If there is no such edge, the component has no required element and a new input set is specified $V_{component}(\overrightarrow{G}) = V_{component}(\overrightarrow{G}) - \{v_i\}$ and the algorithm goes to step 5. Otherwise, proceed.
3. Find the first edge $e_{jk} \in E(\overrightarrow{G})$ where $v_k \in V_{feature}(\overrightarrow{G})$. A direction is from a feature to another feature representing a connection of required feature to a matching provided one. If the edge is not found and the feature is mandatory, it means that a requirement of the component is not fulfilled $\rightarrow$ ERROR. For non-mandatory features, the algorithm goes back to step 2. Otherwise, set $v_{i-1} = v_i$.
4. An edge $e_{kl} \in E(\overrightarrow{G})$ where $v_l \in V_{component}(\overrightarrow{G})$ is selected. The new first vertex $v_i = v_l$ is set and the algorithm goes back to step 2.
5. If $v_j, v_k \neq null$ two sets of vertexes $V_{efp_k} = \{v_o \mid \exists k : e_{ok} \in E(\overrightarrow{G}) \wedge v_k \in V_{feature}(\overrightarrow{G}) \wedge v_o \in V_{efp}(\overrightarrow{G})\}$ and $V_{efp_j} = \{v_p \mid \exists j : e_{jp} \in E(\overrightarrow{G}) \wedge v_j \in V_{feature}(\overrightarrow{G}) \wedge v_p \in V_{efp}(\overrightarrow{G})\}$ are selected. V_{efp_k} is a set of EFPs on the feature v_k and their values are computed first, then EFPs V_{efp_j} on the feature v_j are also computed. The vertexes are removed from the input set $V_{feature} = V_{feature} - \{v_j, v_k\}$.

   The computation of EFP values differs for three types of values (Ježek, 2010a)

   - Direct value: a concrete value assigned to the EFP is directly used;
   - Local value: a value for a context of usage a user aims to compute is used;
   - Mathematical formula: computed using values on connected components which must be certainly known at this time (because the depth first search algorithm must have already visited connected components).

   In addition, the following must hold: $\forall p \exists o : e_{po} \in E(\overrightarrow{G}) \wedge v_o \in V_{efp_k}(\overrightarrow{G}) \wedge v_p \in V_{efp_j}(\overrightarrow{G})$ meaning that all EFPs on the required side must be connected to their provided counterparts, otherwise $\rightarrow$ ERROR.
6. If $v_{i-1} \neq null$ a new initial vertex is set $v_i = v_{i-1}$ else if the set $V_{component} \neq \emptyset$, select another vertex $v_i \in V_{component}(\overrightarrow{G})$. Then go back to step 2. Otherwise, the graph evaluation ends.

The algorithm verifies any inconsistency in the graph in terms of component bindings. The verification finds missing provided component elements connected to the required sides of other components. It furthermore finds missing EFPs on the provided sides matching EFPs on the required sides attached on bound components.

3.2.2 Evaluation of EFP Values in Graph

Once the components are bound and the EFPs are in matching pairs, as a result of the algorithm, it is possible to compare values on the EFPs. This step verifies whether a quality demanded on the required side is guaranteed by the EFPs on the matching provided side.

The verification of values must first select a sequence of required-provided EFP pairs from the graph. The sequence $P(V(\overrightarrow{G}), V(\overrightarrow{G})) = \{(v_x, v_y) \mid \forall x \exists y : e_{xy} \in E(\overrightarrow{G}) \wedge v_x \in V_{efp}(\overrightarrow{G}) \wedge v_y \in V_{efp}(\overrightarrow{G})\}$ contains EFP vertexes to be compared. A sequence of EFP values attached to these vertexes is obtained applying the function:

$$value : V(\overrightarrow{G}) \times V(\overrightarrow{G}) \rightarrow T \times T \quad (7)$$

where T is a set of EFP value instances computed on respective vertexes.

Furthermore a function $\gamma : T \times T \rightarrow Z$ (Section 2.1, equation 1) compares value pairs returning a numeric result. Taking it together, the sequence of vertex pairs is transformed to a set of numbers.

$$\gamma \circ value : V(\overrightarrow{G}) \times V(\overrightarrow{G}) \rightarrow Z \qquad (8)$$

Using the functions, vertexes from the input sequence are finally compared:

$$z_k = \gamma_k(value_k(P(V(\overrightarrow{G}), V(\overrightarrow{G}))_k)),$$
$$k = 1..|P(V(\overrightarrow{G}), V(\overrightarrow{G}))|$$

The resulting sequence of numbers is checked. A non-negative number means that a quality has been satisfied. For that reason the evaluator verifies that $\forall k \exists z_k : z_k \in [0, \infty) \subset Z$ holds. Otherwise the evaluator signals an error for the EFP wrapped in the respective vertex.

For instance, let us assume the property `time_to_process` with numeric values and a γ function $\gamma(x,y) = x - y$ (shorter processing time is better). Following Figure 3 with values assigned to vertexes $v_p := 10$ and $v_o := 30$ the evaluation returns $\gamma(value(v_p, v_o)) = \gamma(value(10, 30)) = 10 - 30 = -20$. The result of the evaluation for these vertexes leads to incompatible EFPs. For different values $v_p := 40$ and $v_o := 30$ the evaluation would succeed.

4 IMPLEMENTATION AND TOOLBOX

As a proof of the concept, the framework has been implemented[1] in the form of a set of modules and tools.

4.1 EFP Repository Server

In our case, the EFP repository has been implemented as a web application storing data in a relational database and producing raw output via web services using SOAP. This approach provides a low-level XML data via standard HTTP protocol. The server, in addition, provides WSDL files describing the web service interfaces with the advantage of generating a client program communicating with the server.

On one hand this technology allows a low-level access by several clients working with SOAP protocol, 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 4, 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.

From a technological point of view, the server is a Java application using the Spring framework[2]. It integrates the Hibernate framework[3] for the database layer and Apache CXF[4] providing web-services. While Hibernate offers an independent access to a variety of database systems, CXF provides a rapid creation of web-services. Currently, Spring is used as an inversion-of-control container, however, its usage allows extensibility in the future. For instance, a web client may be implemented to improve a user comfort to interact with the server. The EFP Client has been created as a web-services client and thus it also uses Apache CXF.

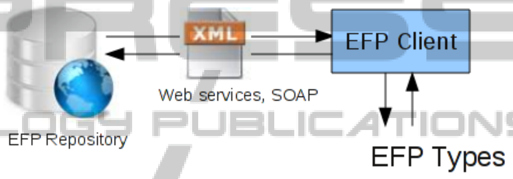
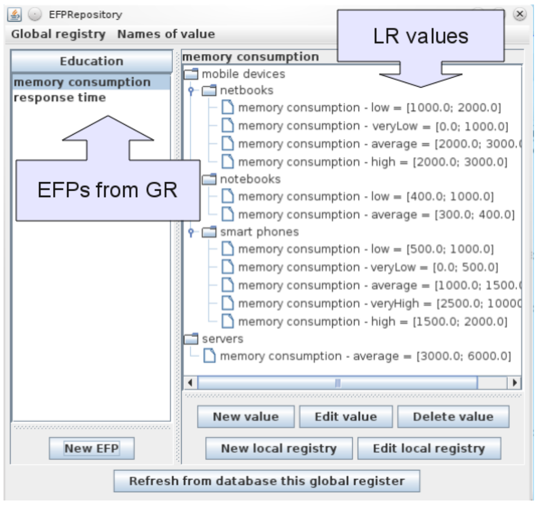


Figure 4: Server Communication.

4.2 EFP Assignment and Evaluator

The two modules for transferring data (EFP Types and EFP Assignment Types) as well as the EFP Assignment module have been developed in a pure Java.



```
memory consumption
mobile devices
  netbooks
    memory consumption - low = [1000.0; 2000.0]
    memory consumption - veryLow = [0.0; 1000.0]
    memory consumption - average = [2000.0; 3000.
    memory consumption - high = [2000.0; 3000.0]
  notebooks
    memory consumption - low = [400.0; 1000.0]
    memory consumption - average = [300.0; 400.0]
  smart phones
    memory consumption - low = [500.0; 1000.0]
    memory consumption - veryLow = [0.0; 500.0]
    memory consumption - average = [1000.0; 1500.
    memory consumption - veryHigh = [2500.0; 1000
    memory consumption - high = [1500.0; 2000.0]
servers
  memory consumption - average = [3000.0; 6000.0]
```

Figure 5: EFP Repository Tool.

[1]https://www.assembla.com/code/cosi/subversion/nodes/efps

[2]www.springsource.org/

[3]www.hibernate.org/

[4]cxf.apache.org/

EFP Types module is implemented as a set of classes implementing generic interfaces that are able to hold a complete EFP representation. Main advantage of this approach is its extensibility (Ježek, 2010a). It allows to modify data processed by the framework in the future by writing different implementation of the generic interfaces.

The sub-module of EFP Asssignment data uses XML files for storing EFP. It creates an off-line mirror of the component related data from the EFP repository. In practice, as the data are loaded and applied on the component, the EFP Assignment module mirrors them to a XML file. Where and how the file is stored is up to the other sub-module.

The other sub-module of EFP Assignment, which adresses differences of component models, is developed as a prototype implementation for the CoSi component framework (Brada, 2008). Components in CoSi have a form of Java JAR files. Hence, the prototype implementation packages the XML file into the JAR archive under the `Meta-Inf` folder. Furthermore, a linkage of EFPs to each service is stored in the `manifest.mf` file. A similar strategy may be used for other component models.

The EFP evaluator has been also written in pure Java, however, it uses the JgraphT[5] library for both creating and discovering vertexes of the graph. It leads to an easier implementation of the evaluating algorithm. The evaluator publishes the results via a set of objects aggregating all incompatibility errors. Client applications may process the results as they desire. For instance, the results may be printed to users or they may be sent to other systems.

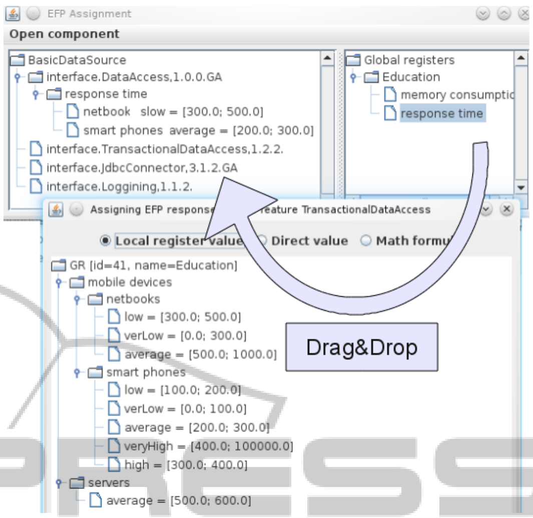
Moreover, all modules are built and distributed as a set of Java JAR files. The distribution form provides other Java-based applications with a possibility to integrate this framework as a third-party library. For instance, the framework may be integrated to a component repository as a tool checking component assemblies.

4.3 Tools

Since modules composing the framework aim at being used in other systems, their direct access by users is impractical. For that reason, we have developed tools offering comfortable graphical interfaces.

The first tool shown in Figure 5 works with the EFP repository to provide control upon the repository data. The implementation is a Java JFC Swing client communicating with the server via web-services. Although the selected technologies would allow to create a web client, the choice taken has been more practical for research purposes, because the process of development simultaneously tests the web-services.

[5]www.jgrapht.org/



```
BasicDataSource
  interface.DataAccess,1.0.0.GA
    response time
      netbook   slow = [300.0; 500.0]
      smart phones  average = [200.0; 300.0]
  interface.TransactionalDataAccess,1.2.2.
  interface.JdbcConnector,3.1.2.GA
  interface.Loggining,1.1.2.
```

```
GR [id=41, name=Education]
  mobile devices
    netbooks
      low = [300.0; 500.0]
      verLow = [0.0; 300.0]
      average = [500.0; 1000.0]
    smart phones
      low = [100.0; 200.0]
      verLow = [0.0; 100.0]
      average = [200.0; 300.0]
      veryHigh = [400.0; 100000.0]
      high = [300.0; 400.0]
  servers
    average = [500.0; 600.0]
```

Figure 6: EFP Assignment Tool.

The other tool shown in Figure 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. In a basic scenario the users drag-and-drop EFPs within one panel to the other panel listing component features. Moreover, each drag-and-drop operation opens a dialog to input a value for the EFP to be attached. Another important aspect is the ability to switch the tool to other component models if the EFP Assignment module contains an appropriate implementation.

5 APPLICATON TO INDUSTRIAL FRAMEWORK

This section briefly shows a possible implementation of the EFP framework for an existing industrial component framework.

5.1 EFPs in Spring Framework

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 (Spring, 2010).

Let us note that despite complexity of Spring, we will for the purposes of this paper use only the

Spring's XML based configuration with setter injection. Details concerning the application of the EFP framework to the Spring's annotation driven configuration are avoided to preserve clarity of the presented approach.

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

Since Spring does not handle extra-functional properties, they must be explicitly added by the EFP framework. The EFP Assignment module must therefore be extended to attach EFPs to Spring Beans.

This may for instance be achieved by extending Spring's XML configuration files using XML namespaces to include also the EFP declarations. We suggest a solution in which the EFP data mirror is a standalone XML file – as has been mentioned in Section 4 – 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.

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>
```

In order to evaluate EFPs attached on Spring Beans, the EFP evaluator must be aware of Beans 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 modify the container life-cycle.

For the purposes of components matching, the `InstantiationAwareBeanPostProcessorAdapter` is useful to monitor bindings of Bundles with one another.

An implementation preparing data for the μ matching function may look like this (in outline):

```
public class EfpAwareBeanPostProcessor
extends
 InstantiationAwareBeanPostProcessorAdapter
{

List<Pair> matchingPairs = ...

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

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

     // "bean" and "anotherBean"
     //are in a matching pair.
     matchingPairs.add(
      new Pair(bean, anotherBean));
   }
   return pvs;
}
```

The presented code contains the method which is invoked by the Spring container each time a new Bean is created. The input parameters hold information about the Bean and its properties which have already been set. Hence, the EFP framework uses it to determine that other Beans have been injected into this one. This information is used for producing the component graph (Section 3.1).

An application of the EFP Evaluator for Spring is straightforward. Using the strategy with Bean Post Processors, the evaluator is invoked as a new Bean is instantiated first, then the attached EFPs are evaluated (Section 3.2). Depending on particular needs, the evaluator can be invoked for each change in the Application Context or only once when the system starts. Any errors found in the evaluating process may cause the Application Context to stop as well as the errors to be logged.

6 RELATED WORK

This work has been partly based on our previous research. Namely, the structure of data stored in the EFP repository is an implementation of our formal definitions published in (Jezek et al., 2010). EFP Types has been implemented using meta-models detailed in (Ježek, 2010a). Hence, the part of the framework concerning the repository is mostly a complement of our previous work while the other part is a new contribution.

There are a lot of other approaches targeting extra-functional properties. They usually cover a rich spectrum of issues, from formal definitions to practical implementations.

An often addressed issue is the description of extra-functional properties. One of the expressing means are specialized languages, for example CQML (Aagedal, 2001) that serves as a complete extra-functional language, CQML+ (Röttger and Zschaler, 2003) that explicitly takes a runtime environment dependency into account, or NoFun (Franch, 1998) that distinguishes between simple and derived extra-functional properties. Furthermore there exist rather specialized languages such as TADL (Mohammad and Alagar, 2008) which is a language describing architectures of systems with a concern of EFPs, HQML (Gu et al., 2001) as a language targeted to web-development, or the SLAng language suited especially for service-level agreement specifications (Lamanna et al., 2003). A general advantage of such approaches is that they provide an answer of what an extra-functional property should stand for. On the other hand they do not address the question of how the properties should be evaluated. Developing our approach, we use these languages to consolidate typical features of extra-functional properties into our model.

Other works propose component frameworks taking extra-functional properties into account as a part of their component models. Let us name at least Palladio (Becker et al., 2009) that targets mainly performance characteristics, Robocop (Muskens et al., 2005; Bondarev et al., 2006) for real-time characteristics, or ProCom (Sentilles et al., 2009). These approaches typically lack modularity in terms of the peculiar ways of using extra-functional properties. It prevents their extra-functional properties to be used in other component frameworks.

Very often, an issue being solved is modeling of extra-functional characteristics. For instance, the OMG group standardized a UML profile (OMG, 2008) covering the quality of services.

Comparing these approaches to our contribution, we aim at a system which is not tied with a concrete component framework, is not intrusive and provides easy integration with other frameworks.

7 CONCLUSIONS

This paper has pointed out a need for extra-functional properties to improve current component based development. Moreover a problem of applicability of extra-functional properties to practice has been mentioned. The most important issue targeted in this paper is a discrepancy of industrial and research component frameworks together with slow application of extra-functional properties to practice.

The contribution of this paper lies in the implementation of an independent framework for working with EFPs in a comprehensive manner. The framework includes a repository of extra-functional properties, a module assigning the properties to each component and an evaluator of the properties. This approach enriches, but does not limits, current industrial component frameworks and aims at filling the gap between application of extra-functional properties and the practically used component frameworks.

The approach has been presented in terms of models and algorithms and the prototype implementation has been introduced.

Since the work is still in the progress, we have some issues to be solved in the future. First of all, our preliminary implementation works with the CoSi component framework. Since CoSi is highly inspired by OSGi, we desire to implement our framework also for OSGi. Furthermore, the implementation for Spring-DM or Spring seems to be also worth considering.

Secondly, the prototype implementation of the repository does not take questions such as security, user access rights, or publishing of new versions into account. We aim at answering these questions as soon as the prototype is fully tested and working.

ACKNOWLEDGEMENTS

The work was partially supported by the UWB grant SGS-2010-028 Advanced Computer and Information Systems and by the Czech Science Foundation project 103/11/1489 Methods of development and verification of component-based applications using natural language specifications.

We would like to thank students Martin Štulc, Lukáš Vlček and Jan Šváb for their considerable help in programming the presented framework.

REFERENCES

Aagedal, J. Ø. (2001). *Quality of Service Support in Development of Distributed Systems*. PhD thesis, University of Oslo.

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

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

Bondarev, E., Chaudron, M. R., and de With, P. H. (2006). 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.

Brada, P. (2008). 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. Springer Verlag.

Chung, L., Nixon, B. A., Yu, E., and Mylopoulos, J. (1999). *Non-Functional Requirements in Software Engineering*. Series: International Series in Software Engineering, Vol. 5, Springer, 476 p, ISBN: 978-0-7923-8666-7.

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

García, J. M., Ruiz, D., Ruiz-Cortés, A., Martín-Díaz, O., and Resinas, M. (2007). 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, Springer-Verlag Berlin, Heidelberg 2007, ISBN: 978-3-540-74973-8*, pages 69–80, Berlin, Heidelberg. Springer-Verlag.

Gu, X., Nahrstedt, K., Yuan, W., Wichadakul, D., and Xu, D. (2001). An xml-based quality of service enabling language for the web. Technical report, University of Illinois at Urbana-Champaign, Champaign, IL, USA.

ISO/IEC (2001). ISO/IEC 9126: Informational technology - product quality - part1: Quality model, international standard iso/iec 9126, international standard organization.

Ježek, K. (2010a). 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*.

Ježek, K. (2010b). 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*.

Jezek, K., Brada, P., and Stepan, P. (2010). 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.

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

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

Mohammad, M. and Alagar, V. S. (2008). 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.

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

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

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

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

Spring (2010). *Spring Framework, ver.3, Reference Documentation*. Spring Comunity, ver. 3 edition. Available at http://static.springsource.org/spring/docs/3.0.x/spring-framework-reference/html/.

Szyperski, C., (with Dominik Gruntz, and Murer), S. (2002). *Component Software - Beyond Object-Oriented Programming: Second Edition*. Addison-Wesley / ACM Press, 624 pages, ISBN-13: 978-0201745726.

Yan, J. and Piao, J. (2009). Towards qos-based web services discovery. In *Service-Oriented Computing ICSOC 2008 Workshops, Lecture Notes in Computer Science, 2009, Volume 5472/2009, 200-210, ISBN: 978-3-642-01246-4*.