Implementation of a data layer for the visualization 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 approaches to the visualization of component-based applications mostly use only associations and dependencies between components and provide limited supplementary information. In this paper, we introduce a data layer that is able to store and later present more information about component elements which are bound together, and through this knowledge provide more understanding about the component itself. These information could be presented in different ways to provide different views for component software developers, designers, and architects. This data layer is general and is able to visualize component based applications of any component model. It is presented here together with its structure, implementation and tooling. We share experiences obtained in the process of designing and implementing this layer. Special care is given to the implementation details which were solved in the process and relevant tools like MOF and EMF are presented. Results from the test application are also part of this paper.*

1 Introduction

Many component based applications are developed in different component environments. Component models like EJB [8], CORBA [6], OSGi [9] and more can be found in commercial applications and even more component models – for example SOFA [19] or CoSi [20] – are the subject of research.

The diversity of component models or even approaches to components [1] poses problem when one needs to visually represent a component or component-based application. UML 2.0 [5] component diagram is often used for the visualization of component dependencies. The problem is that dependencies alone do not provide much information about the component itself.

It would be beneficial if there was a way to know more about the elements that make up the bindings between components. Based on the diversity of component models we can say these elements should bear the information relevant in the concrete component model. This means that general visualization approaches like UML can't represent this extra information.

This problem is best presented on a short example of a CORBA component. Let us presume there is component as shown in Example 1 and we would like to display it in diagram. In UML this component would look like in Figure 1.

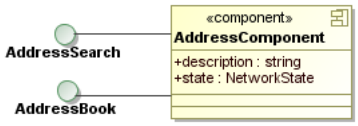

Fig. 1. Address Component in UML.

Example 1.

```
component AddressComponent
{
 // attributes
 readonly attribute string description;
 readonly attribute NetworkState state;

 // facets
 provides AddressBook book;
 provides AddressSearch search;

 // events
 publishes ChangeState stateNotify;
}
```

UML only supports displaying attributes and interfaces the component provides. The CORBA-specific flag "readonly" assigned to the attributes is missing on this diagram and could only be added using stereotypes, which would make the diagram clumsy. Events which the component publishes bring difficulties for the UML component diagram, because it provides no simple way how to represent them.

If we would like to capture such additional information in the visualization of component based applications, appropriate specific meta information would need to be available for its concrete component model. That would lead to as many representations as there are component models, leading to significant fragmentation of visual notations. A better solution is to have a generic meta-model which can be instantiated for individual component models. This means a new data

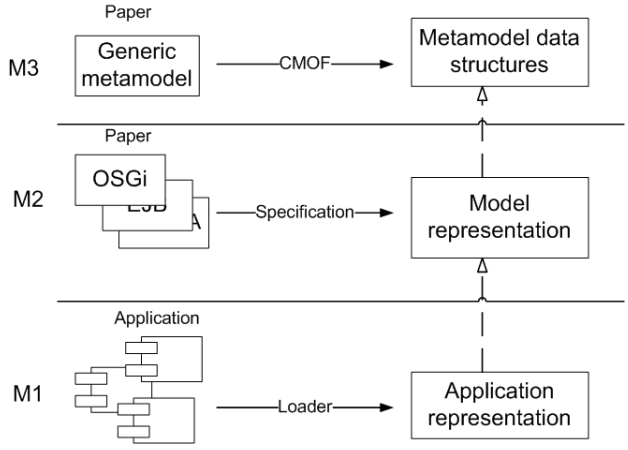

Fig. 2. ENT data container.

layer implementing this meta-meta level has to be designed. The representation of this problem based on MOF (Meta Object Facility) layers [4] is in Figure 2.

On the M3 level there is a data structure able to hold information about both component models and component based application. On the M2 level there is a definition of a component model (defined manually for each component model to provide meaningful representation) which explains the information stored on the M1 level. The M1 level should support automated walking through component-based applications.

In [2] we introduced the ENT meta-model which brought understandable component representation for both automated software agents and humans. It contains support for a classification applicable to component models, description of components and several views to represent this data based on user's point of interest.

1.1 Goal and structure of this paper

In this paper we focus on the design and implementation of a data layer meeting the above requirements on storing and using of these additional information for visualization of component-based applications. The layer implements the ENT meta-model as a MOF structure, using Eclipse Modeling Framework as the underlying technology. Besides describing the data layer itself, we also want share the experiences gained in the process of designing its implementation.

In the next section we briefly review the related standards and research works in the areas of (component) meta-modeling and visual representation. An overview of ENT meta-model is provided in Section 3 to understand the rest of the paper, including the extensions introduced for the description of component models and component based applications.

To build a data container according to the ENT specification, a MOF model [4] was created in which key features of the meta-model were identified and model elements were designed. Implementation characteristics were considered in this phase and the meta-model was slightly modified. The structure of the MOF model and the description of design steps can be found in Section 4.

For its good fit with the data layer needs, the Eclipse Modelling Framework (EMF) was chosen as a model generator. More information about EMF, the generated model and editor can be found in Section 5.

As a proof that the MOF model of ENT meta-model was designed correctly, we implemented representations of several component models. The list of these component models and a case study can be found in Section 6.

2 Component meta-modeling

The diversity of component models or even approaches to components [1] has been mentioned in several studies, e.g. [12, 13]. There are differences in terms of the target use of the component model (desktop GUI, enterprise applications, embedded or real-time systems), the richness of the interface contract type (from a single interface through a set similar to CORBA components, to an extensible model as represented by the iPOJO [14] research framework) as well as between flat models like OSGi and hierarchical ones like SOFA.

The domain of meta-models is best represented by the Meta-Object Facility (MOF) standard [4]. As described in the Introduction, it uses a layered approach to create progressively more specific structures defining the terms of a particular domain, their attributes and relationships. MOF itself has been a subject of rich research. An interesting contribution is Poernonomo's work [15] on providing type-theoretical foundation for the meta-models.

UML [5] component diagram is often used to visualize component based applications and since version 2.0 of the notation it doesn't suffer from problems presented in [17] – mainly the inability to clearly distinguish provided and required interfaces. The UML component diagram is nowadays a common tool for the visualization of component dependencies. An alternative visualization of component-based applications is presented in [16], this approach supports EFP (extra functional properties) on top of a classical component diagram.

3 Overview of the ENT meta-model

The ENT meta-model is a MOF M3 model whose main characteristic is the use of faceted classification

approach [3] to classify characteristic traits of component models. 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 and the characteristic traits of components defined in this model are classified. On the *Application level* the previous definitions are used and the interface elements belonging to individual components are identified. To support additional information, tags are provided which can be added to components or single component elements. The faceted view is used to represent components in way better readable to humans.

For complete ENT meta-model specification, please refer to [2]. Compared to this base specification the meta-model was extended in this work to support relations and dependencies between components, see Section 3.5.

3.1 Classification system

The ENT classification system has eight facets called "dimensions". These dimensions have predefined values and each dimension represents a different point of view on a component. Some facets can have more then one value, for instance Role which says if an element is provided or required – in some cases an element can exhibit both provided and required roles, as e.g. the SOFA behaviour protocol [18].

- Nature = {syntax, semantics, nonfunctional}
- 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}

3.2 Component model level

Complete characteristic features of a given component model are identified on this level of understanding.

Identification of different component types is the first step, because the component model consists of one or more component types. As an example, there is only one component type in OSGi (called Bundle); in EJB on the other hand several different component types can be identified because EJB applications can be built from SessionBeans, EntityBeans or MessageDrivenBeans.

Every component type has its traits definitions that define the kinds of elements (features) the concrete component can have on its surface. Traits thus helps to fully characterize component of such a type. Each trait definition is classified using ENT classification giving different meaning to these trait definitions. For example, CORBA components (cf. Example 1) have traits *facets* (provided interfaces), *receptacles* (required interfaces), *event sinks*, etc.

For other information that are important for the component model and cannot be described using traits, tags are used. Tags can expand information about component types or about elements in traits, for instance to keep track about version, accessibility, range and other additional parameters. Tag definitions are defined on the component model level in order to be available on the application level.

When the component model level is designed, set of data structures for its component-based applications is prepared. These data structures can fully describe all applications implemented in the given component model.

3.3 Application level

Components, from which an application is built from, are represented on this level. The component model has to be already defined on the component model level because the application level references its elements. By creating these references on higher level, the meaning is given to the application elements.

It means a concrete component is assigned a corresponding component type and based on that, a set of its traits is gained. Traits alone do not say anything about the particular component, but elements that belong to the given trait do. Each trait has its own element set – the interfaces, classes, events, etc. found on the component's surface. The component is thus described by several sets of elements grouped together by their characteristic traits. The trait has only grouping purpose and through the reference to its trait definition gives meaning to all elements contained in it. Concrete values of tags can be set on the component and its elements, thus providing their more precise description.

For example, the "facets" trait of the component from Example 1 is a set { *(book,AddressBook), (search, AddressSearch)* } and the *(description,string)* element has a tag set { *(access,readonly)* }.

3.4 Category sets

The level of traits and elements could contain a lot of unwanted information for some sorts of users. For example software architects are interested in other information than programmers of component-based applications. By using such data layer there could be a danger of confusion when representing big and complex applications.

After representing a component-based application according to the Application level, the received information can therefore be organized using category sets. These sets are defined by selector operators on the trait classification, and can be supplemented by any user of the ENT meta-model if another point of view is needed. In [2] five category sets are presented, from which we introduce here only the first one (E-N-T) that gave name to the ENT meta-model.

Category sets say how to group and display traits. The E-N-T category has three groups. In the first group are elements that are contained in traits with dimension {role = provided}, this means those elements which the component **exports**. Required elements are similarly grouped as **needs** and elements that can be both provided and required are called **ties**.

E-N-T (Exports-Needs-Ties)
$f^E = \lambda C.(C.role = \{provided\})$
$f^N = \lambda C.(C.role = \{required\})$
$f^T = \lambda C.(C.role = \{provided, required\})$

Fig. 3. ENT category set.

For example, the attributes and facets of the CORBA component in Example 1 belong to the "E" category set, while the *s*tateNotify event belongs to the "N" set (because it signals the component requires an event sink to which it needs to be connected).

3.5 Extensions of the ENT meta-model

The original ENT meta-model [2] was concerned only with the representation of standalone components. We wanted the ENT meta-model to be able to visualize whole component based applications with dependencies between components and we further wanted to add support for hierarchical components. The extensions of ENT meta-model are presented in this section.

A new meta-model entity *Binding=(Element local, Element alien, direction* $\in$ {*provided, required*}*)* was created to represent bindings between components. Bindings are realized through concrete elements that are physically linked to each other, with additional information about the binding direction – provided means "from local to alien", required means opposite. Every component has its own list of bindings, this list contains all bindings involving the elements of the component. This means a Binding between elements of components **A** and **B** is in the lists of both these components. This method has the advantage of ensuring that the binding list is complete in any component's representation.

The ENT meta-model already contained lists of all elements that can be bound, as element sets contained in traits. This modification only allows to add information which elements are actually bound to other elements and does not create any new element.

The list of components which constitute a hierarchical component was added as an attribute to the parent component structure. This modification together with element bindings allows to represent hierarchical components. This kind of components can add its own elements and export/import only some of the elements that are contained by components it is built from. These inner components are restricted in that they can only bind with each other within the boundaries of their parent hierarchical component. In every other respect they are normal components.

4 ENT model in MOF

This section introduces the data structures which form a concrete implementation of the ENT meta-model, to be used in visualization of component-based applications. The explanations in this section use the description of the process of creating this design and implementation, as it provides a way to share experiences that other projects can draw from.

The MOF is a Domain Specific Language used to define meta-models. The core of MOF is shared with UML and its meta-models can be defined using UML class diagram. This means the ENT meta-model can be defined using MOF and the product of this definition will be UML class diagram which can later be processed. Entities defined in this section can be implemented in specific programming language and used as data layer in any other project.

The creation of a MOF model is most easily started from a formal definition of the corresponding domain abstractions. We have to keep in mind that there can be changes introduced by this MOF model because the formal description doesn't consider implementation limitations and details like references on objects.

We will present the ENT model thus created in three separate parts. The *Classification system* is modeled as a simple class `ENTClassification` with attributes corresponding to classification facets. These facets are modeled as enumerations, which is appropriate given their needs. The facet attributes were identified as mandatory, with "single" multiplicity except for Life-cycle and Role which have "multiple" multiplicity.

The *component model level* is represented by four elements. ComponentModel entity is the main one and is designed to keep all instances together on one place. ComponentTypes and TraitDefinitions will be accessed via references gained from arrays stored in

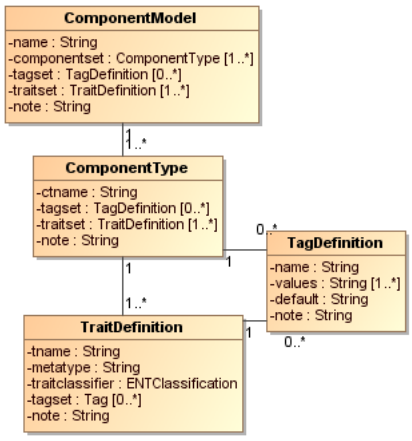

Fig. 4. Component model level in MOF.

ComponentModel. This level quite corresponds to its formal description. The "note" attribute was added to all these elements, to provide for descriptive information about the implemented component model in this data structure.

The *application level* is represented by five elements and it is the most changed part of the ENT meta-model compared to its formal description in [2]. The changes were due to the extensions described in Section 3.5. Component, Trait and Tag entities contain references to their descriptors at the component model level. Binding and Element entities are component model independent.

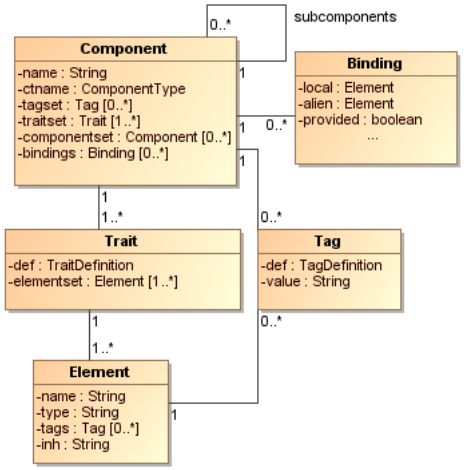

Fig. 5. Application level in MOF.

5 EMF implementation

The MOF model could be converted to a concrete implementation manually, but EMF (Eclipse Modeling Framework) was chosen instead. EMF is used for Model Driven Development and offers additional services such as generating the model classes from UML, and editors for the model in the form of an plugin for the Eclipse integrated development environment. This section describes the advantages brought by the use of EMF.

EMF was used to generate a Java implementation of the data layer from the MOF model of the ENT meta-model. This implementation is called ENTMM as an abbreviation for ENT meta-model. ENTMM consists of interfaces and classes corresponding directly to the model entities presented in the previous section. This implementation thus forms a run-time ENT representation of any given component model and its applications and will be used in every project that uses the ENT meta-model; for some such future projects see Section 7.

An editor of ENTMM data was created to provide GUI for component model definition described in Section 3.2; it is very similar to editor displayed in Figure 6. This editor is implemented as Eclipse plugin and can't be used without the Eclipse IDE.

EMF however offers more than just the advanced code generator and Eclipse plug-ins to support modeling. One of its features is advanced work with the XML format. Editors automatically save all model data in an XML file which can be easily accessed using EMF-generated resource classes. This brought us very usable form of automated storing and loading representations of various component models. This EMF ability goes both ways so in the future there is the possibility of using EMF buit-in features to save current application models in XML.

Eclipse Modeling Framework is able to set many features of the generated model in its editor and it is strongly recommended to use this opportunity instead of manual changes to generated code. EMF transforms the UML diagram into its internal format of "ECORE" file, where all information and settings related to the model are stored. Similarly there is a "GENMODEL" file which is used to store settings for the editors of the generated model and for the generating process itself.

The base model created automatically had several limitations mainly from practical point of view. Fortunately, EMF is able to set many features for the component model representation created in its editor. After an automated transformation of UML into the EMF basic ECORE format, a few additional changes had to be performed.

The most important one was to configure the ECORE editor so that entity references are used (instead of instances) for many model class attributes. This prevented the undesired effect that the same trait definition could not be shared by several component representations.

6 Creating component model representations

The ENTMM editor for Eclipse was used to create the representation of several component models, namely OSGi, EJB, CORBA, and SOFA (which have been already defined) and the CoSi [20] and MVE [11] models (their definitions were newly created). In this process, the plugins created by EMF and described in Section 5 were used.

By adding these plugins, a new file type appears in Eclipse while creating new file, named "ENTMM Model". To start defining the representation, the user has to choose this file type and select "Component Model" as an "Model Object" when asked. ENTMM editor like the one in Figure 6 will appear and by using its simple interface, the user is able to define a new component model easily.

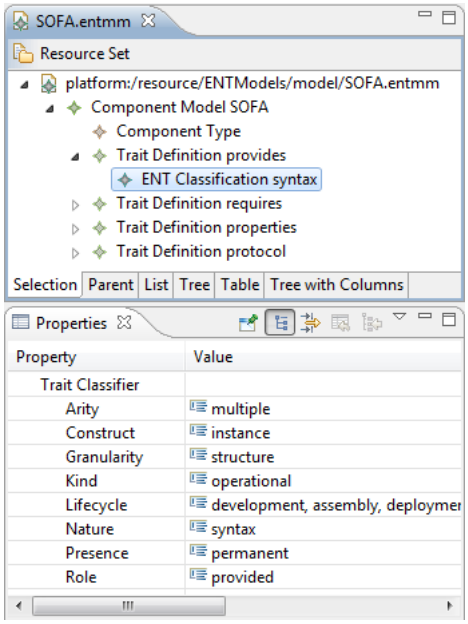

Fig. 6. ENTMM editor in Eclipse IDE.

6.1 Case study of SOFA

As a case study, the representation of the SOFA component model is presented below to show how simple it is to create a definition of a new component model. SOFA [19] is a research component model from Charles University, Prague. A SOFA component is interacting with other components only via designated provided and required interfaces. A component can be viewed as both a black-box and gray-box entity.

Definition of SOFA component model in ENT is provided in [2], and repeated below for understanding of this case study.

Definition of SOFA component model in ENT:

SOFA component framework defines one kind of components, with no component-level and element-level tags and with four traits.

provides - provided interfaces
metatype = *interface*,
classifier = ({*syntax*}, {*operational*}, {*provided*}, {*structure*}, {*instance*}, {*permanent*}, {*multiple*}, *Lifecycle*),

requires - required interfaces
metatype = *interface*,
classifier = ({*syntax*}, {*operational*}, {*required*}, {*structure*}, {*instance*}, {*permanent*}, {*multiple*}, *Lifecycle*),

properties - provided interfaces
metatype = *property*,
classifier = ({*syntax*}, {*data*}, {*provided*}, {*item*}, {*instance*}, {*permanent*}, {*multiple*}, {*development, assembly, runtime*}),

protocol - provided interfaces
metatype = *protocol*, and
classifier = ({*semantics*}, {*operational*}, {*provided, required*}, {*item*}, {*type*}, {*permanent*}, {*na*}, {*development, assembly, runtime*}).

The representation of the SOFA component model was created in ENTMM editor and the final view of this implementation can be seen in Figure 6. As mentioned in Section 5, EMF stores data in XML format. XML version of the SOFA component model representation is given in Example 2. This representation can be recreated by intuitive use of editor and filling data from the formal definition to the prepared data structure.

The generated XML structure does not follow all rules of good XML data but EMF does not support XML customization. This disadvantage is the only tax to pay for automated storing and loading of component models.

Example 2.

```
<?xml version="1.0" encoding="UTF-8"?>
<ENTMM:ComponentModel xmi:version="2.0"
```

```
xmlns:xmi="http://www.omg.org/XMI"
xmlns:ENTMM="http:///ENTMM.ecore"
name="SOFA">

<componentSet traitSet="//@traitSet.2
  //@traitSet.3 //@traitSet.0
  //@traitSet.1"/>

<traitSet name="provides"
    metatype="interface">
  <traitClassifier granularity=
    "structure" arity="multiple"
    construct="instance"
    presence="permanent">

    <role>provided</role>
    <lifecycle>development</lifecycle>
    <lifecycle>assembly</lifecycle>
    <lifecycle>deployment</lifecycle>
    <lifecycle>setup</lifecycle>
    <lifecycle>runtime</lifecycle>
  </traitClassifier>
</traitSet>
<traitSet name="requires"
    metatype="interface">
  <traitClassifier ...>
  </traitClassifier>
</traitSet>
<traitSet name="properties"
    metatype="property">
  <traitClassifier ...>
  </traitClassifier>
</traitSet>
<traitSet name="protocol"
    metatype="protocol">
  <traitClassifier ...>
  </traitClassifier>
</traitSet>
</ENTMM:ComponentModel>
```

7 Future work

Having created the ENTMM implementation, work is currently under way on the implementation of component application loaders for the OSGi and CoSi frameworks. These loaders should analyze component application based on the used component model and load information we are interested in. This information will be stored in the implemented ENT metamodel data structures. In future we will extend the supported component models to include EJB, SOFA, etc. based on actual needs.

We concurrently work on Component model visualizer based on ENT faceted views and of course on the ENT meta model EMF implementation. This application will support multiple ENT views and automated application loading using implemented component application loaders. This should give us a tool able to provide component application visualization for many component models and with multiple views based on ENT philosophy.

All these efforts should result in advanced visualizer of component based applications fulfilling these points:

1. Dynamic loading of any component based application no matter which component model is used.
2. Component displayed with additional information.
3. Support of different views, based on user needs.

The visualized component should look similar to component visualized in Figure 7. This kind of visualization will meet all requirements discussed in this paper.

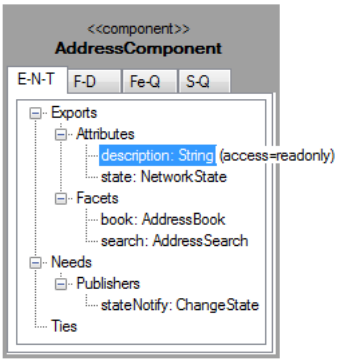

Fig. 7. AddressComponent visualized in ENT style.

8 Conclusion

This paper presented an extended ENT meta-model with the support for inter-component dependencies. The process of creating the MOF-based representation of this model was described to share as much experience as possible, including relevant class diagrams. Based on these diagrams a EMF-generated tool was presented which is used to create representations of concrete component models, with a discussion of its advantages and several interesting points we met in the process of generating the implementation Java code from the class diagrams.

Finally we presented component models that were implemented using the generated Eclipse IDE plugin. Brief description how to use this plugin is also provided and supplemented with a case study of the representation of the SOFA component model. Both the XML format and the graphic view of the final product were presented.

The key contribution of this paper is the description of a MOF-based data layer able to hold and interpret rich information about various component models and their concrete components. This layer can be used in many scenarios, including representing visually complex component-based applications.

This paper can also be used to learn experiences we gained in the process of the transformation from formal model definitions to this implementation. These experiences can be used as a whole, providing tutorial how to transform meta-model from paper to real life application, or separately when the reader is interested only in some parts like creating a MOF model or using the EMF tool.

References

1. C. Szyperski: *Component software: beyond object-oriented programming*, 2nd edition. Addison-Wesley Professional, 2002.
2. P. Brada: *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.
3. R. Prieto-Diaz, P. Freeman: *Classifying software for reusability.* IEEE Software **18** (1), 1987.
4. Object Management Group: *Meta Object Facility (MOF) Core Specification*, Version 2.0. OMG specification formal/06-01-01
5. Object Management Group: *UML Superstructure Specification*, Version 2.2. OMG specification formal/2009-02-02.
6. Object management Group: *CORBA Components*, Version 3.2. OMG Specification formal/02-12-06
7. D. Steinberg, F. Budinsky, M. Paternostro, and E. Merks: *Eclipse modelling framework*, Second Edition. Addison Wesley, 2009.
8. Sun Microsystems, Inc.: *Enterprise JavaBeans(TM) Specification*, Version 2.0. Sun Microsystems, Inc. 2001.
9. OSGi Alliance: *OSGi Servise Platform Core Specification.* OSGi Alliance, 2009.
10. F. Plăšil, D. Bălek, and R. Janeček: *SOFA/DCUP: architecture for component trading and dynamic updating.* Proceedings of ICCDS'98, Annapolis, Maryland, USA, 1998. IEEE CS Press.
11. M. Roušal and V. Skala: *Modular visualization environment - MVE.* Proceedings of International Conference ECI 2000, Herlany, Slovakia.
12. N. Medvidovic and R.N. Taylor: *A classification and comparison framework for software architecture description languages.* In: IEEE Transactions on Software Engineering **26** (1) 2000, 70–93.
13. I. Crknovic, M. Chaudron, S. Sentilles, and A. Vulgarakis: *A classification framework for component models.* Proceedings of the 7th Conference on Software Engineering and Practice in Sweden, 2007.
14. C. Escoffier and R.S. Hall: *Dynamically adaptable applications with iPOJO service components.* Proceedings of 6th International Symposium on Software Composition, Braga, Portugal, 2007.
15. I. Poernomo: *A type theoretic framework for formal metamodelling.* In: Architecting Systems with Trustworthy Components, Lecture Notes in Computer Science 3938/2006, Springer-Verlag 2006.
16. R. Monge, C. Alves, C. and A. Vallecillo: *A graphical representation of COTS-based software architectures.* Proceedings of IDEAS, April 2002.
17. Ch. Lüer, and D.S. Rosenblum: *UML component diagrams and software architecture – experiences from the WREN project.* 1st ICSE Workshop on Describing Software Architecture with UML, 2002.
18. F. Plášil, and S. Višnovský: *Behavior protocols for software components.* IEEE Transactions on Software Engineering, **28** (10), 2002.
19. T. Bures, P. Hnetynka and F. Plasil: *SOFA 2.0: Balancing advanced features in a hierarchical component model.* Proceedings of SERA 2006, IEEE CS, 2006.
20. P. Brada: *The CoSi component model: reviving the black-box nature of components.* Proceedings of the 11th International Symposium on Component Based Software Engineering, October 2008, Springer Verlag, LNCS 5282.