Available online at www.sciencedirect.com

SciVerse ScienceDirect

Electronic Notes in Theoretical Computer Science

Electronic Notes in Theoretical Computer Science 279 (2) (2011) 59–73

www.elsevier.com/locate/entcs

ENT: A Generic Meta-Model for the Description of Component-Based Applications

Jaroslav Šnajberk and Přemek Brada

Department of Computer Science and Engineering
Faculty of Applied Sciences
University of West Bohemia
Pilsen, Czech Republic
{snajberk,brada}@kiv.zcu.cz

Abstract

Current best practice in modeling component-based applications is the use of UML extended by a profile. This solution provides a general and common approach of application description and allows to capture some details based on the concrete component model. It has however disadvantages due to the limitations of UML itself, like little scalability or lack of inherent model semantics. In this paper we propose a solution to overcome these limitations, in the form of a meta-model developed directly for the description of components and component-based applications. Its unique aspect is the use of faceted classification to introduce additional semantics and structuring to the derived models. We describe the features and advantages of this meta-model and illustrate its aspects on a model example of a simple OSGi application. At the end of paper we also propose the usage of this meta-model in visualization of component-based applications.

Keywords: meta-model, component, component model, component-based application, UML

1 Introduction

In component-based software development it is important to know for which component framework is the component-based application developed, because the design of single components and the whole application architecture depend on this knowledge. A component framework is an implementation of component model, which means that component model contains specification of how component looks, which types of components exist, how they communicate, how they behave, etc.

When the architect wants to describe a component-based application using a textual or visual representation, there are two options.

(i) General "boxes-and-lines" description (model) of the application. Such model can't describe all aspects of the application as it has to be sufficiently abstract to be general, on the other hand it can be used for any component-based application and any component model.

1571-0661
doi:10.1016/j.entcs.2011.11.012

(ii) Component model-specific description of the application, which is bound to one concrete component model. Such description can express all the aspects of the application as it is developed for the concrete component model but it can't be used for applications in any other component model.

The component model-specific description is useful only for the exchange of information between the domain experts of the given component model. The specifics of the particular component model make it difficult to read and understand the architecture of application for experts from different domains. Moreover component models often use their own graphic notation (e.g. SaveCCM [4]), making it hard to read for experts not familiar with it.

A general description of application is on the other hand useful for exchange of information between domains but it provides no details bound to the specifics of concrete component model(s) and thus can only provide shallow understanding of the component-based application. The best example of these general description languages is UML 2.0 [7] and its underlying component model. There are therefore component constructs that are very difficult to model in UML, for example events, which can be important even on component diagram level to show the indirect connections between components.

The UML 2.0 component model also has only three groups of elements - attributes, operations, imported/exported interfaces, though stereotypes doesn't have to provide sufficient grouping. Using an extended UML with all stereotypes has negative impact on the complexity and clarity of the final diagrams. UML also suffers from insufficient content awareness which makes it difficult to interpret model data. These deficiencies are also present in UML exported into the XMI format used for the exchange of model information between machines.

The problem therefore is the lack of an approach which would be:

(i) General, so it can describe applications of any component model in a similar fashion.

(ii) Flexible, to provide user implementation details and specifics of concrete component model.

(iii) Rich, with eligible groupings to provide more precise classification of elements than just attributes, operations, and imported/exported interfaces.

(iv) Content aware, so machines can understand the information exchanged.

(v) Customizable, so users can view or analyze only information they are interested in.

(vi) Usable by various roles in the software development process (architect, application assembler, etc.) so they can understand and modify the relevant parts of a model.

1.1 Goal of the Paper

We believe that development of a domain-specific alternative is a better solution than efforts to extend UML for modeling component-based applications.

The advantages for developing a new meta-model are:

- Freedom from legacy models inadequate for current and future needs.
- Developed directly for the problem domain, thus clearer and easier use.
- Visual representation can use advanced features brought by the meta-model.

Potentially problematic on the other hand are the following aspects:

- Divergence from UML which is a standard language for most software engineers may decrease understanding and acceptance.
- Since component models are very diverse, a common meta-model may not sufficiently capture all their idiosyncrasies.

In this paper we therefore present a new meta-model called ENT designed directly for the description of component-based application. It aims to conform to all the requirements listed above while trying to avoid the problematic aspects. While there is not enough place to provide its full description together with a complete case study, an example application is used to show how its parts can be described in ENT.

The following section surveys the related work in the field of component meta-models. Section 3 provides an overview of our model, sections 4–6 describe in detail its formal structure. The Conclusion contains also a brief status of current implementation efforts related to the ENT meta-model.

2 Related work

The area of software composition described in [12] has several common features with our work. The author of this paper is concerned with the creation of generic tool capable of handling different component models used for software composition. The author also uses generic approach and takes advantage of description of component models to achieve the goal of her work.

UML [7] was designed to be a universal and general modeling notation backed by a meta-model, but it also supports extensions that makes it more usable when user need to add some details to the model. UML 2.0 supports extensions through UML profiles, which uses stereotypes, tagged values and constraints. Using UML 2.0 and extending its meta-model through profiles is best practice for description of component-based applications. These efforts are well illustrated on earlier mentioned SaveCCM, which already has its UML profile [10].

Extending UML through profiles is not the only way how it can be extended. It is also possible to extend the core meta-model of UML as described in [9]. The author used this "heavyweight" approach to provide better description of the C3 architectural style described in [15]. However this approach does not meet our requirements on customizability and content-awareness.

There already are research works covering the area of component model description. For instance, Crnkovic et al [3] describe advanced framework able to classify any component model from various angles. On the other hand, Medvidovic [6] uses

ADL for description purposes. While these articles capture a lot of of experience, their aim is only to describe the features of component models without developing a meta-model based on the results.

In [14] the author describes the need for component meta-model capable of modeling the various existing component models to unify components into modeling paradigm. This work however doesn't consider modeling of component-based applications.

3 ENT meta-model

The ENT meta-model is a general model defining the structures of component models and component-based applications; see [1] for a previous version of the model. Its distinguishing characteristics is the use of the faceted classification approach [13] to represent components in a way which is flexible enough for users with different interest. A key structure used in the meta-model is the ENT *classifier*, which is a tuple of identifiers which characterise any component interface element from several orthogonal aspects related to user perception.

The ENT meta-model is structured into two levels: on the *component model level* the main characteristic features of a given component model are defined, on the *application level* the concrete components, their interface elements and their bindings in an application are captured.

3.1 Overview of the Meta-Model

Let us start with a brief overview of the meta-model in plain English; the following subsections will then provide the exact definitions. The structural hierarchy of the meta-model starts with a *component model* as a set of component types. A *component type* is defined by a complete minimal set of *definitions of traits* which describe the possible kinds of interface elements which the component type can support. The traits declare the language meta-type and ENT classifier of these elements, capturing their commonalities like the users do.

As an example, there is only one component type in OSGi called "bundle", with ENT definition described in section 4.1. The ENT meta-model enforces this structuring of component interface (as opposed to a flat collection of items, cf. Figure 7) because it is quite natural for developers to think of e.g. all component's provided services as a group, regardless of their concrete interface types and location in the specification source. In Enterprise JavaBeans on the other hand several different component types can be identified – SessionBeans, MessageDrivenBeans or Entities. The component types, as well as trait's characteristic meta-type and classifier, are therefore based on a human analysis of the concrete component model and its component specification language(s).

At the level of a concrete application, a *component* implementation then conforms to one of the component types defined by its component model. Each component has a set of concrete *interface elements* manifest on the visible surface of its black box. These elements populate some or all of its actual *traits*, which again

conform to the corresponding trait definitions. The component also holds the *connections* of its elements to the counterpart elements in client and/or supplier components, and – in case of hierarchical component models – may list the *sub-components* it is composed from.

In many component models, several run-time *instances* of a concrete component can be created, each with unique identity. The ENT meta-model does not deal with component instances because its domain is the level of component models and component application design, rather than the run-time instantiation level.

The rest of this paper provides a formal definition of these structures, in a top-down fashion.

3.2 Classification System

The ENT meta-model uses a faceted classification system for characterising various aspects of component interface elements, with eight facets called "dimensions". These dimensions have predefined values and each dimension represents a different point of view on a component.

Definition 3.1 The **ENT classification system** is a collection of facets $Dimensions_{ENT} = \{dim_i, i = 1..8\}$ where the dim_i are:

- Nature = {syntax, semantics, extra-functional}
- Kind = {operational, data}
- Role = {provided, required, neutral}
- Granularity = {item, structure, compound}
- Construct = {constant, instance, type}
- Presence = {mandatory, permanent, optional}
- Arity = {single, multiple}
- Lifecycle = {development, assembly, deployment, setup, runtime}

The **ENT classifier** is a tuple $K = (k_1, k_2, ..., k_D)$ where $k_i \subseteq dim_i, dim_i \in Dimensions_{ENT}, D =\mid Dimensions_{ENT} \mid$.

This classification system and the classifier structure are used in the trait and category set definitions, presented in the subsequent paragraphs.

4 The Component Model Level

Identification of different component models and the types of components they define forms the top level of the meta-model.

Definition 4.1 A **component model** is the pair $M = (name, C_S)$ where $name \in$ *Identifiers* is the model's name and $C_S = \{C_{i,def}\}$ is a set of component type definitions.

Component types consist mainly of trait definitions that declare the kinds of

elements (features) the concrete components can have on their surface. Traits thus helps to fully characterize components of such type. For example, OSGi components (cf. Section 4.1.2) have traits *Export packages*, *Provided services*, *Import packages*, etc.

Definition 4.2 A **component type** is a tuple $C^{def} = (name, tagset, T)$ where $name \in Identifiers$ is the name of the component type, $tagset = \{tag_i\}$ is a finite set of extra type information items ("tags"), and the $T = \{T_i^{def}\}$ where i is a finite index is the set of the component type's trait definitions (also called "trait set").

The tags in the tagset are triples $tag_i = (name_i, valset_i, d_i)$ where $name_i \in Identifiers$, $valset_i$ is the set of its possible values, and $d_i \in valset_i \cup \{\epsilon\}$ is the default value (ϵ means "no default"). Tags capture pieces of information that are important for the component model and cannot be described using traits, e.g. component's persistence and transactionality as used in Enterprise JavaBeans.

The component types of one component model must be distinct: $\forall C_i, C_j \in M.C_S, i \neq j : C_i \neq C_j \Rightarrow C_i.name \neq C_j.name$.

Definition 4.3 A **trait definition** is a tuple $T^{def} = (name, metatype, K, tagset, extent)$ where $name \in Identifiers$ is the trait's name, $metatype \in Identifiers$ is the meta-type of the component interface elements grouped by this trait, K is their ENT classifier, $tagset = \{tag_i\}$ is the finite set of allowed tags of these elements, and $extent \in \{one, many\}$ defines the maximum number of elements in the trait [1].

Consistency rule: Traits of one component type must be distinguishable by name, i.e. $\forall T_i^{def}, T_j^{def} \in C^{def}.T, i \neq j : T_i^{def}.name \neq T_j^{def}.name$.

The *metatype* of the trait's elements (such as "interface" or "event") may be related to or derived from the name of the corresponding non-terminal symbol in the grammar of the component's interface specification language particular for the trait. The *tagset* has the same definition and meaning as that of the component, described above, except that the concrete tag values are meant to be assigned to individual elements (not to the trait).

The ENT classifier K describes the classification properties of the trait's elements – this is a unique aspect and key concept of the ENT meta-model, capturing the human-perceived similarity of the elements grouped by a trait.

Concerning the consistency rule, it is actually preferred that traits are distinguished by their classifiers only, i.e. the following stronger assertion holds: $\forall T_i^{def}, T_j^{def} \in C^{def}.T, i \neq j : T_i^{def} \neq T_j^{def} \Rightarrow T_i^{def}.name \neq T_j^{def}.name$. There may however be cases when the ENT classification scheme does not provide enough characteristics to reliably distinguish traits. Then, distinguishing by names is the only practical option and this is reflected in the definition.

When the component model level description is designed according to the ENT meta-model, a set of data structures for modeling component-based applications is prepared. These data structures can fully describe all components implemented

[1] For simplicity, we do not use concrete numbers, ranges and similar features in extent specification.

in the given component model and have to be created manually after analysis of modeled component model. The following section illustrates the ENT component model definition for the OSGi framework.

4.1 Example: The OSGi Component Model and Application

To illustrate the ENT structures, this section presents a subset of the representation of the OSGi component model [8] plus examples of behavioural and extra-funcional element traits. OSGi was chosen for its industrial relevance, simplicity and ubiquity.

4.1.1 Component Types

OSGi has only one component type called **Bundle**. Bundle can have two additional tags originated in manifest file.

(i) **Bundle**
- **tagset**: symbolic_name, version
- **T**: { export_packages, import_packages, provided_services, required_services, native_code, require_bundles, required_execution_environment, use_packages}

4.1.2 Trait Definitions

For demonstration purposes we provide the definitions of just four traits here, see [16] for a complete analysis of OSGi ENT representation:

(i) **export_packages**
- **metatype**: package
- **K**: ({syntax}, {operational}, {provided}, {structure}, {type}, {permanent}, {multiple}, Lifecycle)
- **tagset**: version, parameters
- **extent**: many

(ii) **import_packages**
- **metatype**: package
- **K**: ({syntax}, {operational}, {required}, {structure}, {type}, {permanent}, {single}, Lifecycle)
- **tagset**: bundle_symbolic_name, bundle_version, kind, version_range
- **extent**: many

(iii) **provided_services**
- **metatype**: interface
- **K**: ({syntax}, {operational}, {provided}, {item}, {instance}, {optional}, {single}, Lifecycle)
- **tagset**: service_filter
- **extent**: many

(iv) **required_services**
- **metatype**: interface
- **K**: ({syntax}, {operational}, {required}, {item}, {instance}, {optional}, {multiple}, Lifecycle)

- **tagset**: service_filter, service_arity
- **extent**: many

4.1.3 Behaviour and Extra-Functional Properties

Traits can also represent other than functional elements, for example a quality of service aspect (e.g. [5]) or the expected call sequence protocol [11]. These traits must have value *semantics* respectively *extra-functional* in the dimension *Nature* of the ENT Classification. Sample trait definition for such elements are provided below:

(i) **response**
- **metatype**: attribute
- **K**: ({extra-functional}, {data}, {provided}, {item}, {constant}, {mandatory}, {single}, {runtime})
- **tagset**: ∅
- **extent**: many

(ii) **protocol**
- **metatype**: regular-expression
- **K**: ({extra-functional}, {operational}, {provided}, {structure}, {type}, {optional}, {single}, {assembly, runtime})
- **tagset**: ∅
- **extent**: one

4.1.4 Example OSGi Application

In the subsequent sections we will refer to (parts of) a simple example OSGi application called Parking Lot. It consists of four components as illustrated in Figure 1, the architecture should be self-descriptive.

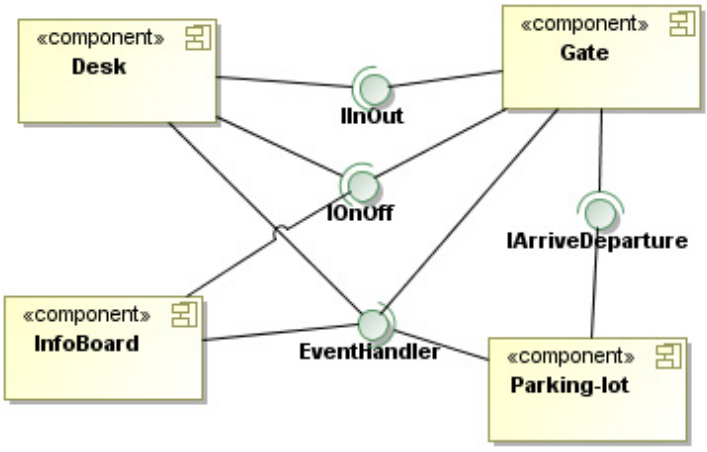

Fig. 1. Component application example — Parking Lot (OSGi application)

5 Application Level

This level of the ENT meta-model provides modeling constructs for concrete components and applications built from them. The component model level has to be already defined because the application level references its elements. These references assign meaning to the application elements; in particular, the set of traits of a concrete component is gained by assigning it the corresponding component type.

Definition 5.1 A **component application** is a direct acyclic graph $A = (C, B, m)$ where $C = \{c_i, i \in \mathbb{N}\}$ are components, $B = \{b_i, i \in \mathbb{N}\}$ their bindings, and $m \in C$ is a main component. We use the term **application context** for a set of all components $A^* = \{c_i, i \in \mathbb{N}\}, A.C \subseteq A^*$ existing in the environment where the component application is deployed.

A *consistent (resolved) application* is such that has all non-optional required elements bound to provided ones within the given context and all its components' inheritance parents exist in the context.

We do not model additional pieces of information associated with applications, like configuration properties, access control lists, and similar – these are used at run-time which is out of scope for ENT meta-model.

```
Manifest-Version: 1.0
Bundle-ManifestVersion: 2
Bundle-Name: Gate
Bundle-SymbolicName: Gate
Bundle-Version: 1.0.0
Bundle-RequiredExecutionEnvironment: JavaSE-1.6
Require-Bundle: Parkinglot;version="1.0.0"
Import-Package: cz.zcu.kiv.parkinglot.parkinglot;version="1.3.0",
 org.osgi.service.event;version="1.2.0"
Export-Package: cz.zcu.kiv.parkinglot.gate
```

Fig. 2. Manifest file for Gate bundle

5.1 Individual Components

In this section an example of the Gate bundle (see Figure 1) will help to ilustrate the representation of component information in the ENT meta-model structure. The manifest file of this bundle is present in Figure 2.

Definition 5.2 A concrete **component** is a tuple $c = (name,\ C^{def},\ G,\ T, P, S)$ where $name$ is the component's name, C^{def} is the (reference to) the appropriate component type, $G = \{(name_i, value_i)\}$ is the set of its tags, $T = \{t_i\}$ is the concrete trait set of the component with traits as defined below, P is a finite, possibly empty set of (references to) concrete components which are c's inheritance parents, and S is a finite, possibly empty set of c's sub-components and their delegation bindings (see subsection 5.3 below).

The following consistency rules must hold:

- $\forall (n_i, v_i) \in c.G\ \exists tag_j \in C^{def}.tagset : n_i = tag_j.name \wedge v_i \in tag_j.valset$, i.e. tags are taken from component's type tagset;
- $\forall p \in P : p.C^{def} = c.C^{def}$, i.e. the parents are of the same component type.

It is also natural that both c and all its sub-components belong to the same component model.

By component interface **element set** $E(c)$ we will understand the set of all specification elements (as defined below) contained in the specification of concrete component c. In case of component inheritance, it is the union of element sets of the transitive closure of c and all its inheritance parents. Subsets $E^P(c)$ and $E^R(c)$ of the element set denote the *provided* and *required elements* of c where it holds that $E^P(c) \cap E^R(c) = \emptyset \wedge E^P(c) \cup E^R(c) = E(c)$.

This representation is a complete model of a concrete component, by which we mean that the original specification of the component can be fully reconstructed from the representation.

Concrete component's trait is a named set of its interface elements with the same meaning, as given by their meta-type and ENT classifier.

Definition 5.3 A component interface **trait** (of a concrete component c) is a pair $t = (T^{def}, E)$ where T^{def} is a (reference to) the trait definition and $E \subseteq E(c)$ is a subset of component's interface elements.

Consistency rules: It must hold for a given component c that

- $E(c) = \bigcup_i t_i.E, t_i \in c.T$ and $\forall t_i, t_j \in c.T, t_i \neq t_j : t_i.E \cap t_j.E = \emptyset$, i.e. that the traits together contain all its elements without duplicates
- $\forall t \in c.T : t.T^{def} \in c.C^{def}.T$, i.e. traits are defined by its component type.

Traits group the interface elements of a component even if in the source these may be specified in various places – either within one specification file (e.g. a SOFA ADL, disregarding the particular ordering of declarations), or even in several ones (e.g. OSGi manifest plus declarative services' `component.xml`).

Traits alone do not say anything about the features of the particular component – they have only grouping purpose and through the reference to their trait definitions give meaning to all interface elements contained in it.

T^{def} = imported_packages,
$E = \{cz.zcu.kiv.parkinglot.parkinglot, org.osgi.service.event\}$

Fig. 3. The *imported_packages* trait of the *Gate* bundle in ENT representation

Definition 5.4 An **interface element** e of a concrete component c with specification written in language L is a tuple $e = (name, type, G)$ where $name \in Identifiers \cup \{\epsilon\}$ is the (possibly empty) element's name, $type \in L$ is a language phrase denoting its type, and $G = \{(n, v)\} \subset Identifiers \times Identifiers$ is the (possibly empty) set of element's concrete tags.

Consistency rule: $\forall e \in t.E, \forall g \in e.G\ \exists d \in t.T^{def}.tagset : g.n = d.name \wedge g.v \in d.valset$, i.e. the tag values of elements in trait t must be taken from the value set in the trait definition.

A specification element is a complete representation of one component interface feature identified by language *name* and/or *type*. All its parts are directly related

to its specification source code (the human classification and understanding of an element is attached to its containing trait). Operations on them are therefore subject to the syntax and typing rules of the language L used for the component interface specification.

The tags represent additional semantic or other extra-functional information pertaining to the particular element (not to its type), like the `readonly` or `final static` keywords. They are important if one needs to e.g. precisely compare two elements or re-generate a valid source code for the element. Note that the element's tags are defined in its trait definition, since all elements of one trait necessarily have the same set of tags.

$name$ = cz.zcu.kiv.parkinglot.parkinglot,
$type$ = package,
$G = \{(version, 1.3.0)\}$

Fig. 4. The *parkinglog* element of the *imported_packages* trait in ENT representation

5.2 Component Bindings

To model bindings between components within the application, we use a set of connections which keep information about source element, target element and which direction information flows (provided / required).

Definition 5.5 Let us have a consistent component application A. The **application connection set** is a finite set $B = \{b_i, b \in \mathbb{N}\}$ where $b = (e^s, e^t) : \exists c_i, c_j \in A.C : e^s \in E^R(c_i), e^t \in E^P(c_j)$ i.e. the connections (arcs in the application graph) lead from required to provided elements.

The **connection set of a component** c is a set of connections which have incidence with the component: $B(c) \subseteq B$, $\forall b \in B(c)$ either $b.e^s \in E^R(c)$ or $b.e^t \in E^P(c)$.

The connection set of a component makes it possible for every component to be aware of all connections realized by its elements, both provided and required.

e^s = Gate::exported_packages::cz.zcu.kiv.parkinglot.gate,
e^t = Desk::imported_packages::cz.zcu.kiv.parkinglot.gate

Fig. 5. The service *cz.zcu.kiv.parkinglot.gate* bound to bundle *Desk* in ENT representation

5.3 Hierarchical Components

Some component models such as SOFA [2] use hierarchical decomposition which means that composite components can be recursively composed from other components. Components which are not composed from any other components are called primitive components.

For composite components, a special set of connections needs to be modeled: the subsumption and delegation bindings between the composite component interface elements and its sub-components.

Definition 5.6 For a given component c in application A, the pair $S = (S^c, S^d)$ in component's tuple captures the **inner architecture** of its composition. $S^c \subset A.C, c \notin S^C$ is the set of sub-components. The S^d is a set of delegate/subsume binding pairs, $S^d = \{(e^c, e^s) \mid e^c \in E(c),\ e^s \in E(s) \cdot s \in S^c\}$, i.e. the e^c and e^s elements belong to the composite component and one of its sub-components, respectively.

Consistency rule (added to those in Definition 5.2): $\forall (e^c, e^s) \in S^d : e^c \in c.t_m, e^s \in s.t_n, t_m.T^{def} = t_n.T^{def}$, i.e. elements in subsume/delegate pairs belong to traits with the same trait definition.

For example, suppose that the *Parking-lot* component from Figure 1 was in fact hierarchical. The handling of client's requests on the `IArriveDeparture` element could be delegated to an equally-typed element in a *Arrivals* sub-component. This would be expressed as an inner architectural binding *(Parking-lot::IArriveDeparture, Arrivals::IArriveDeparture)*. Both elements would belong to the "provided-services" trait of their components.

6 Structuring Level: Category sets

Some traits and elements could be at particular times considered as unwanted information when reading a model of component-based application. For example, software architects are interested in other information than programmers. By using all information contained in both layers of an ENT-based model there could also be a danger of confusion when representing big and complex applications.

After representing a component-based application according to the Application level, the ENT classifier allows us to organize the model information using so called *category sets*. These sets are defined by selector operators on the trait classification which say how to group and display traits.

Definition 6.1 The **category set** over an ENT model is a pair $Catset = (name, \{(c, K, f)\})$ where $name, c \in Identifiers$ are the names of the category set and its categories, and $f = K \times T^{def} \rightarrow boolean$ is a function which determines whether the given trait definition fits the (partial) classifier K.

For example, the E-N-T category set defined in Figure 6 has three groups. In the first group are elements that are contained in traits with $role = \{provided\}$ in their classifier (this means those elements which the component *exports*). Required elements are similarly grouped as *needs* and elements that are both provided and required are called *ties*. This category set gave the name to the ENT meta-model, as it captures the most fundamental split of any component's interface element set.

More category sets are presented in [1], and category sets can be created by any user of the ENT meta-model if another point of view is needed.

6.1 Visualization using the ENT meta-model

The ENT meta-model in general and the category sets in particular have a big use in visualization of components. Component can be visualized very similarly as in UML but the surface of component can be displayed in a tree structure governed by element grouping into traits and category sets. For example, there are three different views on the same OSGi bundle in Figure 7. The ENT and ITC category sets show all traits of bundle component type in different groupings, while the II category set is very selective and shows only imported instances.

The possibilities of grouping and filtering by applying a category set layer over an application model are very rich. Since category sets can be defined by user and switched between easily, this is one of the most useful features of the ENT meta-model itself. This approach ensures that user see only what he wants to consider at the moment.

It is important to note that model data provided by an ENT-based model can be used by a number of different visualization styles; the visualization presented in Figure 7 is only one of the many.

7 Conclusion

In this article we proposed a new meta-model, called ENT, for the description of components and whole component-based applications. This meta-model takes the

E-N-T (Exports-Needs-Ties)
$E : K = \{(role = \{provided\})\}, f = matches$
$N : K = \{(role = \{required\})\}, f = matches$
$T : K = \{(role = \{provided, required\})\}, f = matches$

Fig. 6. The ENT category set

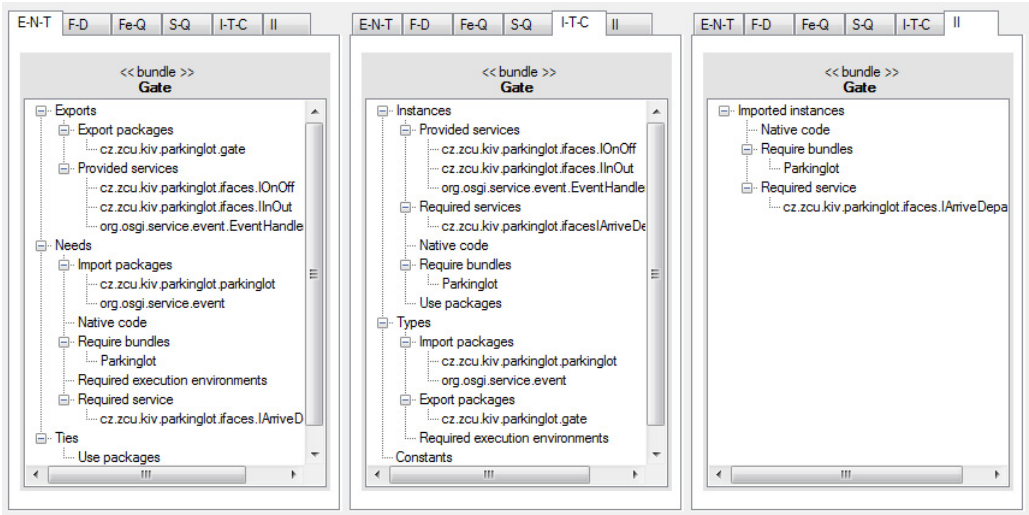

Fig. 7. Visualization of a component using Category Sets

advantage of the close relation between a component model and its real components. It provides structures for the description of component types supported by a component model and, more importantly, it groups component surface elements to so called traits that use a custom classification system to capture their human-perceived characteristics. The classification provides enough information to machine interpret the resulting component representations in different ways. The ENT meta-model also captures the relations between components and supports hierarchical decomposition of components.

The ENT meta-model addresses the requirements stated in the Introduction through the following properties:

(i) generality: verified support for a wide spectrum of component models [1] and application structures;

(ii) flexibility: traits are able to represent various component elements, tags model implementation details particular to a concrete component model;

(iii) richness: the combination of the trait structure and ENT classifier provides a wealth of information;

(iv) content awareness: thanks to the ENT Classification system, the character of traits can be described and machine interpreted;

(v) customization: category sets enable the users to filter and group component elements and their details based on actual needs;

(vi) role support: the information in ENT-based models can be filtered for concrete roles through the Lifecycle classification facet.

We have successfully created an implementation of the ENT meta-model using model driven development, with a XMI format of model definitions. A loader of OSGi bundles into the ENT-based model data structures is already implemented and loaders for other frameworks (EJB and SOFA) are the subject of implementation at the time of writing of this article.

The ENT meta-model is expected to be used in advanced component application visualizations and a corresponding implementation of a basic tool able to use the advantages of this meta-model is under way. In a longer term we would like to improve this visualization tool by conducting research on different kinds of visual representation and maximizing the possibilities of the ENT meta-model used as data layer.

Acknowledgement

The work was partially supported by the UWB grant SGS-2010-028 Advanced Computer and Information Systems.

References

[1] Brada, P., *The ENT meta-model of component interface, version 2*, Technical Report DCSE/TR-2004-14, Department of Computer Science and Engineering, University of West Bohemia (2004), available

at http://www.kiv.zcu.cz/publications/.

[2] Bures, T., P. Hnetynka and F. Plasil, *SOFA 2.0: Balancing advanced features in a hierarchical component model*, in: *Proceedings of SERA2006* (2006), pp. 40–48.

[3] Crnkovic, I., M. Chaudron, S. Sentilles and A. Vulgarakis, *A classification framework for component models*, in: *Proceedings of the 7th Conference on Software Engineering and Practice in Sweden*, 2007.

[4] Hansson, H., M. Akerholm, I. Crnkovic and M. Törngren, *SaveCCM - a component model for safety-critical real-time systems*, in: *Proceedings of the 30th EUROMICRO Conference* (2004), pp. 627–635.

[5] Koziolek, H., *Performance evaluation of component-based software systems: A survey*, Performance Evaluation **67** (2010), pp. 634–658.

[6] Medvidovic, N. and R. N. Taylor, *A classification and comparison framework for software architecture description languages*, IEEE Transactions on Software Engineering **26** (2000), pp. 70–93.

[7] Object Management Group, *UML superstructure specification*, OMG specification formal/2009-02-02 (2009).

[8] The OSGi Alliance, "OSGi Service Platform Core Specification," (2009), release 4, Version 4.2.

[9] Perez-Martinez, J. E., *Heavyweight extensions to the uml metamodel to describe the c3 architectural style*, ACM SIGSOFT Software Engineering Notes **28** (2003), p. 5.

[10] Petricic, A., L. Lednicki and I. Crnkovic, *Using UML for domain-specific component models*, in: *Proceedings of the 14th International Workshop on Component-Oriented Programming*, 2009.

[11] Plášil, F. and S. Višnovský, *Behavior protocols for software components*, IEEE Transactions on Software Engineering **28** (2002).

[12] Presso, M. J., *Declarative descriptions of component models as a generic support for software composition*, in: *Workshop on Component-Oriented Programming (WCOP'00)*, Nice, France, 2000, position Paper.

[13] Prieto-Diaz, R. and P. Freeman, *Classifying software for reusability*, IEEE Software **18** (1987).

[14] Rastofer, U., *Modelling with components - towards a unified component meta-model*, in: *Proceedings of the Model-based Software Reuse Workshop*, Springer-Verlag, Malaga, Spain, 2002 .

[15] Shaw, M. and D. Garlan, "Software architecture. Perspectives on an emerging discipline." Prentice Hall Publishing, 1996.

[16] Valenta, L. and P. Brada, *OSGi component substitutability using enhanced ent metamodel implementation*, Technical Report DCSE/TR-2006-05, Department of Computer Science and Engineering, University of West Bohemia (2006).