Enhancing OSGi with Explicit, Vendor Independent Extra-Functional Properties*

Kamil Ježek, Premek Brada, and Lukáš Holý

Department of Computer Science and Engineering
University of West Bohemia
Univerzitni 8, 30614 Pilsen, Czech Republic
{kjezek,brada,lholy}@kiv.zcu.cz

Abstract. Current industry and research organisations invest considerable effort to adopt component based programming which is promising rapid development process. Several issues, however, hinder its wider adoption. One of them is the practical use of extra-functional properties (EFPs) that research community aims at integrating to component composition but for which industrial applications are still rare. When extra-functional properties are not considered or mis-interpreted, inconsistencies in application performance, security, reliability, etc. can result at run-time. As a possible solution we have proposed a general extra-functional properties system called EFFCC. In this paper we show how it can be applied to an industrial component model, namely the OSGi framework. This work analyses OSGi from the extra-functional properties viewpoint and shows how it can be enhanced by EFPs, expressed as OSGi capabilities. The proposed benefits of the presented approach are seamless integration of such properties into an existing framework and consistency of their interpretation among different vendors. This should support easier adoption of extra-functional properties in practice.

Keywords: Component, Extra-functional, Compatibility, Binding, OSGi.

1 Introduction

With today's need for large and complex software, industry and the research community invest considerable effort to component based programming. Despite partial success, several issues remain unsolved. One of the important ones concerns the usage of extra-functional properties (EFPs) that should improve compatibility verifications of the components.

On the one hand, EFPs and their use in component models are an area of active research. On the other hand, practically used industrial models such as

* 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.

C.A. Furia and S. Nanz (Eds.): TOOLS Europe 2012, LNCS 7304, pp. 108–123, 2012.

OSGi or Spring only slowly adopt systematic EFP support. One of the reasons may be wide misunderstanding of what EFPs are [9] that eventually leads to only a partial and non-systematic EFPs adoption in practice. As a suggested solution, a general mechanism consolidating EFPs understanding among different vendors as well as different applications, called EFFCC[1], has been proposed in our previous work[2] [13]. In this paper, the application of the mechanism to OSGi as a demonstration of its abilities is presented.

Dealing with the discrepancies in EFP terminology, the following generic definition will be used within this paper while other possible options of EFP understanding will be omitted:

> An extra-functional property holds any information, explicitly provided by a software system, to specify a characteristic of the system apart from its genuine functionality to enrich client's understanding of usage of the system supported by technical [computational] means.

This paper first overviews other related approaches in Section 2. Then, a summary of the current OSGi specification and its abilities to express EFPs (according to the given definition) is provided in Section 3. Finally, Section 4 introduces the application of EFPs in OSGi using the EFFCC mechanism [13] and discusses pros and cons of the approach.

2 Related Work

EFP systems have been addressed from several directions and a lot of approaches have been proposed. For instance, one group of approaches proposes independent descriptions of EFPs [7,1,17,11,18]. While this group splits the EFP description from their application, a different group concerns modelling of the EFPs as a part of a software design [16]. These groups treat EFPs rather independently. On the other hand, comprehensive component models exists which take EFPs natively into account [2,27].

Concerning EFP specification, one of the approaches is to employ an ordinary programming code. Examples are: NoFun [7] describes EFPs in a general manner first, then the definitions are applicable to software systems; CQML [1] uses named *Quality Characteristic* with their constraints assigned in *Quality Statement* and put into *Quality Profile* on a component; CQML+ [26] is an extension of CQML allowing to define dependencies of components on runtime environment. These dependencies also allow to express Deployment Contracts [17]. There also exist a lot of specialized languages: TADL [18] for security and safety, HQML [11] for web-development, or SLAng for service-level agreement [15].

QML/CS deals with EFPs in terms of a language as well as formal approach. A system is modelled using Temporal Logic of Actions (TLA+) [16] that specifies

[1] Extra-functional properties Featured Compatibility Checks.
[2] Project hosted at: `http://assembla.com/spaces/efps`

a system as states with traces among the states. The system is called feasible (fulfilling requirements) if the combination of intrinsic, resource and container models implies the extrinsic ones. Other approaches cover component models and frameworks with native EFP support.

Palladio [2] comprises a software architecture simulator, which focuses on performance, reliability, maintainability, and cost for the architecture realization prediction. All of these can be analysed in the Palladio Component Model allowing to make the balance among EFP to achieve desired trade-offs. It is able to accomplish a model-to-model transformation of an architectural model to a stochastic process algebra automatically and then evaluate the algebraical expressions analytically. It uses extended UML State Diagrams to model states of a system with performance characteristics on the diagram edges. Palladio allows deriving the right software architecture design, on the other hand stochastic probabilities and models settings rely on correct human estimation.

Procom [27] defines Attributable elements that consist of a type and one or more values. The ability to store multiple values for one attribute comes from the Procom's concept of refining attributes during the development process. The values consist of data, metadata and conditions under which each attribute is valid. Data is a generic structure which can be specialized into simple data types or a reference to an object. It also enables attribute compositions to get an attribute for the composite component as well as defining the attribute value for composite component explicitly. User is able to select the attributes by versions or configuration filters. The selection may comprise a number of conditions combining AND/OR operators, attribute metadata and Latest, Timestamp, and Version-name keywords. As a result, Procom can define EFPs for different deployment runtime and compose complex properties.

SOFA 2.0 [5] has components and connectors and the key abstraction is metamodelling using MOF [21]. SOFA's concepts such as automatic reconfiguration, behaviour specifications or dynamic evaluation of an architecture at runtime extends a simple functionality and it may be understood as extra-functionality.

Further, specialised component models are: Robocop [19,3] dealing with real-time characteristics for portable or embedded devices; PECOS [8,20] using Petri Nets to model timing and synchronising behaviour of a system. Developers separately model behaviour of each component and the components are composed using a scheduler to coordinate the composition of Petri Nets.

An issue also being solved is modelling of EFPs. For instance, the OMG group standardized a UML profile [22] covering the quality of services, or MARTE [23] for real-time and embedded systems. Furthermore, UML Profiles for NoFun [4] or CQML [1] have been developed.

On the other side, widely used industrial frameworks OSGi (detailed in Section 3), Spring [28], EJB [6] use advanced features such as localisation, transaction, user privileges, synchronisation, composition specification (e.g. `@Qualifier` annotation in Spring allows to specialise candidates to the Dependency Injection) that may be understand as EFPs. However, they still do not provide a systematic EFP approach.

3 Overview of Extra-Functionality in OSGi

The OSGi specification [25] is an approach to modularised Java-based applications adopting several aspects of component based programming as it has been stated by Szyperski [29]. In a nutshell, a component is a Java .jar file (possibly) developed by third parties and independently deployable to its runtime.

OSGi introduces the concept of Bundles as the components. The bundles encapsulate main communication mean, services, in a form of Java classes implementing Java interfaces. Each bundle explicitly exports and imports its packages and explicitly registers and obtains services. The package and service based communication is the main concept and other extensive means such as remote service calls will be omitted here. OSGi runtime then accesses only these packages and services while other ones are hidden. The hiding is technically realised by using separate classloaders for each bundle. This considerably distinguishes OSGi bundles from the standard Java .jar files where only coarse-grained accessibility modifiers (`public`, `private` or `protected`) are provided.

The OSGi specification allows to express restrictions that influences binding of the packages as well as the service lookup. Although they are not called EFPs in the OSGi specification, they may be understood as EFPs.

It is worth mentioning OSGi supports e.g. security concerning signed .jar files, certificates or Remote services calls. Although it goes behind genuine services' function, this work concerns only the properties extending the binding process.

3.1 Motivation: Components Enriched with EFPs

This section shows an example component application enriched with EFPs. The example has been inspired by the Nokia adaptation of the web browser product line [12]. In Nokia's project, components to develop a variety of browsers have been designed. The components follow a structure of web browsers presented in [10] a subset of which is shown in Fig. 1. For simplicity, only the sub-part rendering web pages loaded via the Internet is shown. Following Fig. 1, `RenderingEngine` loads raw HTTP data via the `Networking` interface and renders HTML pages using `CSS`, `HTML` and `JS` components. The `UI` interface accesses platform dependent libraries to draw User Interface primitives. Finally, the pages are provided by the `RenderingQuery` interface. Their consequent processing is omitted here.

Fig. 1 furthermore shows some EFPs (shown as UML notes) expressing selected web browser characteristics. The EFPs concern several disjunctive domain dependent areas. A first domain covers performance (a network speed and a speed to process HTML, CSS and JavaScript), a second domain covers different operating systems (a user interface library – `ui_library` and dynamic HTML support – `dhtml` – depending on graphical libraries provided by the operating system) and the last one covers domain constant characteristics (JavaScript, CSS and HTML versions).

Section 3.2 will show ability and limitation of expressing such properties in current OSGi while Section 4 will show our approach aiming at avoided the detected limitations.

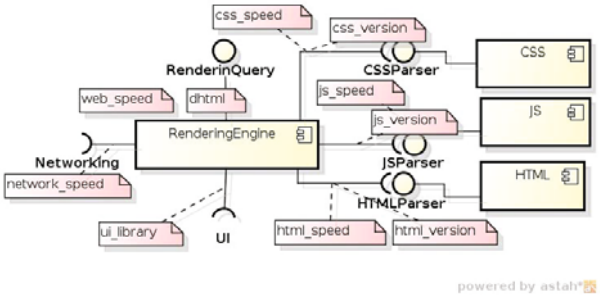

Fig. 1. Example Web Browser Components

3.2 OSGi Parameters, Attributes and Filters

The following parts of this section show four OSGi aspects that have been localised as allowing to express certain EFPs.

Parametrised Exports and Imports. OSGi binding process matches exported and imported packages that consist of a name and a set of parameters. For instance, a bundle implementing operations on HTML pages for a web browser may declare the following package export:

```
Bundle-Name: HTML
Export-Package: cz.zcu.kiv.web.html;version=1.3,html_version=5.0
```

A bundle rendering the web pages requires the `html` package to parse the web pages:

```
Bundle-Name: RenderingEngine
Import-Package: cz.zcu.kiv.web.html;version=1.3;
  resolution:=optional,html_version=5.0
```

A variety of built-in as well as user defined parameters may be used. In this example, `version` and `resolution` are built-in parameters while `html_version` is a user-defined one. Specialising fine-grained binding constraints, `html_version` has been used to define a HTML version these bundles provide/require.

Although these parameters may express EFPs, OSGi lacks their common understanding. Some parameters have a specialised meaning, however, the meaning is defined only in the OSGi text specification with no technical support. User defined parameters are only ad-hoc definitions. Hence, it is a question of how these parameters may be used across different vendors. For instance, if one vendor provides the `html_version` parameter, a different vendor does not have to even know of the existence of the parameter.

Provided and Required Capabilities. In contrary to the packages' parameters, the capabilities are attributes and filters counter elements. The attributes are typed name-value pairs while filters are LDAP conditions over the attributes.

In addition, the capabilities are bound to name spaces that are supposed to explicitly assign them with semantics.

For instance, a bundle from the web browser domain responsible for drawing user interfaces may require some graphical libraries:

```
Bundle-Name: UIBackand
Require-Capability: osgi.ee;
  filter:="(|(ui_library=GTK)(ui_library=Qt))"
```

The example shows an LDAP filter that constraints `ui_library` to either GTK or Qt libraries. The Execution Environment name space (`osgi.ee`) is an OSGi predefined one, however, users may also use their own ones.

Provided capabilities are set as OSGi runtime parameters or by other bundles. Using the latter case, a bundle may provide the `ui_library` (a different name space signals it is not a Runtime Environment attribute):

```
Bundle-Name: UILibrary
Provide-Capability: cz.zcu.kiv.web.ui;ui_library=Qt
```

A set of predefined name spaces exists[3] as well as any user defined one may be created. Despite the name spaces aim at providing the parameters with semantic, it still lacks of common understanding. There is no technical mean to distribute existing name spaces across vendors.

Parametrised Services. Each bundle may register a service and the registration may contain a set of attributes. An LDAP filter may be then used to obtain the service according to the attributes used in its registration. This principle is equivalent to the capabilities.

OSGi API is called within the bundles' source code to register and get the services. This approach is awkward because a developer of one bundle should have to study the source code of another bundle to find out which parameters are actually used.

A way to register and get a `Networking` service stating its *Network speed* attribute may look as follows:

```
Hashtable ht = new HashTable();
ht.put("network_speed", 10);
bundleContext.registerService(Networking.class, this, ht);
```

It is assumed that `bundleContext` is a reference to the `BundleContext` class specified in OSGi. The different bundle may obtain it according to a filter condition:

```
Collection<ServiceReferences<Networking>> services = bundleContext
  .getServiceReferences(Networking.class,
    "(network_speed >= 10)");
```

[3] Current list is available at `http://www.osgi.org/headers` (2011).

Notice that apart from the capabilities, there is no name space definition. Together with different concepts to express import/export parameters, filters used for capabilities and non-transparent usage of parametrised services, this brings yet another ambiguity to the concept.

Parametrised Declarative Services. Introduction of Declarative Services to OSGi specification aims at avoiding drawbacks of Services mentioned above. The registration concept has been replaced by Dependency Injection defined in XML files distributed with bundles.

Although the explicit publication of Declarative Services outside the source code is considerable improvement, the attribute-filter concept remains unchanged. The XML files contain specific elements to express the attributes and filters described in the OSGi Service Compendium [24].

4 Our Approach: Explicit Extra-Functional Properties in OSGi

Although OSGi contains variety of properties that may be considered as EFPs, it has been already mentioned its approach is weak in terms of semantics, general understanding, exchange and evaluation of EFPs. In this section a novel approach aimed at these deficiencies will be presented. The approach uses a general extra-functional properties framework called EFFCC [13] applied to OSGi.

In nutshell, EFFCC is the implementation of a general extra-functional properties support consisting of remotely accessible EFPs storage, tools to apply the EFPs to components and an embeddable evaluator. The EFPs storage is implemented as a JEE[4] Server and the tools are Java desktop applications described in our previous work [13]. The embeddable evaluator provides a generic evaluation mechanism also implemented in Java that is included into existing Java-based component frameworks via a set of extension points.

Fig. 2 overviews the main usage concept: describing the image from left to right, independent (worldwide) vendors develop their components accessing common EFPs storage first. Another vendor then uses the components to assembly an application verifying the components are compatible for the assembly.

Another EFFCC novelty is the split of component development and deployment phases in which different process of EFPs work-flow is used. First, the EFPs are loaded from the repository (using the tool) via the Internet and applied to components in the development phase. The EFPs are copied from the repository and mirrored on the components which furthermore causes the components may be used in the off-line mode without the Internet connection in the assembly or deployment phase.

Main advantage of this solution lies in the repository that unifies EFPs and thus improves EFPs' understanding among the vendors. Apart from any kind of written documentation, the EFP understanding is here supported by a technical mean instantly available to general usage.

[4] Java Enterprise Edition.

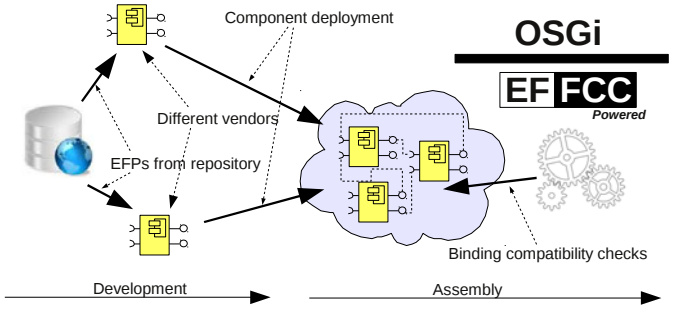

Fig. 2. Development Process with EFFCC on OSGi

As a result, this concept should prevent misunderstanding and improve consolidation of EFPs used among components provided by different vendors and integrated to a final application. As it will be shown later, the repository is capable of holding more detailed information about EFPs than e.g. the OSGi name spaces.

In this work, EFFCC is embedded to OSGi enriching its binding process explicitly considering the binding parameters, attributes and filters as EFPs. The main idea, which will be detailed later in this work, is that the OSGi binding integrates data from the EFP repository unifying their semantics.

The EFFCC mechanism uses approach that splits component model dependent and independent part. The independent part holds the most logic related to EFPs' operations in an abstracted form while the dependent part is a light-weighted layer customising the abstract form to the concrete component model implementation. A benefit is that a common logic may be re-used among multiple component models while a little of additional code must be written to apply the approach to a concrete component model. A detailed description of the independent part, abstract EFPs' definitions and the evaluator algorithm have been already given in our previous work [13] while this work details the application of the component model dependent part, in particular to OSGi.

4.1 Structure of the EFP Data

Although the abstract EFP definitions have been already formalised in [14], a short overview will be provided here to support the fluency of the text. The repository stores EFPs as tuples $e = (n, E_d, \gamma, V, M)$ where tuple elements respectively represent: n the name of the property, $E_d = \{e_i \mid i = 1 \cdots N, N > 0\}$ a set of properties deriving (composing) this property, $\gamma : V \times V \to \mathbb{ZZ}$ a gamma function comparing two property values, V data type and M extensible set of meta informations.

The repository itself is named Global Registry and formalised as: $GR = (id, name, \{e_i \mid i = 1 \cdots N, N > 0\})$. It means the repository has a unique

identifier (id) a human readable name ($name$) and it lists the EFPs. One Global Registry represents one domain of usage and it is furthermore segmented to a set of sub-domains for different area-of-usage. Each such a sub-domain is called Local Registry formalised as: $LR = (id, GR, name, \{LR_i \mid i = 1 \cdots N, N > 0\}, S, D)$ holding a unique identifier (id), a reference to its Global Registry (GR), a name ($name$), a set of other Local Registries (LR_i) that this Local Registry aggregates.

S and D are sets holding values assigned for this particular sub-domain. Shortly, the S set assigns a name to property values giving them semantic and rounding their precision while the D set assigns formulas evaluating the derived properties from the deriving ones. Due to the space constrains, detail formalisation is not provided here and may be found in [14].

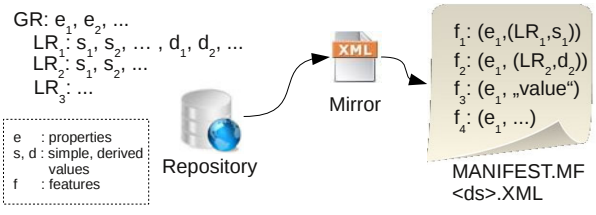

Fig. 3. A Mirrored Repository Assigned to a Bundle

4.2 Specifying EFPs as OSGi Attributes

The application of this EFP mechanism in OSGi is shown in Fig. 3. Its realisation stores the component model independent EFP data as an XML file distributed together with each bundle. In addition, this EFP data are linked to a bundle in the manifest file format or the declarative service XML descriptor via the existing means described in Section 3. It creates a set of all possible assignments $F \times E \times V_A$ where F is a set of all component features (e.g. Bundle packages or services), E is a set of properties from current GR ($e \in E \in GR$) and V_A is a set of assigned values that includes a value from LR, directly assigned value of the V type or a computed value.

The advantage of this solution is that the EFP data are written in a manner similar to the existing OSGi attributes and filters. In addition, a more precise semantic is stored in the unified XML file. While the OSGi resolving process treats these EFPs as standard attributes and filters, the EFP mechanism triggered in the resolving process evaluates also these additional data, resulting in more precise compatibility checks.

OSGi EFP Attributes. Designing a new concept of EFPs, still compatible with current OSGi attributes, lead to the following proposed structure of the EFP attributes:

```
<gr-id>.<efp>=<lr-id>.<value>
```

Here `efp` is a string name of an EFP and `gr-id` is a unique identifier of Global Registry this property is defined for. Furthermore, `lr-id` is a unique identifier of Local Registry and `value` is a value assigned to the EFP. The value form will differ depending on a Registry used. If the `lr-id` is omitted, the EFP has a value assigned that is valid among the domain Global Registry it has been created for. In this case, either a concrete value or a computing formula may be used. It is proposed to use a string representation of concrete values and string representation of formulas evaluating computed values.

If the `lr-id` prefix is used, the value is related to Local Registry (sub-domain of Global Registry) and a named value or a formula deriving a derived property may be used. It is proposed to use a string name for the named values (e.g. "small", "slow", "high", "fast") directly to represent named intervals assigned in respective Local Registry and the literal `"computed"` for the deriving rules. The deriving rules itself will be stored in the XML file.

For instance, two attributes may be defined as: `1.network_speed=1.fast` and `1.html_version=[4.0..5.0]` where *fast* = [10..100] Mb/s is an interval stored in XML and [4.0..5.0] is a directly assigned interval. A small drawback of this solution may be that the named values are not known without looking into related XML files. However, it is assumed users will access the data via the provided tools and thus do not need to access manifest files manually.

EFP Attribute Data Type. A task related to EFP attributes is their evaluation. Original OSGi specification allows to define a data type of the value (e.g. Long, Double, String, Version, using a colon syntax) that in essence denotes the comparing method to be used. User defined types may be also used. Hence, the writing `:EFP` (e.g. `1.html_version:EFP=[4.0..5.0]`) causes the values to be bound to the new EFP data type and OSGi tries to load a respective class named `EFP`. This class is implemented in EFFCC as an adapter which delegates the evaluation to the EFP system, concretely to the respective gamma function defined for each EFP and stored in the XML file. Since not all OSGi implementations currently handle the data type concept[5], a temporal solution is to store all EFP-related attributes in a separate file distributed along the manifest file.

The attributes defined this way may be used in all OSGi parts mentioned in Section 3 (exported packages, provided capabilities and registered services). One exception is that the parameters on exported packages do not allow to define the data type. These parameters use only string comparison. This is a restriction inherited from older OSGi specification which cannot be overcome for compatibility reasons. On the other hand, the parametrised package export/import may be replaced by capability wiring allowing full attributes/filter evaluation. For instance, a provided/required capabilities' pair may look like:

```
Provided-Capability: osgi.wiring.package;
  (osgi.wiring.package=cz.zcu.kiv.web.html,
    1.html_version=[4.0..5.0]
```

[5] Apache Felix for example, which is used in our experiments.

```
Required-Capability: osgi.wiring.package;
  (&(osgi.wiring.package=cz.zcu.kiv.web.html)
    (1.html_version=4.0))
```

This notation expresses package export and import. It is also worth pointing out that the study of the Apache Felix OSGi implementation have also shown a tendency to internally express export/import packages as these wiring capabilities. Therefore, there is a chance this approach may be a recommended practise in the future while export/import package headers may be marked as obsolete.

4.3 EFP Queries as OSGi Filters

Having the attributes defined according to rules mentioned in previous section, the OSGi filters may be used the same way as it is described in the OSGi specification. When OSGi applies the filters, it compares values required by the filter with the values of provided attributes. As long as the provided attributes are defined together with a Bundle, EFFCC loads them and a filter is evaluated delegating each value comparing to the EFFCC implementation (described in Section 4.5).

For instance, a filter may be defined as: `(1.network_speed>=1.slow)` to filter a network service with at least a "slow" connection where the value $slow = [1..10]Mb/s$ will be loaded from the XML file when the filter is being evaluated.

As a result, the filters are used transparently without explicitly considering EFPs while EFFCC running behind treats the values as extra-functional ones. The main advantage is that OSGi implementation is enriched with EFPs providing better semantic than OSGi standard means, however, modification to neither OSGi specification nor implementation is required.

4.4 An Example: EFPs in XML Mirror and OSGi Manifest

Section 3.1 showing a few OSGi bundles will be in this section completed by the example of the EFP data assigned to the bundles.

Fig. 4 depicts a shortened version of the EFP XML repository and attributes plus filters referring to these EFPs. The XML contains definitions of EFPs originating from one Global Registry (the `gr` element) and several disjunctive Local Registries (the `lr` element).

The example shows two Local Registries with IDs 463 and 464 designed for different performance platforms and Local Registry with ID 465 for a concrete operating system. It is assumed these values are understood as the most typical ones for respective platforms. Hence, deployment of such components may use compatibility verifications according to these platforms. This may look as a weakness because this approach requires measurement of all components in all assumed platforms. On the other hand, vendors should test their products for all the platforms to ensure quality and this approach provides a technical mean to express and publish component measurement results together with the component.

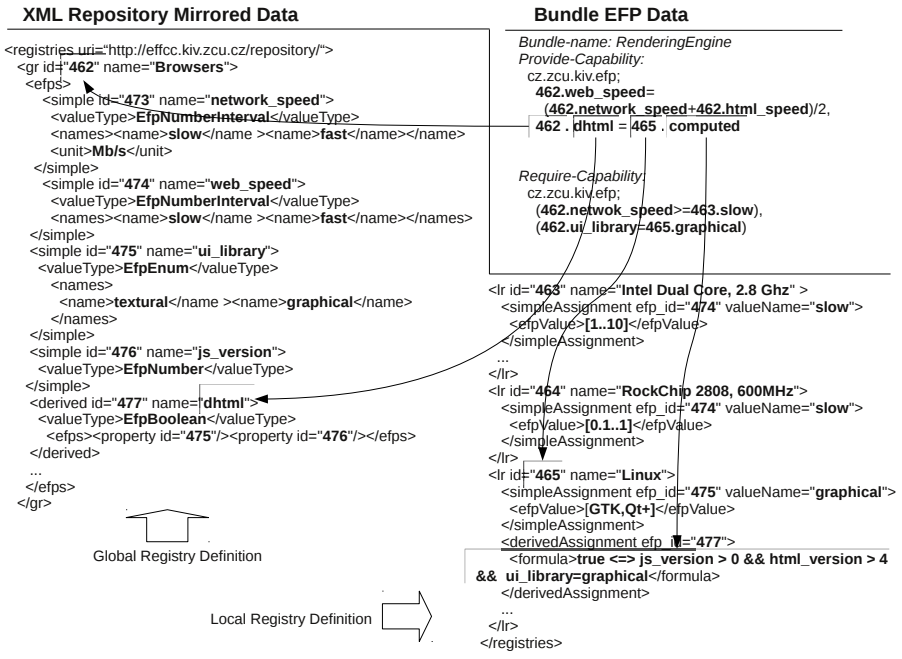

Fig. 4. Example EFPs Applied in OSGi

The case-study furthermore shows several kind of assignable values: (1) `web_speed` is specialised using a mathematical function, (2) `dhtml` has a value assigned for which computation is expressed as a logical formula in Local Registry with ID 465, and (3) `network_speed` is used in a filter referencing the value `slow` from Local Registry ID 463. The last type of value – directly assigned – is not used in the example.

4.5 EFP Evaluation Connected to OSGi Binding

A crucial moment to integrate EFFCC to OSGi is to invoke the evaluation of EFPs at the moment OSGi performs binding of bundle features (e.g. packages or capabilities) or bundles service lookup. OSGi version 4.3 have brought the concept of hooks that may be used to observe and modify the bundles' binding in their life-cycle. A hook is an implementation of a specific interface with a set of call-back methods invoked once a particular operation is performed. According to features described in Section 3 two hooks have been implemented to bridge OSGi binding and EFP evaluation.

The first hook implements the `ResolverHook` interface with a method:

```
void filterMatches(BundleRequirement requirement,
  Collection<BundleCapability> candidates)
```

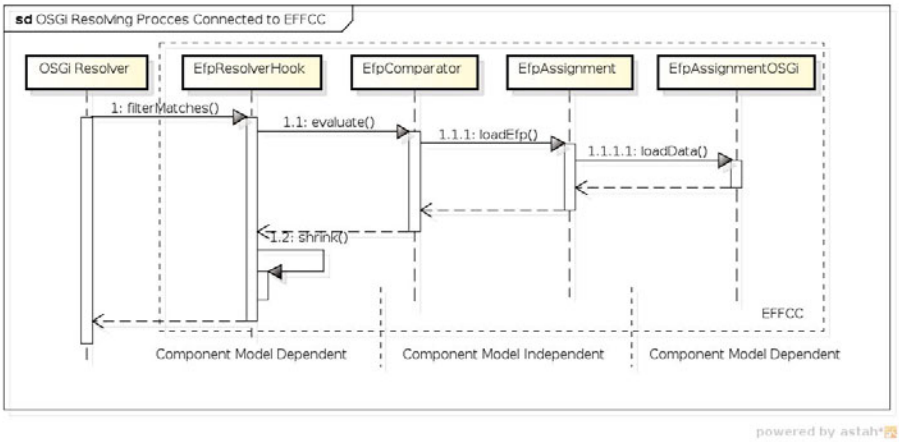

Fig. 5. EFFCC Connected to OSGi Resolver Process

This method is called every time the framework is to bind one feature to another one. The first method argument is a requirement that can be fulfilled by capabilities in the second argument. Concrete implementation may remove an item from the capabilities collection to prevent the binding.

This is the moment at which EFFCC evaluation is called. The sequence of calls is depicted in Fig. 5. Once the `filterMatches` method is called, EFFCC adaptor to OSGi (`EfpResolverHook`) invokes `EfpComparator` that consequently loads EFP data from a bundle using `EfpAssignment`. The data are evaluated and a result is returned to the hook. If the matched EFPs are incompatible, the capabilities collection is shrunk which excludes the capability from the binding.

The only implementation necessary to adapt EFFCC to OSGi have been the `EfpResolverHook` class and an EfpAssignment sub-module (`EfpAssignmentOSGi`) loading the EFP data from the OSGi bundles. All other parts are component model independent and re-usable among variety of different systems. It is also important to highlight that the amount of program code written in `EfpResolverHook` and `EfpAssignmentOSGi` is considerably smaller comparing to the code used in other EFFCC modules. As a result, a noticeable strength of this approach is that it requires only a little code to be implemented while most logic have been already pre-prepared.

The other hook implements the `FindHook` interface. This hook covers service dynamism and it is invoked every time a bundle tries to get a service from its bundle context or a declarative service is to be injected. The only method of the hook is:

```
void find(BundleContext context, String name, String filter,
  boolean allServices, Collection<ServiceReference<?>> references);
```

The implementation of this hook may filter the service references, returning a shrunk collection of available references. The work-flow is similar to that presented for the Resolver Hook: once the EFFCC implementation of `FindHook`

receives a request for a service, the EFFCC modules are invoked to load and compare EFP data. If the EFP data are incompatible, the respective service reference is removed from the `references` collection preventing a bundle to obtain it. Because of the similarity with the Resolver Hook process, a detailed sequence diagram is not provided here.

To sum up, the implementation of the mentioned two hooks covers both static binding of bundle features stated in the manifest file and dynamic finding of services among bundles. Therefore, static features (export/import packages, capabilities) as well as dynamic features (services, declarative services) from Section 3 are covered which fulfils the main goal of this work.

5 Conclusion

This paper has pointed out a discrepancy in the use of extra-functional properties in research and industrial component models caused among other reasons by a weak standardisation of properties' understanding. OSGi has been selected as one of the industrial models and discussion of its possibilities to express EFPs have been provided. The discussion has shown that OSGi is capable of defining certain EFPs, however, their semantic is weak.

The main contribution of this paper is the demonstration of how an existing component model may be enriched with a strong EFP support. It has been demonstrated that our previously proposed mechanism that consolidates EFP understanding among vendors and component models can be used to enhance OSGi with better EFP support. Thanks to the design of the mechanism which separates the component model dependent and independent parts, we could reuse the latter part in full. The former part consists of a semantic rich EFP definitions stored in unified XML format.

In this particular work, two extensions to OSGi have been proposed. First, OSGi parameters, attributes and filters are used to link independent EFP definitions with concrete OSGi bundle features. Secondly, this work uses the concept of OSGi hooks that invokes EFP evaluation as bundles are being resolved or bundle services are being found. Comparing to plain OSGi, this new approach adds better EFP semantics with EFPs consolidated among multiple vendors since EFPs come from the common repository. The means to express EFPs for different applications are also provided using a layered structure of the repository where each layer targets a concrete application.

References

1. Aagedal, J.Ø.: Quality of Service Support in Development of Distributed Systems. Ph.D. thesis, University of Oslo (2001)
2. Becker, S., Koziolek, H., Reussner, R.: The Palladio component model for model-driven performance prediction. Journal of Systems and Software 82(1), 3–22 (2009), special Issue: Software Performance - Modeling and Analysis

3. Bondarev, E., Chaudron, M.R., de With, P.H.: Compositional performance analysis of component-based systems on heterogeneous multiprocessor platforms. In: Proceedings of Euromicro Conference on Software Engineering and Advanced Applications, pp. 81–91. IEEE Computer Society (2006)
4. Botella, P., Burgues, X., Franch, X., Huerta, M., Salazaruml, G.: Modeling non-functional requirements. In: Proceedings of Jornadas de Ingenieria de Requisitos Aplicada JIRA 2001 (2001)
5. Bures, T., Hnetynka, P., Plasil, F.: SOFA 2.0: Balancing advanced features in a hierarchical component model. In: Software Engineering Research, Management and Applications, pp. 40–48. IEEE Computer Society (2006)
6. EJB: Enterprise JavaBeans, Version 3.0. EJB Core Contracts and Requirements. Sun Microsystems (May 2006), JSR220 Final Release
7. Franch, X.: Systematic formulation of non-functional characteristics of software. In: Proceedings of International Conference on Requirements Engineering (ICRE), pp. 174–181. IEEE Computer Society (1998)
8. Genssler, T., Christoph, A., Schulz, B., Winter, M., Stich, C.M., Zeidler, C., Müller, P., Stelter, A., Nierstrasz, O., Ducasse, S., Arevalo, G., Wuyts, R., Liang, P., Schönhage, B., van den Born, R.: PECOS in a nutshell. Pecos Handbook (September 2002)
9. Glinz, M.: On non-functional requirements. In: Requirements Engineering Conference, pp. 21–26. IEEE Computer Society, Los Alamitos (2007)
10. Grosskurth, A., Godfrey, M.W.: A reference architecture for web browsers. In: Proceedings of the 21st IEEE International Conference on Software Maintenance, pp. 661–664. IEEE Computer Society, Washington, DC (2005)
11. Gu, X., Nahrstedt, K., Yuan, W., Wichadakul, D., Xu, D.: An XML-based quality of service enabling language for the web. Journal of Visual Language and Computing, Special Issue on Multimedia Language for the Web 13, 61–95 (2001)
12. Jaaksi, A.: Developing mobile browsers in a product line. IEEE Software 19, 73–80 (2002)
13. Ježek, K., Brada, P.: Correct matching of components with extra-functional properties - a framework applicable to a variety of component models. In: Evaluation of Novel Approaches to Software Engineering (ENASE). SciTePress (2011) ISBN: 978-989-8425-65-2
14. Ježek, K., Brada, P.: Formalisation of a Generic Extra-functional Properties Framework. In: Evaluation of Novel Approaches to Software Engineering. CCIS. Springer, Heidelberg (to be published, 2012)
15. Lamanna, D.D., Skene, J., Emmerich, W.: Slang: A language for defining service level agreements. In: IEEE International Workshop of Future Trends of Distributed Computing Systems, p. 100. IEEE Computer Society (2003)
16. Lamport, L.: Specifying Systems, The TLA+ Language and Tools for Hardware and Software Engineers. Addison-Wesley (2002)
17. Lau, K.-K., Ukis, V.: Defining and Checking Deployment Contracts for Software Components. In: Gorton, I., Heineman, G.T., Crnković, I., Schmidt, H.W., Stafford, J.A., Ren, X.-M., Wallnau, K. (eds.) CBSE 2006. LNCS, vol. 4063, pp. 1–16. Springer, Heidelberg (2006)
18. Mohammad, M., Alagar, V.S.: TADL - An Architecture Description Language for Trustworthy Component-Based Systems. In: Morrison, R., Balasubramaniam, D., Falkner, K. (eds.) ECSA 2008. LNCS, vol. 5292, pp. 290–297. Springer, Heidelberg (2008)

19. Muskens, J., Chaudron, M.R.V., Lukkien, J.J.: A Component Framework for Consumer Electronics Middleware. In: Atkinson, C., Bunse, C., Gross, H.-G., Peper, C. (eds.) Component-Based Software Development for Embedded Systems. LNCS, vol. 3778, pp. 164–184. Springer, Heidelberg (2005)
20. Nierstrasz, O., Arévalo, G., Ducasse, S., Wuyts, R., Gao, X.-X., Müller, P.O., Zeidler, C., Genssler, T., van den Born, R.: A Component Model for Field Devices. In: Bishop, J.M. (ed.) CD 2002. LNCS, vol. 2370, pp. 200–209. Springer, Heidelberg (2002)
21. OMG: MOF 2.0 core. OMG Document ptc/06-01-01 (January 2006)
22. OMG: UML profile for modeling quality of service and fault tolerance characteristics and mechanism specification 1.1. Tech. rep., OMG - Object Management Group (2008), formal/2008-04-05
23. OMG: UML Profile for MARTE: Modeling and Analysis of Real-Time Embedded Systems. OMG (2009), formal/2009-11-02, `http://www.omg.org/spec/MARTE/1.0/PDF` (2010)
24. OSGi: OSGi Service Platform Service Compendium 4.2. The OSGi Alliance (2009), `http://www.osgi.org/Download/Release4V42` (2011)
25. OSGi: OSGi Service Platform Core Specification 4.3. OSGi Aliance (2011), `http://www.osgi.org/`
26. Röttger, S., Zschaler, S.: CQML+: Enhancements to CQML. In: Bruel, J.M. (ed.) Proc. 1st Int'l Workshop on Quality of Service in Component-Based Software Engineering, Toulouse, France, pp. 43–56. Cépaduès-Éditions (June 2003)
27. Sentilles, S., Štěpán, P., Carlson, J., Crnković, I.: Integration of Extra-Functional Properties in Component Models. In: Lewis, G.A., Poernomo, I., Hofmeister, C. (eds.) CBSE 2009. LNCS, vol. 5582, pp. 173–190. Springer, Heidelberg (2009)
28. Spring Comunity: Spring Framework, ver.3, Reference Documentation. SpringSource, ver. 3 edn. (2010), `http://static.springsource.org/spring/-docs/3.0.x/spring-framework-reference/html/`
29. Szyperski, C., Gruntz, D., Murer, S.: Component Software - Beyond Object-Oriented Programming, 2nd edn., 624 pages. Addison-Wesley / ACM Press (2002) ISBN-13: 978-0201745726