University of West Bohemia in Pilsen
Department of Computer Science and Engineering
Univerzitni 8
30614 Pilsen
Czech Republic

The ENT Meta-Model of Component Interface, version 2

Přemysl Brada

Technical Report No. DCSE/TR-2004-14 (draft)
September, 2004

Distribution: public

Technical Report No. DCSE/TR-2004-14 (draft)
September 2004

The ENT Meta-Model of Component Interface, version 2

Přemysl Brada

Abstract

Software modules and components have always played a key role in software engineering, primarily as key abstractions that embody the principle of information hiding. Modelling components is an increasingly important task, especially with the current interest in model-driven development and server-side component technologies.

This technical report presents an update to the ENT meta-model for structuring component interfaces, originally defined in an earlier report (TR-2002-10 "The ENT Model: A General Model for Software Interface Structuring") and author's PhD thesis [6]. While the core of the meta-model has not changed, three important improvements have been made which warrant the creation this version 2 of ENT.

Firstly, several clarifications of key concepts have been incorporated. This includes more precise terminology and enhancements to the classification system.

Secondly, the meta-model has been re-structured for ease of comprehension. It now starts with the definition of component which has been enriched with component-level tags, and is recursively decomposed into traits and elements. Former categories have been moved outside the structure and, hopefully more appropriately, called *views*.

Lastly, our recent work has resulted in the creation of the ENT model for Enterprise JavaBeans. It is included in this report in Appendix C, complementing SOFA and CORBA Component models. Also a XML representation of the ENT data has been added to this report.

This work originated with support from the Research Plan MSM235200005 funded by the Ministry of Education of the Czech Republic.

Copies of this report are available on
http://www.kiv.zcu.cz/publications/
or by surface mail on request sent to the following address:

University of West Bohemia in Pilsen
Department of Computer Science and Engineering
Univerzitni 8
30614 Pilsen
Czech Republic

Contents

1 Introduction

This paper presents an updated version of the ENT meta-model, originally defined in [5, 6]. It is concerned with the interface of software components [41] and marginally also modules [29], the key abstractions which support the separation of interface and implementation, driven by the principle of information hiding.

Component *models* [14] define, among others, the kinds of structures visible on the component interface[1] and their concrete syntax. Models are put into operation by component *frameworks*; examples are the CORBA Component Model [23], EJB [40], Koala [44] in industrial use, or SOFA [31] and Fractal [8] from the research community.

Component *meta-models* (the M3 level in the Meta Object Facility [24]) define the vocabulary and structures from which the capabilities of concrete models are derived. Some meta-models are created "a-priori" (for example the UML EDOC Profile [25], at least to a large extent), other ones by distilling the commonalities of existing component models ("derived meta-models") for analysis or technology conversion purposes. An example of the latter is Rastofer's metamodel [33].

The problem with current meta-models is that the structures and relations they define are mostly straightforward generalisations of the present state of the technology. Except for [37], they offer few forward-thinking ideas and provisions to handle future developments. In a longer run, this lack of abstraction hinders the development and adoption of advanced component features necessary for tackling the increasing complexity of software.

Worse yet, even some of the "penetrating" technological features currently in wide use (persistence, reliability, concurrency) are handled in an ad-hoc manner on a per-model basis, instead of being defined at the meta-level. This leads to a duplication of effort and problems in component interoperability.

New, enhanced meta-models are therefore needed to accommodate both the state of the technology and the upcoming developments (streaming media [9], mobility, emphasis on quality of service). The ENT[2] meta-model, defined in this paper, is able to fulfill these needs and additionally allows multi-faceted views and analyses of the component interface.

1.1 Who is Interested in Component Interface

The component interface declares the features through which the component and its environment interact. Thus it is principal to understanding component's us-

[1] The rules for the creation, composition and communication of individual components, also defined by the models, are not our primary interest.

[2] The name comes from the abbreviation of a key set of structures – Exports-Needs-Ties; see Section 2.3 below.

age. We define four classes of clients with distinct, and different, interests (or *viewpoints*) in such understanding.

buyer Ordinary users consult component interface specification to see what it does or how it differs from what they currently have.

application assembler Application assembler need to know how to incorporate the component into an component-based application – what types and operations it provides, which interfaces it depends on, what properties govern the usage of the component, etc. She is interested in the business interface of the component, as well as in the dependencies to other interfaces/components and lifecycle management of the component instances.

The tools used by assemblers need to be able to reliably compare component specifications (for the purpose of linking or interconnecting components), extract type information, etc.

deployer The deployer's role is to deploy an assembled application onto its target runtime environment (e.g. a container). She is therefore interested in component dependencies, declared configuration and run-time properties such as persistence or concurrency, as well as in security issues.

run-time client The run-time software clients execute/use the components of an deployed application. They are thus primarily concerned with their provided business interfaces, but also with component lookup, identity and lifecycle management.

We note that for all these reasons, the specification of component interface should at the same time be clear, precise, rich and human readable – which unfortunately is rarely fulfilled by current component models.

1.2 Goal and Structure of the Paper

This report presents an updated version of the ENT derived component meta-model [5, 6]. The key design goal for the meta-model is to facilitate the *understanding* of components in a way equally useful for both the human and the software-based clients. It addresses this challenge by defining its structures based on an analysis of several existing component models and frameworks, driven by the viewpoints described above. This approach also allows ENT to be an open meta-model which easily encompasses realizations of the "penetrating" and future technologies manifest on the interface level.

The need for update of the model has arisen for several reasons. Firstly, our recent work resulted in better understanding of component models and required a

more precise interface element classification system. The second important reason was the desire to re-structure the model for better readability; linked to this was the need to enhance the representation of the whole component by component-wide tags. There were also some implementation developments which we to include in the updated definition.

The core of the updated ENT model is described in Section 2 of this paper. The enhancements since the first version include a more precise and detailed classification system of interface elements, and a better structuring of the model itself. Section 3 describes two prototype applications which use the ENT meta-model: a CORBA Component Model editor, and a generic viewer which uses XML for ENT-representation of the component specifications. Next, we evaluate the model, compare it with some related work and mention the current open issues in Section 4. After the conclusion, the appendices provide ENT-based definitions of the SOFA, CORBA Components and Enterprise JavaBeans component models. Appendix C provides the specification of the XML form of ENT component representation.

2 The ENT Meta-Model of Component Interface

To create an open meta-model for components, an understanding of the current and foreseeable technological trends is substantial. We have therefore conducted an analysis [6] of key component meta-models, concrete models and the designs of important modular programming languages. In the approach and classification used, we have been inspired by the comparative study by Medvidovic and Taylor [16].

For our analysis, we have among others selected the following frameworks: SOFA [31] and Han's model [11] for their interesting properties, CORBA Component Model (CCM) [23] and Enterprise JavaBeans [40] for their industrial relevance, and Fractal [8] as a new research effort. In addition, we briefly surveyed the ArchJava language [2] with its alternative approach, package specifications in Ada [12] because of its acclaimed language design, and several older, research component models.

2.1 Commonalities in Current Component Models

Results of the analysis told us that each component model uses slightly different terms for the same or very similar concepts. We can therefore distill the commonalities into high-level abstractions, similarly to other existing meta-models.

In general, the specification of a given component's interface consists of *elements* which define its capabilities accessible from outside of the component. Most elements are uniquely named (within the scope of the component interface) and can be distinguished – apart from their language type – by various characteristics as observed by humans (users, developers, deployers of the component). A range of such characteristics which we obtained by the analysis is described in this section; a schematic view which motivates the approach is given in Figure 1.

The key distinguishing characteristics of elements is their *nature*. A component's interface can contain *syntactic* elements ("features"), or elements which declare its *semantic* or *non-functional* properties ("rules").

This nature of the element abstracts away not only from the particular interface specification languages with their syntax and type systems, but also from the individual characteristics of component models. Examples of syntactical features are an IDL interface of a CORBA or SOFA component, an event sink of a CORBA component, a log file created and written to by a web server module, etc. Typical instances of semantic rules are behaviour protocols in SOFA, state transition descriptions in Rapide [13], or the "illities" Han's model [11]. Non-functional rules are e.g. the quality of service indications [10].

Next, we have observed several other properties of the elements which we consider important from user's point of view. A fundamental distinction of the

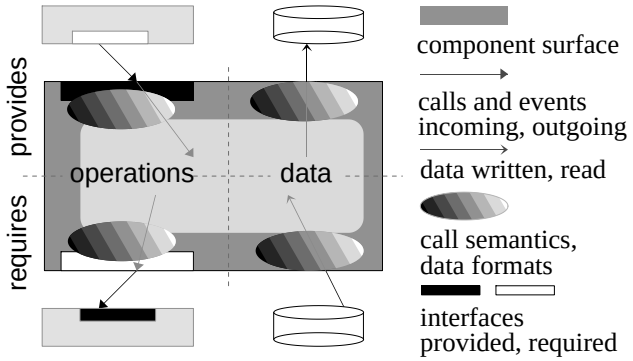

Figure 1: Interesting parts of software component interface

elements is by what we call the *kind* of the element. The *operational* features and rules describe or are used to invoke functionality (e.g. interfaces, events). The *data* features describe (sets of) data which the component exchanges with its environment (most often, these are called attributes, as in the CORBA component model). There can also be features and qualities which contain a mix of these two characteristics.

An orthogonal property is the element's *role* in component interactions. Each component *provides* elements which its clients can use to invoke its functionality and which thus represent the purpose of the component. On the other hand, the component may *require* the connection to or existence of some elements in its environment for correct linking or execution. Some kinds of elements (e.g. the behaviour protocol in SOFA components) describe the *ties* between these two parts of component interface, i.e. exhibit both provided and required roles. This distinction of element roles is explicit in the component-based systems and in many modular programming languages.

A user may be interested in the *granularity* of the element, since coarser elements tends to be more abstract and consequently better aligned with the granularity of the component as a whole. An element which is a single *item* is not structured in inter-component interactions, as is common with the data kind elements (e.g. CORBA attributes, JavaBeans properties). At at the operational level we prefer *structure*s as sets of items (e.g. whole interfaces). An extrapolation, not found in current models, is a *compound* of structures.

From the point of view of the specification language, it is sometimes important to distinguish the language *construct* of the element declaration. In most cases, the element will define an *instance* of a type; in rare circumstances (e.g. properties in UniCon [38]) also a *constant* value. Sometimes however, the element will contain just type information in the form of *type definition* or *type reference*. Then

its contents is not accessible via an identifier within the scope of the component declaration – as, for example, the `supports` interfaces of CORBA components.

In some systems, an element's necessity of *presence* can be designated. Ordinarily an element is *permanent* which means that it will always be present on the component interface at run-time; a *mandatory* element moreover has to be declared in the interface for the component to be valid. On the other hand, *optional* elements may be missing at run-time and still the component conforms to its specification. An example of a component model which uses this distinction is the Fractal framework [8].

Next, each feature may have different *arity* with respect to the bindings on that feature. We differentiate two cases, *single* arity for 1:1 bindings, and *multiple* for 1:N links. An example of using arity are CORBA Component Model's event publishers (which allow multiple sinks) and emitters (for one-to-one communication).

Lastly, we can differentiate features and rules according to their usage during or applicability to different stages in component *lifecycle*. Current models distinguish several such stages: *development* for correct compilation, static or dynamic linking, and packaging (when e.g. component assemblies are created from individual pieces), *assembly* (or design) for the integration stage of creating component interconnections in a visual tool and configuring the composed application, *deployment* which covers the phase of (re)configuring the application in the actual deployment environment, *setup* stage of application initialization and tear-down, and *run-time* stage which exercises interface elements during application execution for inter-component communication. Again, some elements may be relevant in several phases of the lifecycle – for example provided interfaces of a CORBA component are useful in compile-time, design-time as well as run-time stages.

2.2 Classification System of Element's Characteristics

We now formalize these findings in a classification system which uses the faceted classification approach [32]. The system has, at the present stage, eight facets called "dimensions" suitable for the classification of component interface features and quality attributes as described above.

The term space of each facet is represented as a set of identifiers that are defined as a set $Identifiers$ which contains strings described by the regular expression `[a-zA-Z_][a-zA-Z0-9_]*`.

We use a set $Id^{spec} = \{\epsilon, na, nk\} \subset Identifiers$ of special identifiers which denote an empty value, a not applicable case, and an unknown value, and any value, respectively. The na value is used in the cases when the given dimension is not applicable to the given feature or quality. The nk value (not known) is used when the class cannot be clearly determined.

Definition 2.1 (ENT **classification system)** *Let the term* interface element classification system *denote a faceted classification system suitable for classifying component interface elements, using a facet collection* $Dimensions = \{dim_1, dim_2, \ldots, dim_D\}$ *where* $dim_i = \{i | i \in Identifiers\} \cup Id^{spec}$. *Let the term classifier denote an ordered tuple* $(d_1, d_2, \ldots, d_D)$ *such that* $d_i \subseteq dim_i$.

The ENT classification system *is an interface element classification system which uses an ontology (based on the understanding of interface elements by human users and developers)* $Dimensions_{ENT} = \{Nature, Kind, Role, Granularity, Construct, Presence, Arity, Lifecycle\}$ *where*

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

The classification of an object is done via an ENT classifier. *This is an ordered tuple* $(nature, kind, role, granularity, construct, presence, arity, lifecycle) = (d_1, d_2, \ldots, d_D)$ *such that* $d_i \subseteq dim_i$, *and* $dim_i \in Dimensions_{ENT}$. *This classifier structure, which is a net result of the conducted analysis of component models and frameworks, is used as a key part of our meta-model described in the following section.*

While this classification system has been found sufficient for the description of several existing component models, it is independent of the number of dimensions and open to further development. On the other hand, there may be systems which can unambiguously distinguish interface elements using a subset of the core ENT classification. For example, the $\{Contents, Kind, Role\}$ facet collection would be sufficient for the current SOFA component model.

2.3 The Meta-Model: Component, Traits and Elements

This section provides a complete definition of the structures which form the ENT meta-model as such. Its overall structure is as follows. The meta-model describes a set of concrete *component model*s. The primary meta-object in a component model is a *component type* with the meaning corresponding to the classic Szyperski's definition [41, 6]. The component is composed of (and its type defined by) a set of characteristic *traits* which group individual interface *elements* sharing the same classification properties.

The rest of this section provides a formal definition of these structures, in a bottom-up fashion, followed by a notion of trait *categories* which provide a user-defined abstraction layer. At each layer, both the definition and representation of the given structure are described.

2.3.1 Component Model

A concrete component model is in the ENT meta-model defined by a set of its component types.

Definition 2.2 (Component model) *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.*

The component type definition is described next. We do not define component model representation, since the definition is itself a model's representation (a model is the meta-level for component representations described below).

2.3.2 Component

The top of the meta-model structure for a concrete component model is the component type, plus a corresponding representation of concrete components of this type. As mentioned above, the types of components in a component model are defined by describing the commonalities of their interface elements using the notion of component traits.

Definition 2.3 (Component type) *A* component type *is defined as a tuple* $C_{def} = (ctname, tagset, T_S)$ *where* $ctname \in Identifiers$ *is the name of the component type,* $tagset = \{(name_i, valset_i, default_i)\}, name_i \in Identifiers$, $valset_i \subseteq Identifiers$, *and* $default_i \in valset_i \cup \{\epsilon\}$ *is the definition of possible component-level tags, and the* $T_S = \{T_{i,def}\}$ *is the definition of the component type's trait set.*

The following consistency rules must hold for components in a component model M*:*

1. *Component types of one component model must be distinct, i.e.* $\forall C_i, C_j \in M.C_S, i \neq j : C_i \neq C_j \ \wedge \ C_i.ctname \neq C_j.ctname$.

Definition 2.4 (Component) *An ENT representation of a concrete* component *is a tuple* $c = (name, ctname, tags, ts)$ *where* $name$ *is the component's name*[3]*,* $ctname$ *is the name of a component's type,* $tags = \{(name_i, value_i)\}, value_i \in Identifiers$ *is the set of its tags, and* $ts = \{t_i\}$ *is the concrete trait set of the component with traits as defined below.*

The following consistency rules must hold for a concrete component representation to be valid:

1. *The type name of a concrete component must refer to one of the model's component type names, i.e.* $\forall c \, \exists C_{i,def} \in M.C_S : c.ctname = C_{i,def}.ctname$.

2. *Tag values in the component representation* c *must be taken from the value set in its type definition, i.e.* $\forall t \in c.tags \ \exists d \in C_{i,def}.tagset : t.name = d.name \wedge t.value \in d.valset$ *where* $c.ctname = C_{i,def}.ctname$.

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 other words,* E_c *completely represents of the component's interface in our model.*

The component type is the meta-level definition for a concrete component model. For instance, "session bean" is one component type in the Enterprise JavaBeans component model with its characteristic trait set (see Appendix A.3). The $tagset$ part of the component type defines – in a declarative form – the optional semantic or non-functional information attached to the whole component. An example of such concept are the persistence and transaction management tags defined for Enterprise JavaBean components. Each tag has a name, a set of possible values (which is an enumeration of identifiers), and a default value. The special default ϵ is used to denote the case when there is no default, i.e. the value of the tag must be specified explicitly.

Every value of each tag is expected to map to some language phrase(s) of the specification language(s) used by the concrete component model (e.g. the keyword `readonly` in CORBA component model, or the `session-type` XML element plus `implements javax.ejb.SessionBean` Java language phrase for Enterprise JavaBeans).

The component representation concerns a concrete component defined according to one type available in a component model. For instance, `AddrBookManager`

[3]Note that in many component models, several *instances* of a concrete component can be created, each with unique identity. We do not deal with instances at the ENT meta-model level.

```
component Parking
{
  provides ParkingAccess barriers;
  readonly attribute PlaceNumber capacity;
  attribute string description;
  provides ModifyState for_admin;
  readonly attribute PlaceNumber free;
  readonly attribute ParkingState state;
  publishes ChangeState state_notify;
};
```

Figure 2: Interface specification of an example CORBA component

is a concrete EJB session bean. In the representation, the $tags$ tuple contains the single concrete value of each tag. If the interface specification from which the representation is obtained does not contain a corresponding specification language phrase with the tag's content, the default value from the definition is used.

Note: At present, the ENT meta-model does not deal with component *applications* as sets of interconnected concrete components. This is reserved for future work.

2.3.3 Component's Characteristic Traits

As was said at the beginning of this chapter, we would like our model to handle the declarations of elements in the component interface specification in a manner natural to our human perception. For example, it is quite natural for us to think of all component's provided interfaces as a group, regardless of their concrete interface types and location in the specification source. The meta-type and classifier element parts help us in creating this abstraction, which we call the characteristic traits of the component.

Definition 2.5 (Trait definition) *Let C^T be a tuple $(ct_1, ct_2, ..., ct_D)$ called* trait classifier*, where $ct_i \subseteq dim_i, dim_i \in Dimensions_{ENT}$. A component* trait definition *is a tuple $T_{def} = (tname, metatype, C^T, tagset)$ where $tname \in Identifiers$ is the trait's name, $metatype \in Identifiers$ is the meta-type of the elements in this trait, and $tagset$ is the set of allowed tags of these elements.*

The following consistency rules must hold for a valid trait definition:

1. *Traits of one component type must be distinct, i.e. $\forall T_i, T_j \in C_{def}.T_S, i \neq j : T_i \neq T_j \ \wedge \ T_i.tname \neq T_j.tname$.*

Definition 2.6 (Trait) *An ENT representation of an interface* trait *(of a concrete*

component c) is a pair $t = (def, E)$ where def is a (reference to) trait definition and $E \subseteq E_c$ is a subset of component's interface elements.

The following consistency rules must hold for a valid trait representation:

1. *For any concrete component c in a given component model M, its trait set must conform to the model's definition, i.e. $\forall t \in c.ts \exists C^{def} \in M.C_S : t.def \in C^{def}.T_S$.*

2. *The trait set definition of a component type must cover all interface elements of any concrete component c without duplicates, i.e. $\forall t_i, t_j \in c.ts : t_i.E \cap t_j.E = \emptyset \wedge \cup_{t_i \in c.ts} t_i.E = E_c$.*

In the trait definition, the $metatype$ denotes the meta-type of the trait's elements (such as "interface" or "event"). In practice, it may be related to or derived from the name of the corresponding non-terminal symbol in the grammar of L. The C^T element is the ENT classifier which uniquely describes the classification properties of the trait's elements. The $tagset$ has the same definition and meaning as that of the component, described above, except that the concrete tag values are assigned to individual elements (not to the trait).

The information about the meta-type and classification is based on an a-priori human analysis or design of the concrete component model, its component types and the meaning of the phrases of its interface specification language(s). The purpose of such effort is to create a complete but minimal set of meta-types and classifier combinations which can reliably distinguish the desired characteristic traits (of specification elements) in the model's component types. Once this work is done, its results are built into the parsers/generators of the interface specification language(s) for the component model.

A trait (the representation) is then a named set of interface elements with the same meaning, given by the trait definition, which represents their human perception. Thus we can for example get traits of provided events, required interfaces, provided design-time qualities, etc., mirroring user's view of the component.

Traits group elements of a component even if in the source these may be written in various places (as shown in Figure 3 on the following page). This allows us later to analyse the interface specification by the meaning of its parts rather than by their place of occurence or language type. This approach is similar to *connection protocols* described in [1].

We should note that not all combinations of element classification dimension values need make sense in the given specification language[4]. This in practice greatly reduces the number of traits and thus the complexity of the model. For

[4]In such cases we can use partial classification like $(quality, na, nk, runtime)$, providing as much information as practical.

example, the SOFA system provides the component specifier with just four traits of elements, as shown in Appendix A.1.

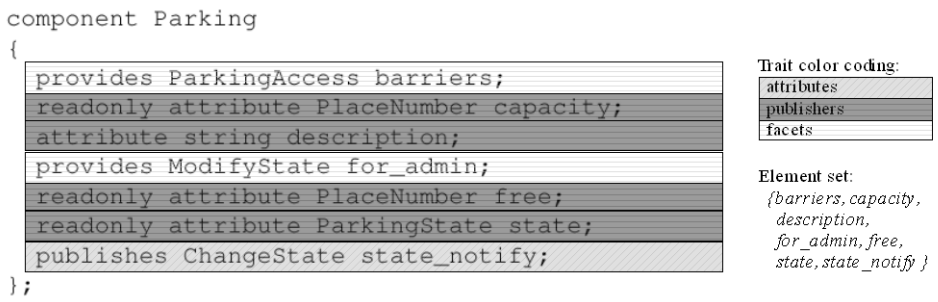

Figure 3: A CORBA component with ENT structures highlighted

2.3.4 Interface Elements

The smallest parts of the interface specification which are of interest to component users are called elements in the ENT meta-model. They are the ultimate subject of analysis and manipulation of the component interface specification.

Definition 2.7 (Interface element) *An* interface element e *of a concrete component* c *with interface specification written in language* L *is a tuple* $e = (name, type, tags, inh)$ *where* $name \in Identifiers$ *is the element's name,* $type \in L$ *is a language phrase denoting its type,* $tags = \{(name_i, value_i)\}, name_i \in Identifiers, value_i \in Identifiers$ *is the set of element's concrete tags, and* $inh = (i_1, \ldots, i_n); n \geq 0, i_m \in Identifiers$ *is the source of the element in* c*'s inheritance hierarchy.*

The following consistency rule must hold for an element in a trait:

1. *Tag values of trait's* t *elements must be taken from the value set in the trait definition, i.e.* $\forall e \in t.E \forall t \in e.tags\ \exists d \in t.def.tagset : t.name = d.name \wedge t.value \in d.valset$.

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 $tags$ part is a (possibly empty) ordered set of named tags which represent – in a declarative form – additional semantic or other non-functional information pertaining to the particular element (not to its type); similarly to the component-level tags. 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.

They are important if one needs to e.g. precisely compare two elements or regenerate a valid source code for the element. For example, in `final static int x = 5`, the `final static` keywords would be mapped to the $tags$ part of x. If the interface specification from which the element representation is obtained does not contain a corresponding language phrase with the tag's content, the default value from the element's trait definition is used.

The inh part denotes, as an ordered tuple of identifiers, the fully qualified name of the concrete component from which the element is inherited. For example, an element inherited from a component `::core::foo::Bar` will have $inh = (core, foo, Bar)$. This inheritance information is used to reconstruct the necessary inheritance clauses in the component specification. The ENT metamodel thus provides for component inheritance[5].

$name$ = capacity,
$type$ = PlaceNumber,
$tags$ = $\{(access, readonly)\}$,
inh = $\emptyset$

Figure 4: The $capacity$ element of the $Attributes$ trait in ENT representation

Completeness of the element means that it includes all the information about the feature contained in the component interface specification (with respect to both the language declarations) even if this information is not available in a single language phrase. For example, in SOFA CDL an interface variable belongs to either the `provides` or `requires` section but these keywords are not part of the interface variable declaration itself.

2.4 Categories: User-Defined Views on Components

Although traits are a useful grouping of specification declarations, for an architectural level view of a component their granularity is still too small. In high-level analyses of software we often come into situations where would like to handle for example "all provided features" as a single group. Such groups are called categories in our model.

[5]Although the author is not convinced of the usefulness of this concept.

Definition 2.8 (Category set definition, category) *The* definition of category *of interface traits is a tuple* $K^{def} = (kname, f^K)$ *in which* $kname \in Identifiers$ *is the name of the category and the* trait selection function $f^K : (d_1, \ldots, d_D) \rightarrow Boolean; d_i \in Dimensions_{ENT}$ *is a boolean function on ENT classifiers. A* category set definition *is a set of category definitions* $K_S = \{K_1^{def}, K_2^{def}, ..., K_n^{def}\}$ *such that* $\forall K_i, K_j \in K_S, i \neq j : K_i \neq K_j \wedge K_i.kname \neq K_j.kname$.

The category *of a concrete component's (c) interface traits is a tuple* $k^c = (def, ts)$ *where* def *is a (reference to) category definition and* $ts \subseteq c.ts$ *is a set of traits. A concrete component's* category set *is a set of categories* $k_S^c = \{k_i\}$.

The following consistency rules must hold for category set definitions and representations:

1. *The trait selection functions of any category set definition* K_S *must generate non-overlapping categories, i.e.* $\forall c \forall k_i, k_j \in k_S^c, i \neq j : k_i.ts \cap k_j.ts = \emptyset$.

2. *Any concrete component's category set* k^c *must conform to a category set definition, i.e.* $\forall k_S^c \; \exists K^{def} \; |k_S^c| = |K^{def}| \wedge \forall k_i \in k_S^c \exists K_i \in K^{def} : k_i.def = K_i$.

3. *A trait* t *belongs to a category if its trait selection function evaluates to true on the trait's classifier, i.e.* $f^K(t.C^T) = true$.

(Note that the category set definition is not required to cover all traits of a component model, i.e. each trait from the component trait set belongs to at most one category of a given category set: $\forall c : \cup_{k_i \in k_S^c} k_i.ts \subseteq E_c$.*)*

Categories group traits which are similar in some aspect(s) from human point of view. This is expressed in our model by sharing the values in some of their classification dimensions while disregarding other dimensions, as specified by the trait selection function f^K.

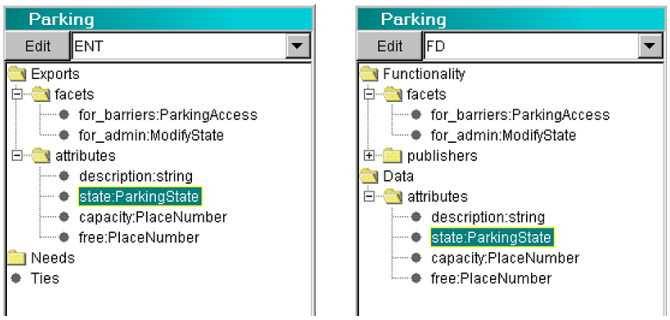

Figure 5: A CORBA component structured by different category sets

The other notable characteristic of categories is that they group elements of different meta-types. They therefore allow operations based on a high-level meaning of elements rather than on the syntax or the typing system of the language.

E-N-T (Exports-Needs-Ties)

$f^E = \lambda C.\ C.role = \{provided\})$
$f^N = \lambda C.\ C.role = \{required\})$
$f^T = \lambda C.\ C.role = \{provided, required\})$

F-D (Functionality-Data)

$f^F = \lambda C.\ (C.kind = \{operational\})$
$f^D = \lambda C.\ (C.kind = \{data\})$

Fe-Q (Features-Qualities)

$f^{Fe} = \lambda C.\ (C.nature = \{syntax\})$
$f^Q = \lambda C.\ (C.nature \subseteq \{semantics, nonfunctional\})$

S-Q (Server-side view)

$f^S = \lambda C.\ (C.nature = \{syntax\} \wedge C.role = \{provided\})$
$f^Q = \lambda C.\ (C.nature \subseteq \{semantics, nonfunctional\} \wedge$
$C.role = \{provided\})$

aPR (assembly-relevant Provided and Required)

$f^P = \lambda C.\ (C.role = \{provided\}) \wedge (assembly \in C.lifecycle)$
$f^R = \lambda C.\ (C.role = \{required\}) \wedge (assembly \in C.lifecycle)$

Figure 6: Example category sets

Since the category selection fuction is independent of any concrete component model (being defined on the classifiers, not on traits themselves), categories create a unifying abstraction layer on the component interface structure. Furthermore, these functions and thus the category sets are user-defined, which provides a way to express different sets of interests in the component interface. Thus we can define any number of different category sets independently of component models. The category sets, superimposed on ENT component interface representation by traits, can give us completely different views of the component. These two aspects make categories a good vehicle for the analysis of components on a fairly abstract level.

The category sets that can be useful in the ENT model applications are shown in Figure 5. The set of categories we find most useful is obtained by focusing on the $Role$ dimension. This way we get three categories, “Exports”, “Needs” and “Ties” – an “ENT” – which emphasises the different aspects which each part of

the interface has from the point of view of the component interconnections[6]. This view is crucial for both the developers and component framework implementations to ensure proper functionality of component applications [7].

[6]The T category explicitly sets apart the elements which express the bindings between the two parts of the component interface, such as SOFA behaviour protocols or the parametrised contracts [35].

3 Applications of the Model

The model of component interface structuring presented in the preceding section is applicable to a wide range of module- and component-based systems. It is also general enough to serve different purposes, suitable for the component developers (mainly in component understanding) as well as their tools (automated component comparisons). In this chapter we briefly present these aspects of the model, together with tools that are available as a result of ENT model development.

3.1 Applicability to Current Component Models

Although the ENT meta-model is designed to encompass future developments, it was developed by analysing current state of the technology. Its first version was mostly based on the CORBA Component Model (CCM [23]) and the SOFA framework [31], and these models can be defined in terms of the ENT meta-model cleanly (see Appendix A.2 and A.1). In a similar manner, other component or modularization systems which use an IDL-like language for the specification of component interface [19, 42] can utilize the ENT model.

Our subsequent work concerned the component models built on the Java platform, in particular the JavaBeans [39] and Enterprise JavaBeans [40] (EJB). The ENT-based model of the former is included in [6], for the latter see Appendix A.3 in this report.

The application of the ENT model to these two models however shows their key deficiency: the lack of explicit declaration of externally available component features. In other words, some interface elements are only declared within the source code of the beans and their methods, and thus are not accessible even by the Java reflection API. In JavaBeans, this is manifested by the design-time vs. run-time designation of elements by testing the `isDesignMode()` dynamic property[7] inside method bodies. In EJB this is mainly the case of messages accepted by message-driven beans.

This approach of Sun's component models makes it very difficult to reconstruct an ENT-based representation of existing components. Unless a sophisticated analysis of method body code (which is not always available) is used, it is impossible to correctly set the element's classification properties or even to find the elements themselves.

Based on our experiences, we expect the ENT model should be applicable also to widely used modular programming languages, such as to Delphi units, Ada packages or C language modules. However, the same problems as with the JavaBeans/EJB component models can be expected in these cases.

[7] From the `java.beans.DesignMode` interface

3.2 Design of New Component Models

Based on the experiences gained in developing the ENT meta-model, we believe it should be actually quite easy to design component models with rich sets of features and precise specifications.

The meta-model's key benefit in this respect is the guidance it would give to the model designers. By considering the values from the ENT classification system, the designer can concentrate on the desired high-level properties of the components, express them in appropriate traits, and then devise suitable syntactical structures to represent them in a specification language of their choice.

The following IDL-like code shows how we envisage a full-featured component specification, using the experiences gained in developing the ENT model described in this paper. The notable enhancements agains current IDL/ADL languages are:

- Explicit declaration of *data features* like files and streams, including means for datatype and/or format specification of their records.
- Use of annotations to describe semantic, classification and quality of service properties of individual elements as well as of the whole component.
- The ability to include versioning information in the component IDL specification.

```
dataformat LogFile [ascii]
{
  DateTime date;
  String<20> object;
  int result;
}

component ExampleCo [remote rev=3.1.1]
{
  provides:
    InterfaceA a1 [rev=4.1 synchronized arity=1];
    InterfaceB b1 [arity=any bind-after=a1]
     [* response-time:avg=1ms,max=30ms *];
    LogFile log [filename=SystemLogDir."ExampleCo.log"]
     [* growth-rate-avg=230 *];
  requires:
    InterfaceX x1 [rev-from=2.0 rev-to=3.3 synchronized];
```

```
    ConfigFile cfg
     [filename=SystemConfigDir."ExampleCo.cfg" read-write];
  properties:
    int MAX = 256 [design-time run-time];
    String<80> WindowTitle [design-time];
    String SystemLogDir = "/var/log/";
    String SystemConfigDir = "/etc/components/";
  state:
    int count;
    float[] data;
  invariant:
    count >= 0 and count < MAX;
  protocol:
    <init> { log.open ; cfg.read } ;
    ( ?a1.a { !x1.a ; log.write }
       || ?a1.b { !x1.a ; !x1.b ; log.write } )
    + ?b1.q ;
    <finishing> { log.close }
}
```

In our opinion the design of models based on the ENT meta-model, compared to the current state of the practice, can lead to better component interoperability, potential for code generation from the specifications, and more robust software because of a smaller number of hidden interdependencies.

3.3 Use for Humans: Flexible Visual Representation

Component developers or application assemblers, who use visual modeling of components would benefit if the component appearance could be affected according to a desired viewpoint. This means visual software presentation in user terms rather than (as common now) in language terms. The ultimate aim is to provide for easier and less error prone evaluation of component-based applications, thus facilitating tasks such as visual design, re-engineering and maintenance. The ENT structures together with trait categories provide the abstractions that support such presentation.

Following this idea, we developed a visual representation of software components that is inspired by the UML notation [21]. It tries to stick with the UML representation of class/interface structures — the component is shown as a box with its name in the top row, and constituent parts in separate boxes.

However, the contents of the component is structured according to a selected category set. The possibility to define category sets as desired provides a degree of flexibility in this structuring. Our model therefore allows us to parametrize the visual representation and additionally to group the constituent parts hierarchically, unlike the standard UML profile which prescribes a fixed flat structure of the class box.

Figure 7 gives an example of the resulting proposed visual representation of components, when applied to a CORBA component. On the left, the component is shown in the E, N, T category set while on the right in the Functionality-Data set. As can be seen, the use of category sets enables us to look at the component interface in completely different ways. We could similarly create a custom category set to e.g. show only the event-based part of CORBA components' interfaces, important when designing an application with asynchronous communication. The other interface elements, which would only distract from the primary design goal, would be completely elided from view.

We have developed two prototype tools which support this representation. One is a prototype application (shown on Figure 8) which uses XML representation of the ENT component model and component instances data, plus a set of XSLT stylesheets to convert and render this data. The second tool is a plug-in for the Borland JBuilder IDE for IDL3 (CORBA Components) editing, that enables to switch between source and visual views. The visual view uses the XML ENT model representation to parametrise component visualisation. At present, only separate components are displayed; the representation of inter-component links is planned for future work.

We propose that such view parametrization can have three applications:

1. In assembly (binding) of components into applications, e.g. in solving the tasks "now I want to see just the links between the provided and required ifaces" in CORBA components, or "let's see how events propagate" by showing just event sinks/sources with event names.

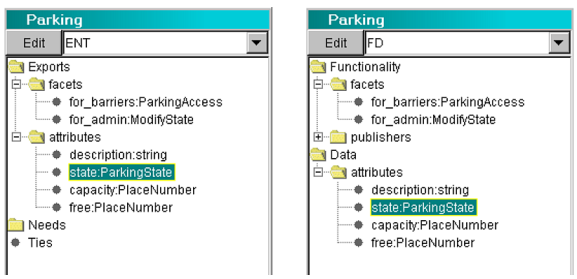

Figure 7: Two views of a CORBA component

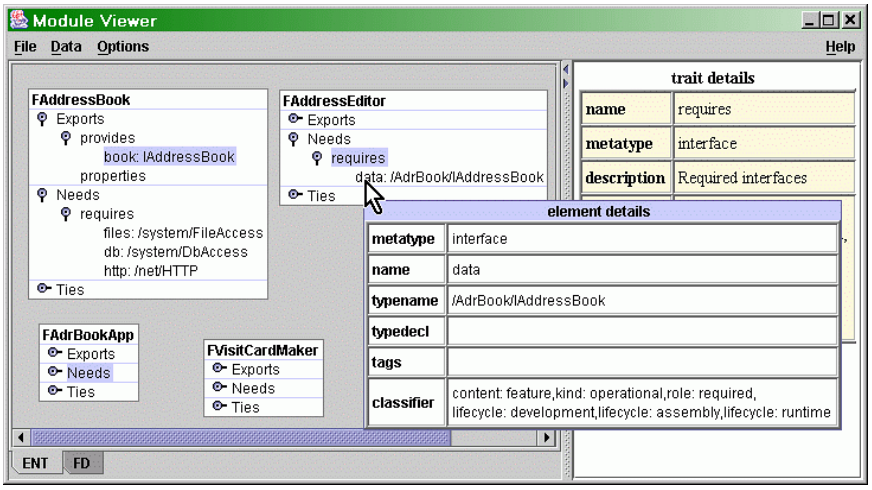

Figure 8: The ENT-VIS tool displaying a set of SOFA components

2. In search/evaluation, the model can provide a tree view of a single component in which the user can expand category, trait, and specification element contents to trace down a particular feature. Another such use is narrowing a search result set by augmenting the search methods (e.g. fulltext search in descriptions, signature matching, etc.) using the classifiers and other metadata associated with elements, traits and categories.

   For example, in seaching for the `animationRate` property of the `Juggler` JavaBean, the developer would use the Operational-Data categories and unfold, in sequence, the "Data" category and the "Properties" trait, to find the property in a candidate component. Alternatively, she could restrict a full-text search to just syntactical data elements.

3. In maintenance or servicing, the maintainer can test change propagation in "what-if" scenarios using the *Role* of elements (see Figure 9 on the next page) – change is OK if the proposed modification is an extension of the provided or a reduction of the required features.

Another option how to utilise the ENT concepts in visual representation would be different colourings of interface elements depending on their enclosing trait and/or category. This would allow an enrichment of current standard component notations like that used in UML 2 [26].

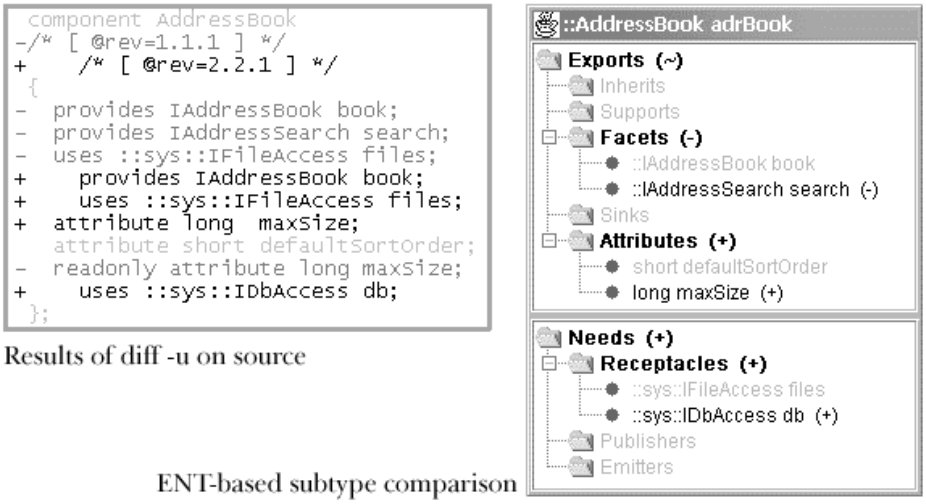

Figure 9: Text- vs. grammar-based component comparison

3.4 Use for Tools: XML-based Representation

In the area of tool-based component processing, the ENT model can provide support for automated component evaluation. The unique and interesting feature in this respect is the possibility to represent in the same format the interface of component from different models. This opens the opportunity for side-by-side comparison, high-level analyses, and even for translation of component interfaces between platforms.

For easier program manipulation with specifications structured according to the ENT meta-model, we have designed a simple XML data structure. It consists of two parts: definition of the concrete component model in the meta-model terms, and representation of a given component according to this definition.

The following example shows parts of the ENT XML of the SOFA model description and the E, N, T category set definition. For simplicity, the current data model uses simple conjunction as the category selection function; this is a sufficient approximation for the category sets defined above. The full specifications of the XML formats are given in Appendix B.1.

```
<model>
  <name>CCM</model>
<ctype>
  <name>component</name>
  <trait>
    <name>facets</name>
    <metatype>interface</metatype>
```

```
    <classifier>
      <dim name="nature">syntax</dim>
      <dim name="kind">operational</dim>
      <dim name="role">provided</dim>
      ...
    </classifier>
  </trait>
  ...
</ctype>
...
</model>

<catset><!-- a category set ->
  <name>ENT</name>
  <category>
    <name>Exports</name>
    <classifier>
      <!-- AND assumed between dim's -->
      <dim name="role">provided</dim>
    </classifier>
  </category>
  ...
</catset>
```

After a component model is defined, a concrete component's representation in the XML format looks as shown on the next example. The full specification of the format is in Appendix B.2.

```
<entrep>
  <component model="sofa">
    <provider/>
    <namespace>::network::http</namespace>
    <name>HTTPClient</name>
    <element>
       <name>connection</name>
       <type>HTTP</type>
       <tags/>
       <trait>facets</trait>
    </element>
    <element>
       <name>timeout</name>
       <type>long</type>
       <tags><tag name="access">readonly</tag></tags>
       <trait>attributes</trait>
    </element>
    ...
  </component>
  ...
</entrep>
```

The XML representation is used by the prototype tool for component visualisation, described in the previous section.

4 Evaluation and Discussion

The purpose of creating models is to abstract away details of the subject which are not interesting from the particular point of view. Therefore, care must be taken to balance simplicity and precision in the model definition. The subject of our work, modular and component-based software systems, exhibit a great degree of variation. Thus the goal of our work may be noble but is not easy to attain.

In this section, we would therefore like to discuss in more detail the ENT meta-model, its advantages and weaknesses. This opens the way for further work on applications and improvements of the model, as well as in related areas.

4.1 Advantages of the Model

The main objectives of the model are conceptual simplicity and close correspondence to human (primarily developer's) view on software components. The simplicity lies primarily in the use of a restricted set of classification facets and metadata items attached to interface elements, and in straighforward rules for their grouping into traits and categories. The model should thus be easy to comprehend and implement in code.

The application of the model to a given component framework or modular programming language results in a representation of components that is easy to visualise and comprehend. This stems from the selection of classification facets and from the natural hierarchy of elements, traits and categories. The model is thus a contribution to the area of program understanding.

Additionally, the model allows to manipulate the software specification (analyse, compare, transform) based on interesting semantic properties. While these properties are not always directly expressed in the syntax of the language, it is relatively easy to augment the given parser to extract them. Furthermore, the model hints the possible improvements in specification languages (see Section 3.2 on page 20).

Considering the primary role of meta-models, the ENT's novel approach to meta-modelling allows the designers to reason about the desired usage properties of components, rather than restricting them to the low-level problems of component wiring. In other words, the model directs towards what is useful and possible rather than merely about what is currently implemented.

The model was designed to be very general and independent of any particular technology or specification language. It is thus and applicable to many research and industrial platforms (see the concrete models redefined in ENT terms in Appendix B). As the definition of trait and category is not bound to a predefined classifier, the model allows to flexibly define ENT-based software representations for various purposes. The main practical application is the ability to reduce the

interface specification to an "interesting" subset (e.g. to the `provides` part of the component interface) depending on the concrete interests of the users.

Finally, the model is open for extensions. It was noted in Section 2 that the facet collection used in ENT classification is not fixed. Should the analysis of platforms, frameworks and languages not covered by our research reveal new classification dimensions, they can be added without directly affecting the model itself. Similarly, the comparison relation (subsumption) can be changed, e.g. using the approach to relaxed signature matching presented by Zaremski and Wing [45].

4.2 Disadvantages and Open Issues

The ENT model presented here has however several shortcomings that need the attention in future research. The primary problem as we see it is the need for manual classification of specification elements in the given language, when an ENT-based meta-model of a current component model is defined. This need arises because automated classification is in general a difficult problem [4, 45], in this case further complicated by the lack of expressiveness of some specification and programming languages. This opens room to different interpretations and thus imprecise classification of features and properties (e.g. along the *Lifecycle* dimension).

The second problem concerns the fact that elements are taken as atomic units without considering the details of their internal structure. For example, in the element `property: readonly int count;` the keyword `readonly` expresses mainly semantics of the property but this information is largely disregarded in the current model. Similar case are methods in Eiffel [18] with pre- and post-condition expressions.

This calls for a more accurate handling of the $tags$ part of the specification element. The desired effect would be achieved by attaching a classifier to the tags. This would make the model match reality better but at the expense of readability and simplicity. We therefore accept the simpler approach and consider declarations as monolithic, classified by its overall proximity to the classification facet terms. The internal structuring of declarations will be re-considered in future if such need arises.

In the present version, the ENT meta-model is concerned only with individual components, not with component applications as sets of interconnected components. It would be beneficial to include modeling formalism for these interconnections since their properties are equally interesting for application assemblers (this is e.g. the case of entity bean relationships specified in the Enterprise JavaBeans model).

Last but not least, the implementation of the ENT model for some languages requires non-trivial amount of work. In some cases it is necessary to redesign

the language grammar so that elements and traits are easier to separate. In any case the approach depends on the creation of suitable parsers which extract the relevant data from the specification source. These two tasks combined pose a challenge mainly in the case of syntactically rich programming languages like C/C++ or Java.

4.3 A Note on Current Component Specifications

There are however a few problems outside of the ENT meta-model, in specification languages themselves, which may hinder the full use of our approach to interface structuring. The most unfortunate one is the lack of expressiveness of current specification languages. For example, while the support for the `provides` role is common, only several research and a few industrial languages allow to specify required features [31, 23].

Similarly, the languages allow the specification of only a limited number of data feature types. The only common one are data properties, but in reality software components often depend on or create various data files and streams. No component framework in widespread use provides support for file or stream specifications that would capture this important aspect of their functionality.

The result is that the model presented in this paper can easily accomodate today's specifications but is not used to its full potential. Thus our reasoning about features and properties provides hints on what can (and should) be done in terms of improving component specifications. Section 3.2 on page 20 shows how we envision a component specification with some of these aspects implemented.

Taking this issue further, we have learned how important it is to have interface specification data for a component in an accessible form. It is difficult to re-create such information, be it in ENT-based or some other form, from the component models that do not use (full) separate interface specification, for example EJB. The consequence is that the users of such models have problems comprehending the component as a cohesive black-box entity which ultimately diminishes the desired effects of component-based programming.

5 Related Work

Meta-models. The work on the ENT meta-model was started by a comparative analysis of several component models. A similar, more detailed study of high-level similarities of various systems was done by Medvidovic and Taylor [16]. It concentrates around the principal purpose of both modules and components, for the basic understanding of the concepts. It uses a classification system which separates the features that can be specified on component interface into the syntactical structures ("interface"), semantics, and non-functional properties. We find this a useful classification scheme, and term it the *nature* of the component's features. On the other hand, they do not differentiate operational from data features, a distinction which we find important.

There exist several meta-models, both from the industry and research community, for software components. What we are interested in here are abstractions at the M3 (meta-metamodel) layer of the metadata ar-chitecture described in OMG's Meta Object Facility specification [24, Section 2.2].

The UML Profile for Enterprise Distributed Object Computing Specification (EDOC) [25, Chapter 3] defines the Component Collaboration Architecture, a UML profile for component-based modelling. It provides good modelling features and flexibility in terms of current industrial standards. EDOC is interesting in the distinction of three kinds of component interfaces which include mixed in-out interfaces ("protocol ports") and data-oriented interfaces ("flow ports").

The problems we find with EDOC are several ones. First, the term "component" is very loosely defined in the specification ("something that is composable" [25, Section 3.3.3]) which makes it difficult to interpret its meaning and relate meta-model's structures to concrete models. Next, the protocol ports allow to mix the specification of syntax (operations) and semantics (choreography) without distinction by identifiers, associations or language constructs. We believe that a clear separation of these concepts on the meta-model level is crucial for component modelling, implementation and analysis.

As a last point, the EDOC meta-model allows recursive composition of interfaces. This feature adds flexibility but we have doubts about the practical applicability of such abstraction, and feel that recursively defined ports are overly complex to understand.

Based on the (now withdrawn) Java Specification Request 26 "UML/EJB Mapping Specification" (`http://jcp.org/`), two industrial proposals for an EJB meta-model have been published. Both are presented as UML profiles for UML and are based on the UML 1.1 specification which makes them rather outdated nowadays (it is true though that the EJB 3.0 specification, now in preparation [**?**], may render any EJB meta-model obsolete including the one presented in this report in Appendix A.3). The profile by OMG [27] models a bean as a class which

has composition relationships to its various provided and required elements. This makes it possible to use a bean as a standard design element in static structure UML diagrams. However, neither the nature nor the role of the elements (in our terminology) are differentiated, and the specification does not provide mapping to the EJB implementation. These deficiencies make practical use of the meta-model for everyday development harder.

A proposal by Rational [**?**] is a draft specification presented by the JSR 26 workgroup. It is more detailed than the OMG one and specifies precisely implementation mappings (including the corresponding parts of deployment descriptor DTD) and constraints. However, it uses stereotyped package/subsystem for bean representation, probably motivated by its heterogenous implementation consisting of several artefacts. We believe this to be inappropriate since the meta-model is bound to implementation details instead of providing support for the more important design operations.

The research meta-model described by Seyler and Aniorte [37] is unique in the separation of the data and control flow in component description. The component interface is split into functional (control) and data (information) parts, and orthogonally into the standard required and provided roles. This meta-model provides features we think the commonly known models are lacking, and supports our position that data elements should be specified on component interface. On the other hand, its notion of information points is a very general concept which needs more concrete mapping on real objects – files, data streams, tables etc.

Rastofer [33] has developed a simple meta-model which is derived directly by extracting common basic features of ADLs and major industrial component frameworks. Although it includes connectors and constructs for describing composite components, we find its expressiveness (the range of features) rather limiting.

Component models. In the area of component models, there exists several widely known research and industrial systems. Some of them (C2 [42], SOFA [31], CORBA [23]) provide a reasonable component model which uses interface specification language (under various names – ADL, IDL, CDL) with syntactic distinction of provided and required parts of the interface. In some cases, a form of semantic properties specification is also available. There are also other frameworks (e.g. COM [19], EJB [40]) which do not match our general model too well but are interesting due to their widespread practical use. However, there seems to be a lack of explicit work on modeling module and component interfaces [28, 17].

The notion of module-based programming [29] first introduced the concept of information hiding and the separation between interface and implementation in software. This is now taken as one of the fundamental principles in software engineering and in the pure form is represented by the systems which use various IDL-like languages [22, 30, 36].

There are several languages representing the module-based programming pa-

radigm that are interesting from our point of view. First, the Ada programming language [12] provides very rich and precise means for specifying module and class interfaces. The Eiffel language [18] is interesting from the point of view that its class declarations can contain semantic properties in the form of pre- and post-condition plus class invariant. Also, the Eiffel compiler set includes a tool to generate a digest form of class declaration which is close to the specification languages mentioned above.

Among the languages that are more common in practical use, we might mention Java with its `javadoc` tool that can be used to generate a documentation of the interface to public elements in the class declaration.

Feature classification. The ENT model uses an (admittedly simple) faceted classification system first introduced to software by Prieto-Diaz [32]. However, the main inspiration comes from the implicit classification of features (keywords like `provides`, `imports`, etc.) found in some IDL languages [23, 31, 42]

Specification analysis. There are several works which deal with the analysis of software source code, be it interface specification or the implementation code. Medvidovic and Taylor [17] mention in this respect mainly enforcement of properties, simulation and code generation. Zaremski and Wing [45] extract method signatures from Standard ML code and then apply various forms of their matching in a library search approach.

CASE tools (e.g. Together [43], Rational Rose [34]) usually support so-called "round-trip engineering" where source code can be generated from an UML model, modified and then parsed to re-create the model.

Software visualization. Most component-based systems support some form of visual design mode in which components represented as elements without internal structure can be interconnected. However as Medvidovic [17] notes "support for other views is sparse".

There have been several proposals at a visual notation for software components (e.g. [20]), none of which has gained wider support. It is primarily due to the strength of UML as the de-facto standard modeling notation. With this role, the authors of UML2 [26] could have, in our opinion, paid a greater attention to the component meta-model built into the new version of the notation. In its present form, it cannot support even existing industrial component models (e.g. message-based communication used in both CCM and EJB). The visual representation of the components shown in the examples in Section 3 is inspired by the UML approach but goes further in the possibility to collapse individual levels (categories, traits, elements) and to parametrize the display by the category set used.

In the area of software comprehension, the work on visualizing change propagation (see e.g. [**?**]) touches a topic related to our work. The use of the ENT model can bring the benefit of separating dependencies between individual traits, allowing to focus on change propagation only in particular aspects.

6 Future Work

The ENT meta-model in its current version is, after several years of effort, quite rich and mature. Nevertheless there are several opportunities for future work. They are listed below in no particular order.

First, it would be beneficial to verify the model on a wider range of modular programming languages, component frameworks and specification languages. The main purpose of this expected work is to verify the structure of the specification element and the choice of classification dimensions. Among the candidate systems are .NET assemblies [**?**], research frameworks like the Wright language [3] and the ACME ADL [**?**], and also modular programming languages especially the Ada language packages [12].

The lack of meta-modeling support for component interconnections has been detected as one of the model's deficiencies. We expect that component link modeling can greatly benefit from the tailorable component visualization using category sets. This issue will therefore be one of our priorities in further research and implementation efforts on the ENT meta-model.

The possibility to visualise the component's ENT representation opens the way to further research its possible use in software visualisation, comprehension and modeling. While there exist various notations and approaches in this area that help software developers the ENT model can add the possibility to parametrize the visual representation by chosen category set. This may be interesting e.g. in relation to showing version differences or change propagation [**?**]. Obviously, we will be looking for ways how to use ENT modeling features to enhance the UML notation.

While we mention the Meta Object Facility in several places, a MOF representation of ENT metamodel is still missing. This gap will have to be closed in a near future, together with a closer study of the relation of the concepts behind ENT to those of MOF meta-models.

To support automated ENT data extraction, more research into the possibilities of grammar tagging is needed. The purpose of this work is to tag specification language grammar rules to indicate, whether the rule contributes to a particular element, trait, or their class. Such grammar tagging would enable us to automatically generate ENT parsers or at least their skeletons, thus eliminating the most mundane work on the model implementation.

There is also need for more work on the implementation of the model as presented in this paper. CORBA Component modeling has been partially implemented (see Section 3 on page 19) recently. At present, this work is concentrating on the Enterprise JavaBeans framework for which prototype tools are being designed.

7 Conclusion

In this report we have presented an updated version of the ENT component meta-model. It is a general model for natural structuring of the interface of software modules and components, motivated by the need to to enable analyses based on user understanding of the software. The model uses a rich faceted classification of interface elements derived from their various aspects as perceived by human developers and users. Based on this classification, the meta-model introduces a novel structuring of the interface into characteristic traits of like elements, and provides user-defined views.

The resulting structuring of interface into, among others, the exported and needed elements formalizes and extends the notion of software component as defined by Szyperski [41].

The key feature of the model is its extensibility and applicability to different component- and module-based systems. While being general enough to cover most current systems, its aim is to actively lead developers of future component models by concentrating on important properties of components rather than on their implementation details. In addition, the meta-model can easily incorporate ongoing developments by extending the classification system and/or enhancing the comparison methods used. As it is not tied to a concrete system it may serve, among other uses, as a unifying platform for software visualization.

Our subsequent work on the model will be mainly driven by the needs to further improve some aspects of the model, mainly in order to make it more precise in modeling detailed aspects of the component interface and interconnections. Also, several applications of the model will be explored together with a research into techniques facilitating further automation.

A ENT Definitions of Selected Component Models

A.1 SOFA Framework

The research SOFA component framework [31] defines one kind of components, with no component- and element-level tags and with four traits. The trait definitions are ordered by relative importance.

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 – data attributes
metatype = $property$,
classifier = $(\{syntax\}, \{data\}, \{provided\}, \{item\}, \{instance\}, \{permanent\}, \{multiple\}, \{development, assembly, runtime\})$, and

protocol – behavioural specification
metatype = $protocol$,
classifier = $(\{semantics\}, \{operational\}, \{provided, required\}, \{item\}, \{type\}, \{permanent\}, \{na\}, \{development, assembly, runtime\})$.

A.1.1 Example

```
frame FAddressBook {
  provides:
    IAddressBook book;
    IAddressSearch search;
  property short maxSize;
  requires:
    ::system::FileAccess files;
    ::OfficeApps::IPhoneBook phone;
  protocol:
    // this protocol is incomplete and inaccurate, but will do for
    // illustration purposes
    ?book.addPerson { !files.create? ; !files.write }
    ;
    ( ?book.clear { !files.write }
      | ?book.getPerson { !files.read }
```

```
        ...
    )*
}
```

The representation of the `FAddressBook` component in traits is as follows, ommitting empty traits and element classifiers.

provides $= \{(book, IAddressBook, \emptyset, \emptyset),$
$(search, IAddressSearch, \emptyset, \emptyset)\}$

requires $= \{(files, :: system :: FileAccess, \emptyset, \emptyset),$
$(phone, :: OfficeApps :: IPhoneBook, \emptyset, \emptyset)\}$

properties $= \{(maxSize, short, \emptyset, \emptyset)\}$

protocol $= \{(\epsilon, ?book.addPerson \ldots, \emptyset, \emptyset)\}$

A.2 CORBA Component Model

The CORBA Component Model specification [23] defines one meta-type of components, with a mixture of element metatypes and some element-level tags. Its trait definitions are ordered by relative importance.

supports – component-level interfaces
metatype = *interface*,
classifier = $(\{syntax\}, \{operational\}, \{provided\}, \{structure\}, \{type\}, \{permanent\}, \{na\}, Lifecycle)$

facets – provided interfaces
metatype = *interface*,
classifier = $(\{syntax\}, \{operational\}, \{provided\}, \{structure\}, \{instance\}, \{permanent\}, \{multiple\}, Lifecycle)$

receptacles – required interfaces
metatype = *interface*,
classifier = $(\{syntax\}, \{operational\}, \{required\}, \{structure\}, \{instance\}, \{permanent\}, \{single, multiple\}, Lifecycle)$
tags: arity = $\{single, multiple\}$

publishers – multicast events pushed out
metatype = *event*,
classifier = $(\{syntax\}, \{operational\}, \{required\}, \{item\}, \{instance\}, \{permanent\}, \{multiple\}, Lifecycle)$

emitters – events pushed out
metatype = *event*,
classifier = $(\{syntax\}, \{operational\}, \{required\}, \{item\}, \{instance\}, \{permanent\}, \{single\}, Lifecycle)$

sinks – accepted events
metatype = *event*,
classifier = $(\{syntax\}, \{operational\}, \{provided\}, \{item\}, \{instance\}, \{permanent\}, \{multiple\}, Lifecycle)$

attributes – data attributes
metatype = *attribute*,
classifier = $(\{syntax\}, \{data\}, \{provided\}, \{item\}, \{instance\}, \{permanent\}, \{na\}, \{development, assembly, deployment, runtime\})$
tags: access = $\{readonly, readwrite\}$

A.2.1 Example: The `Parking` Component

The source (from OpenCCM [15] examples):

```
component Parking
{
  // parking states.
  readonly attribute string description;
  readonly attribute ParkingState state;
  readonly attribute PlaceNumber capacity;
  readonly attribute PlaceNumber free;
  // parking facets.
  provides ParkingAccess for_barriers;
  provides ModifyState for_admin;
  // parking events ports.
  publishes ChangeState state_notify;
};
```

The representation of the `Parking` component in traits is as follows, ommitting empty traits and element classifiers.

facets $= \{(for\_barriers, ParkingAccess, \emptyset, \emptyset),$
$(for\_admin, ModifyState, \emptyset, \emptyset)\}$

publishers $= \{(state\_notify, ChangeState, \emptyset, \emptyset)\}$

attributes $= \{(description, string, \{(access, readonly)\}, \emptyset),$
$(state, ParkingState, \{(access, readonly)\}, \emptyset),$
$(capacity, PlaceNumber, \{(access, readonly)\}, \emptyset),$
$(free, PlaceNumber, \{(access, readonly)\}, \emptyset)\}$

A.3 Enterprise JavaBeans: the ENT meta-model

This appendix contains the specification of the Enterprise JavaBeans component model (version 2.1) [40] in terms of the ENT meta-model. It consists of the tag set for the EJB components and the definition of their trait set.

Since the EJB component model is hard to model to a full detail, included in the model is primarily type-related data. See section A.3.4 below for what has been omitted and what compromises had to be made.

Notes on conventions used: Metatypes used by traits are briefly described in section A.3.4. Tags are specified by name and the set of permitted values. The "Source" clauses point to the parts of EJB 2.1 component source where the given trait or tag is primarily defined. In other words, it is a hint (not a complete specification) on the mapping between the model and the implementation. The word "specification" denotes the Enterprise JavaBeans 2.1 specification [40].

A.3.1 Component Types

The Enterprise JavaBeans meta-model defines three component types: SessionBean, EntityBean, and MessageDrivenBean. For brevity of their definition we declare the following two sets of tag and trait definitions common to all of them:

$tagset_{common}$ = { security_id }

$traitset_{common}$ = $T_{provided} \cup T_{required}$ where $T_{provided}$ = { business_interfaces, home_interfaces, security_roles } and $T_{required}$ = { business_references, msg_destination_references, web_service_references, home_references, env_entries, resource_managers, resource_env_references, timer_service }.

SessionBean

tags: $tagset_{common} \cup$ { state, transaction }
trait set definition: $traitset_{common} \cup$ { web_service_endpoint } (a provided trait)

EntityBean

tags: $tagset_{common} \cup$ { persistence, reentrancy, schema }
trait set definition: $traitset_{common} \cup$ { attributes } (a provided trait)

MessageDrivenBean

tags: $tagset_{common} \cup$ { transaction, msg_type }
trait set definition: $traitset_{common} \cup$ { msg_consumed, msg_activation } (provided traits)

A.3.2 Tags on the Component Level

The EJB specification defines several component-level tags which mostly describe non-functional properties. The tags are listed in alphabetical order.

msg_type = *Identifiers*, default `javax.jms.MessageListener`
Java interface for message reception callback methods.
Source: deployment descriptor (the `messaging-type` element). Reference: specification section 23.5 (XSD definition of `message-driven--beanType`).

persistence = $\{container, bean\}$, default ϵ
Persistent state manager (for entity beans).
Source: deployment descriptor (the `persistence-type` element).

reentrancy = $\{reentrant, non\_reentrant\}$, default $reentrant$
Reentrancy mode.
Source: deployment descriptor (the `reentrant` element). Reference for the default value: specification section 10.5.12.

schema = *Identifiers*, default ϵ
Abstract schema name.
Applies only when $persistence = container$. Source: deployment descriptor (the `abstract-schema-name` element).

security_id = $\{use\_caller, run\_as\}$, default use_caller
Which identity to use for authentication.
Source: deployment descriptor (the `security-identity` element). Reference for the default: specification section 21.3.4.

state = $\{stateful, stateless\}$, default ϵ
State handling mode.
Source: deployment descriptor (the `session-type` element).

transaction = $\{container, bean\}$, default ϵ
The transaction manager to use.
Source: deployment descriptor (the `transaction-type` element).

A.3.3 Trait Definitions

Trait definitions are ordered alphabetically by trait name.

attributes – persistent fields of entity beans.
metatype = $attribute$

classifier = ($\{syntax\}$, $\{data\}$, $\{provided\}$, $\{item\}$, $\{instance\}$, $\{permanent\}$, $\{multiple\}$, $Lifecycle$)
tags: status = $\{normal, primary\_key\}$

Notes:

1. Source: deployment descriptor for container-managed entity beans (the `cmp-field` and `primkey-field` elements), Java source (!) for bean-managed persistence.

business_interfaces – bean's primary functionality.
metatype = $interface$
classifier = ($\{syntax\}$, $\{operational\}$, $\{provided\}$, $\{structure\}$, $\{type\}$, $\{mandatory\}$, $\{multiple\}$, $\{development, assembly, deployment, runtime\}$)
tags: locality = $\{local, remote\}$

Notes:

1. Source: deployment descriptor (the `remote` and `local` elements); Java source

business_references – depended-upon functionality.
metatype = $interface$
classifier = ($\{syntax\}$, $\{operational\}$, $\{required\}$, $\{structure\}$, $\{instance\}$, $\{permanent\}$, $\{single\}$, $Lifecycle$)
tags: locality = $\{local, remote\}$

Notes:

1. Source: deployment descriptor (parts of the `ejb-ref` and `ejb-local-ref` elements; see also the *home_references* trait). Contrary to the EJB deployment descriptor we separate these traits by the use of the elements rather than by the (in our opinion far less important) locality of the referenced beans.
2. $Arity$ is $single$ because only one server will satisfy the dependency link at a time.
3. This trait covers the whole lifecycle; the dependencies have to be set-up by the bean itself on its startup.
4. The $Construct$ dimension is classified as $instance$ since the references are named in the deployment descriptor. This is not entirely accurate (the name denotes the environment entry of the reference, not

the name of interface instance) but this way we emphasize the presence of the name which we find important.

env_entries – environment entries the bean needs.
metatype = $structure_{env\_entry\_type}$
classifier = $(\{syntax\}, \{data\}, \{required\}, \{item\}, \{instance\}, \{permanent\}, \{single\}, Lifecycle)$
tags: $\emptyset$

Notes:

1. Source: deployment descriptor (the `env-entry` element)
2. The environment entries concrete type within the structure is of Java's primitive type wrappers, i.e. `java.lang.Integer` and similar.
3. The arity is based on the specification: "An environment entry is scoped to the enterprise bean whose declaration contains the `env-entry` element. This means that the environment entry is inaccessible from other enterprise beans at runtime (...)"

home_interfaces – lifecycle management.
metatype = $interface$
classifier = $(\{syntax\}, \{operational\}, \{provided\}, \{structure\}, \{type\}, \{mandatory\}, \{multiple\}, \{development, assembly, deployment, setup\})$
tags: locality = $\{local, remote\}$

Notes:

1. Source: deployment descriptor (the `home` and `local-home` elements); Java source.
2. Although home interfaces can theoretically be used at run-time, their primary role (from the providing bean's viewpoint) is at the set-up time (cf. the key `create()`, `findBy...()` and `remove()` methods.)

home_references – access to depended-upon beans.
metatype = $interface$
classifier = $(\{syntax\}, \{operational\}, \{required\}, \{structure\}, \{instance\}, \{permanent\}, \{single\}, \{development, assembly, deployment, setup)$
tags: locality = $\{local, remote\}$, bean = $Identifiers$

Notes:

1. Source: deployment descriptor (the `ejb-ref` and `ejb-local-ref` elements).
2. See notes on the *business_references* trait.
3. The tags capture the added information about which bean should be used to satisfy this dependency ($bean$ corresponds to the `ejb-ref-name` element). Strictly speaking, this information is not interesting for the developer, and will be added during the assembly or deployment stages. The `ejb-ref-type` element is a pure duplication of information and as superfluous is therefore omitted in this model. The `ejb-link` element is outside of scope of an individual bean and cannot be captured in this model.

msg_activation – communication activation properties for message-driven beans.
metatype = map,
classifier = $(\{syntax\}, \{data\}, \{provided\}, \{item\}, \{instance\}, \{permanent\}, \{multiple\}, \{development, deployment\})$
tags: $\emptyset$

Notes:

1. Source: deployment descriptor (the `activation-config-property` element). Reference: specification section 15.4.9.
2. The activation configuration properties listed specifically in the specification (message acknowledgement, selectors and destination) pertain only to JMS-based bean communication and are therefore not part of the EJB component model – rather, they are part of its particular implementation.

msg_consumed – names of message classes accepted by the bean.
metatype = $class$,
classifier = $(\{syntax\}, \{operational\}, \{provided\}, \{item\}, \{instance\}, \{permanent\}, \{multiple\}, \{development, assembly, runtime\})$
tags: $\emptyset$

Notes:

1. Source: Java source (the `onMessage()` method implementation)
2. This trait is included in the model for fundamental reasons: the set of messaged accepted by the bean is key to understanding its functionality. However, from practical point of view it will be very difficult to extract this information from existing beans due to the source of

the trait elements. Therefore, we suggest that a single element with `javax.jms.Message` interface name be provided as default trait contents in cases where the source is unavailable or impossible to parse correctly.

3. Arity is *multiple* because in general the message may originate from a multicast. Lifecycle includes *assembly* because the knowledge of messages interchanged between beans is vital for application assembler; although in current reality there is no provision for declaring the messages emitted by other beans in EJB.

msg_destination_references – destinations for message-based communication.
metatype = $structure_{msg\_destination\_ref\_type}$,
classifier = $(\{syntax\}, \{data\}, \{required\}, \{item\}, \{instance\}, \{permanent\}, \{single\}, Lifecycle)$
tags: $\emptyset$

Notes:

1. Source: deployment descriptor (the `message-destination-ref` element). Reference: specification section 20.7.
2. For explanation of arity, see *env_entries* trait notes.
3. The type structure for the metatype is given by the XSD definition of `message-destination-refType`.

resource_env_references – objects needed by resource factories.
metatype = map
classifier = $(\{syntax\}, \{data\}, \{required\}, \{item\}, \{instance\}, \{permanent\}, \{single\}, \{development, deployment, setup, runtime\})$
tags: $\emptyset$

Notes:

1. Source: deployment descriptor (the `<resource-env-ref>` parts)
2. The $kind$ dimension is $data$ because these elements are objects used by or associated with the resource managers as "parameters".

resource_managers – resource factories needed.
metatype = map
classifier = $(\{syntax\}, \{operational\}, \{required\}, \{structure\}, \{instance\}, \{permanent\}, \{single\}, \{development, deployment, setup, runtime\})$
tags: sharing = $\{shareable, unshareable\}$, authentication = $\{application, container\}$.
Notes:

1. Source: deployment descriptor (the `<resource-ref>` parts)
2. The arity is based on a similar clause as in the *env_entries* trait. The *sharing* tag – as far as can be deduced from the specification (section 20.5.1.1) – applies to the objects acquired from the given resource manager, not to the manager itself.

security_roles – security roles associated with business interfaces.
metatype = $identifier$,
classifier = $(\{nonfunctional\}, \{operational\}, \{provided\}, \{item\}, \{constant\}, \{permanent\}, \{na\}, \{assembly, deployment\})$
tags: $\emptyset$

Notes:

1. Source: deployment descriptor (the `security-role-ref` element). Reference: specification chapter 21.
2. The role is based on the fact that security roles are associated with business interface methods. The lifecycle values are based on the specification which says that the values are used only by the application assembler and deployer; at runtime, the EJB container (not the bean) takes care of calling only methods corresponding to current security role.

timer_service – reaction to timed events.
metatype = $interface$
classifier = $(\{syntax\}, \{operational\}, \{required\}, \{structure\}, \{type\}, \{permanent\}, \{single\}, \{development, setup, runtime\})$
tags: $\emptyset$

Notes:

1. Source: Java source (bean class implements the `javax.ejb.TimedObject` interface). Reference: chapter 22 of the specification. This trait is exceptional in that it defines a dependency that is satisfied by the EJB container, rather than by some other bean.
2. The callback `ejbTimeout()` method which the bean as a result implements (as if provides) is just a technical means to realize the functionality of the dependency; therefore the $required$ role.
3. The arity is derived from the fact that only one timer service is available to a bean; see the `javax.ejb.EJBContext` interface.

web_service_endpoint – access to functionality via web services.

metatype = $interface_{sei}$
classifier = ($\{syntax\}$, $\{operational\}$, $\{provided\}$, $\{structure\}$, $\{type\}$, $\{permanent\}$, $\{multiple\}$, $\{development, assembly, deployment, runtime\}$)
tags: $\emptyset$

Notes:

1. Applies only to beans with tag value $state = stateless$.
2. Source: deployment descriptor (the `service-endpoint` element); Java source
3. The metatype is "service endpoint interface" as per the Web Services for J2EE specification [**?**].

web_service_references – depended-upon web services.

metatype = $structure_{service\_ref\_group}$
classifier = ($\{syntax\}$, $\{operational\}$, $\{required\}$, $\{structure\}$, $\{instance\}$, $\{permanent\}$, $\{multiple\}$, $\{development, deployment, setup, runtime\}$)
tags: $\emptyset$

Notes:

1. Source: deployment descriptor (the `service-ref` element).
2. The arity of this trait needs to be confirmed.

A.3.4 Notes on the EJB Model

The meta-types used in the definitions have the following meaning. The $interface$ and $attribute$ correspond to Java interface and class types. The $structure_X$ type denotes a XML element whose structure is given by the DTD or XSD specification of the X; $relation$ is just a special case of structure. Finally, map is to be interpreted as "name-value pair" where the interpretation of the value depends on the name of the property (e.g. for the trait *msg_activation* below, the JMS-based beans use values which are strings, enumerates and class names).

Several EJB features are omitted from this model, mostly because they have no meta-level counterparts in the ENT meta-model. While this prevents the resulting model to be used in round-trip transformations (between the model and code), at present there is no clean solution to this deficiency. Features of EJB 2.1 not modeled are:

- Entity bean relations (the `<relationships>` section of the deployment descriptor for CMP and the code implementing them for BMP beans) since

these describe the links between components and are thus not a part of their interfaces.

- Finder and select queries for CMP beans – queries are implementation rather than interface elements.
- The implicit dependency on bean context (`javax.ejb.EJBContext`) since this is always accessible – being mandatorily provided by the container – and its use is not manifest in the component interface (contrary to the timer service).
- Sub-element aspects, in particular transaction and security properties of business interface methods (see the `assembly-descriptor` part of deployment descriptor). The ENT meta-model does not currently capture sub-element properties.
- The deployment descriptor contains an optional element `client-ejb-jar` with references to `.jar`s with helper classes used by clients to compile/link with the beans; however, since it is outside the scope of individual components, this provided information cannot be modelled.
- Features and patterns of preceding versions of the EJB specifications (2.0 and older) were ignored in the model development, i.e. things that can be done both old- and new-way are modelled only as the new-way.

Finally, we should warn that this model is based on detailed study of the EJB 2.1 specification, 2 books, and experiments with core EJB features. Message-driven beans and their aspects, environment and resources, and web services have yet to be tried in practice to validate (possibly update) the model.

A.3.5 The EJB Component Model in Light of ENT

Several decisions have to be made to match EJB features with the ENT modelling viewpoint. The one which we would like to mention here is how to model dependencies on other beans (the `ejb-ref` part of the deployment descriptor). The options were (1) as per EJB 2.1 deployment descriptor specification, to model this dependency as an required bean element with home and business interfaces as tags; (2) is sync with the provided side of the interface as three traits, that is the bean, the home and the business interface.

What the dependency really means is the client will want to access the business interface of the referenced bean, and from this point of view the bean itself and the home interface are just supporting implementation details. Furthermore, if we want to use this dependency in substitutability checks, we need to work

with business interface, not the bean. Therefore we model the dependency by the *business_references* and *home_references* traits.

A second decision worth mentioning is the specification of aspects pertaining to specific kinds of beans only (for example, only stateless session beans can have web service endpoint defined). Since there is currently no provision for such rules in the ENT meta-model, the solution is to provide their written specification in the form of the "Applies only to..." clauses.

Our study of EJB 2.1 has shown that it has several weaknesses from the black-box component point of view. Firstly, it has several traits which are not declared in the externally available deployment descriptor – the *msg_consumed* and *timer_service* traits in full, and *attributes* in case of bean-managed persistence. The bean's source code has to be parsed to obtain the corresponding elements of existing beans; this makes it impossible to distill full ENT-based model in the standard case of deployment form of the bean (the `.jar` file). Complete information about these elements should be included in the deployment descriptor in order to make EJB components fully black-box.

Secondly, several pieces of information are duplicated in the source code and deployment descriptor, sometimes unnecessarily. A prime example is the specification of business and home interfaces; the duplication is necessary for the Java implementation to work. Additionally however, the business and home references needlessly force the application assembler and deployer to specify information which is already available elsewhere. The `home` and `remote` sub-elements of `ejb-ref` can be deduced from the required bean's deployment descriptor. The specification of its meta-type (the `ejb-ref-type` element) is in our opinion completely irrelevant.

A.3.6 Example: A Sample SessionBean

The following is an excerpt from the deployment descriptor of a simple session bean component, which is the primary source of its interface specification. The declarations of types listed need to be obtained from the component's `.jar` file by introspection or from its source code by syntactic analysis.

```
<enterprise-beans>
  <session>

     <ejb-name>AddrBookListing</ejb-name>
     <home>hello.beans.AddrBookListingRemoteHome</home>
     <remote>hello.beans.AddrBookListingRemote</remote>
     <ejb-class>hello.beans.AddrBookListingBean</ejb-class>

     <session-type>Stateless</session-type>
     <transaction-type>Container</transaction-type>
```

```
    <ejb-ref>
     <ejb-ref-name>ejb/AddressBook</ejb-ref-name>
     <ejb-ref-type>Entity</ejb-ref-type>
     <home>hello.beans.AddrBookRemoteHome</home>
     <remote>hello.beans.AddrBookRemote</remote>
     <ejb-link>AddrBook.jar#AddrBook</ejb-link>
    </ejb-ref>

</session>
...
```

The representation of the `AddrBookListing` component in traits is as follows.

component-level tags: state = $stateless$, transaction = $container$

business_interfaces $= \{(\epsilon, hello.beans.AddrBookListingRemote,$
$\{(locality, remote)\}, \emptyset)\}$

home_interfaces $= \{(\epsilon, hello.beans.AddrBookListingRemoteHome,$
$\{(locality, remote)\}, \emptyset)\}$

business_references $= \{(ejb/AddressBook, hello.beans.AddrBookRemote,$
$\{(locality, remote)\}, \emptyset)\}$

home_references $= \{(ejb/AddressBook, hello.beans.AddrBookRemoteHome,$
$\{(locality, remote)\}, \emptyset)\}$

B XML Representation

This appendix contains full document type definitions (DTDs) for the XML representation of ENT structures. Examples of their use are given in Section 3.4 on page 24 “Use for Tools: XML-based Representation”.

B.1 Component Model and Category Set Definitions

The following DTD defines the XML data structure for a concrete component model definition in ENT terms. Only one component model can be defined in one data file. The default extension of component model definition data files shall be “.emd” for “ENT model definition”.

```
<?xml version="1.0" encoding="UTF-8" ?>
<!--
  ENT component model definitions (EMD) DTD
  -->

<!-- component model -->
<!ELEMENT model (name, description?, ctype+) >
<!-- component meta-type -->
<!ELEMENT ctype (name, description?, trait*) >
<!-- trait definition -->
<!ELEMENT trait (name,description?,metatype,classifier) >
<!-- trait classifier -->
<!ELEMENT classifier (dim*) >

<!-- trait's elements metatype -->
<!ELEMENT metatype (#PCDATA) >
<!-- single classification dimension; the names and their
corresponding values are defined in Section 2 -->
<!ELEMENT dim (#PCDATA) >
<!ATTLIST dim
  name NMTOKEN #REQUIRED >

<!-- generic descriptive elements -->
<!ELEMENT name (#PCDATA) >
<!ELEMENT description (#PCDATA) >
```

The following document type definition defines the XML data structure for ENT category set definitions. Multiple category sets can be defined in one data file. The default extension of category set definition data files shall be “.ecd” for “ENT category set definition”.

```
<?xml version="1.0" encoding="UTF-8" ?>
<!--
```

```
  ENT category set definitions (ECD) DTD
  -->

<!ELEMENT categories (catset*) >
<!-- category set -->
<!ELEMENT catset (name,description?,category+) >
<!-- category definition; attribute "hidden" refers to default
display of the category in visual tools -->
<!ELEMENT category (name,description?,classifier)>
<!ATTLIST category
  hidden (yes|no) "no" >
<!ELEMENT classifier (dim)*>
<!ELEMENT dim (#PCDATA)>
<!ATTLIST dim
  name NMTOKEN #REQUIRED >
```

B.2 Concrete Component Representation

This document type definition defines the XML data structure for ENT component representation. The representation allows multiple components from several component models to be mixed in one data file. The default extension of component representation data files shall be ".emr" for "ENT module/component representation".

```
<?xml version="1.0" encoding="UTF-8" ?>
<!--
  ENT module/component representation (EMR) DTD
  -->

<!ELEMENT entrep (component)* >
<!ELEMENT component (provider, namespace, name, tags?, element*)>
<!ATTLIST module
  model NMTOKEN #REQUIRED>

<!-- component naming -->
<!ELEMENT provider (#PCDATA)>
<!ELEMENT namespace (#PCDATA)>
<!ELEMENT name (#PCDATA)>

<!-- generic description element -->
<!ELEMENT description (#PCDATA) >

<!-- single interface element -->
<!ELEMENT element (name,typel,tags,trait) >
<!-- element language type -->
<!ELEMENT type (#PCDATA)>
<!-- component- or element-level tags -->
```

```
<!ELEMENT tags (tag)* >
<!-- single tag; name-value pair -->
<!ELEMENT tag (#PCDATA) >
<!ATTLIST tag
  name CDATA #IMPLIED >
<!-- name of the trait the element belongs to; must refer to
one trait name defined in the EMD for the component model
named in the enclosing "component" element's "name"
attribute. -->
<!ELEMENT trait (#PCDATA) >
```

(Note: There is no separate XML representation for categories applied to a concrete component, since such representation is created by applying category set definition on a concrete component representation. It is only useful at run-time and need not be persistent.)

References

[1] María José Presso. Declarative descriptions of component models as a generic support for software composition. In *Workshop on Component-Oriented Programming (WCOP'00)*, Nice, France, June 2000. Position Paper.

[2] Jonathan Aldrich, Craig Chambers, and David Notkin. ArchJava: connecting software architecture to implementation. In *Proceedings of International Conference on Software Engineering (ICSE'02)*, Orlando, Florida, May 2002. ACM Press.

[3] Robert Allen and David Garlan. Formalizing architectural connection. In *Proceedings of the Sixteenth International Conference on Software Engineering*, pages 71–80, Sorrento, Italy, May 1994.

[4] Juergen Börstler. Feature-oriented classifciation for software reuse. In *Proceedings of the 7th International Conference on Software Engineering and Knowledge Engineering*, pages 204–211, Rockville, MD, USA, June 1995.

[5] Premysl Brada. The ENT model: A general model for software interface structuring. Technical Report DCSE/TR-2002-10, Department of Computer Science and Engineering, University of West Bohemia, Pilsen, Czech Republic, 2002.

[6] Premysl Brada. *Specification-Based Component Substitutability and Revision Identification*. PhD thesis, Charles University in Prague, August 2003.

[7] Přemysl Brada. Metadata support for safe component upgrades. In *Proceedings of COMPSAC'02, the 26th Annual International Computer Software and Applications Conference*, Oxford, England, August 2002. IEEE Computer Society Press.

[8] T. Coupaye et al. *The Fractal Composition Framework (Version 1.0)*. The ObjectWeb Consortium, July 2002.

[9] Fraunhofer Institute FOCUS et al. *Streams for CORBA Components RFP - Initial Submission*, October 2003.

[10] Svend Frolund and Jari Koistinen. Quality of service specification in distributed object systems design. In *Proceedings of the 4th USENIX Conference on Object-Oriented Technology and Systems (COOTS)*, Santa Fe, New Mexico, April 1998.

[11] Jun Han. A comprehensive interface definition framework for software components. In *Proceedings of 1998 Asia-Pacific Software Engineering Conference*, pages 110–117, Taipei, Taiwan, December 1998. IEEE Computer Society.

[12] International Organization for Standardization. *Ada 95 Reference Manual: Language and Standard Libraries*, 1995. International Standard ISO/IEC 8652:1995(E), Version 6.0.

[13] David C. Luckham and James Vera. An event-based architecture definition language. *IEEE Transactions on Software Engineering*, 21(9):717–734, 1995.

[14] Chris Lüer and André van der Hoek. Composition environments for deployable software components. Technical Report UCI-ICS-02-18, University of California Irvine, August 2002.

[15] R. Marvie, P. Merle, and M. Vadet. The OpenCCM platform. http://corbaweb.lifl.fr/OpenCCM/index.html, 2001.

[16] Nenad Medvidovic and Richard Taylor. A classification and comparison framework for software architecture description languages. *IEEE Transactions on Software Engineering*, 26(1), January 2000.

[17] Neno Medvidovic. A classification and comparison framework for software architecture description languages. Technical report UCI-ICS-TR-97-02, University of Carolina, Irvine, 1997.

[18] Bertrand Meyer. *Eiffel: The Language*. Prentice Hall, 1992.

[19] Microsoft Corporation and Digital Equipment Corporation. *The Component Object Model Specification*, October 1995.

[20] R. Monge, C. Alves, and A. Vallecillo. A graphical representation of cots-based software architectures. In *Proceedings of IDEAS*, pages 126–137, La Habana, Cuba, April 2002.

[21] Object Management Group. *The Unified Modeling Language v1.4*, 2001.

[22] Object Management Group. *The Common Object Request Broker: Core Specification (Version 3.0)*, December 2002. OMG Specification formal/02-12-06.

[23] Object Management Group. *CORBA Components, Version 3.0*, 2002. OMG Specification formal/02-06-65.

[24] Object Management Group. *Meta Object Facility (MOF) Specification, version 1.4*, 2002. OMG Specification formal/02-04-03.

[25] Object Management Group. *UML Profile for Enterprise Distributed Object Computing Specification*, 2002. OMG Specification ptc/02-02-05.

[26] Object Management Group. *UML 2.0 Superstructure Specification*, 2003. OMG Final Adopted Specification ptc/03-08-02.

[27] Object Management Group. *Metamodel and UML Profile for Java and EJB Specification*, February 2004.

[28] Allen Parish, Brandon Dixon, and David Hale. Component based software engineering: A broad based model is needed. Technical report, The University of Alabama, Tuscaloosa, AL, USA, April 1999.

[29] David L Parnas. On the criteria to be used in decomposing systems into modules. *Communications of the ACM*, December 1972.

[30] M. Peterson. *DCE: A Guide to Developing Portable Applications*. McGraw-Hill, 1995.

[31] František Plášil, Dušan Bálek, and Radovan Janeček. SOFA/DCUP: architecture for component trading and dynamic updating. In *Proceedings of ICCDS'98*, Annapolis, Maryland, USA, 1998. IEEE CS Press.

[32] R. Prieto-Diaz and P. Freeman. Classifying software for reusability. *IEEE Software*, 18(1), January 1987.

[33] Uwe Rastofer. Modeling with components – towards a unified component meta-model. In *ECOOP Workshop on Model-based Software Reuse*, Malaga, Spain, 2002. Available at http://www.info.uni-karlsruhe.de/-˜pulvermu/workshops/ECOOP2002/papers.shtml.

[34] Rational Software. *Rational Rose*. http://www.rational.com/products/rose/.

[35] Ralf H. Reussner and Heinz W. Schmidt. Using parameterised contracts to predict properties of component based software architectures. In Ivica Crnkovic, Stig Larsson, and Judith Stafford, editors, *Workshop On Component-Based Software Engineering (in association with 9th IEEE Conference and Workshops on Engineering of Computer-Based Systems), Lund, Sweden, 2002*, April 2002.

[36] D. Rogerson. *Inside COM*. Microsoft Press, 1997.

[37] Frédérick Seyler and Philippe Aniorte. A component meta model for reused-based system engineering. In Jean Bezivin and Robert France, editors, *Workshop in Software Model Engineering*, Dresden, Germany, 2002. Available at http://www.metamodel.com/wisme-2002/.

[38] Mary Shaw et al. Abstractions for software architecture and tools to support them. *IEEE Transactions on Software Engineering*, 21(4):314–335, March 1995.

[39] Sun Microsystems, Inc. *JavaBeans API Specification (version 1.01)*, 1997. Available at http://java.sun.com/products/javabeans/docs/spec.html.

[40] Sun Microsystems, Inc. *Enterprise JavaBeans(TM) Specification (Version 2.0)*, August 2001. Available at http://java.sun.com/products/ejb/docs.html.

[41] Clemens Szyperski. *Component Software*. ACM Press, Addison-Wesley, 1998.

[42] Richard N. Taylor et al. A component- and message-based architectural style for GUI software. *Software Engineering*, 22(6):390–406, 1996.

[43] TogetherSoft Corporation. *Together ControlCenter 6.0 User Guide*, 2002. http://www.togethersoft.com/products/controlcenter/.

[44] Rob van Ommering. Configuration management in component based product populations. In *Proceedings of 10th International Workshop on Software Configuration Management (part of ICSE 2001)*, Toronto, Canada, May 2001.

[45] Amy Moormann Zaremski and Jeanette Wing. Specification matching of software components. *ACM Transactions on Software Engineering and Methodology*, 6(4), October 1997.