

A Classification Framework for Component Models

Ivica Crnkovic, *Member, IEEE*, Séverine Sentilles, *Student Member, IEEE*, Aneta Vulgarakis, *Student Member, IEEE*, and Michel Chaudron, *Member, IEEE*

Abstract— The essence of component-based software engineering is embodied in component models. Component models specify the properties of components and the mechanism of component compositions. In last decade a rapid growth, a plethora of different component models has been developed, using different technologies, having different aims, and using different principles. This has resulted in a number of models and technologies which have many similarities, but also principal differences, and in a lot cases unclear concepts. Component-based development has not succeeded in providing standard principles, as for example object-oriented development. In order to increase the understanding of the concepts, and to easier differentiate component models, this paper provides a Component Model Classification Framework which identifies and discusses the basic principles of component models. Further the paper classifies a certain number of component models using this framework.

Index Terms— Software components, software component models, component life cycle, extra-functional properties, component composition



1 INTRODUCTION

Component-based software engineering (CBSE) is an established area of software engineering. The inspiration for “building systems from components” in CBSE comes from other engineering disciplines, such as mechanical or electrical engineering, software architecture. The techniques and technologies that form the basis for component models originate mostly from object-oriented design and Architecture Definition Languages (ADLs). Since software is in its nature different from the physical world, the translation of principles from the classical engineering disciplines into software is not trivial. For example, the understanding of the term “component” has never been a problem in the classical engineering disciplines, since a component can be intuitively understood and this understanding fits well with fundamental theories and technologies. This is not the case with software. The notation of a software component is not clear: its intuitive perception may be quite different from its model and its implementation. From the beginning, CBSE struggled with a problem to obtain a common and a sufficiently precise definition of a software component. An early and probably most commonly used definition coming from Szyperki [1] (“A software component is a unit of composition with contractually specified interfaces and explicit context dependencies only. A

software component can be deployed independently and is subject to composition by third party”) focuses on characterization of software component. In spite of its generality it was shown that this definition is not valid for a wide range of component-based technologies (for example those which do not support contractually specified interface or independent deployment). In the definition of Heineman and Council [2] (“A software component is a software element that conforms to a component model and can be independently deployed and composed without modification according to a composition standard”), the component definition is more general – actually a component is specified through the specification of the component model. The component model itself is not specified. This definition can be even more generalized in respect to the component specification, but component model can be expressed more precisely [3]:

Definition: *A Software Component is a software building block that conforms to a component model. A Component Model defines standards for (i) properties that individual components must satisfy and (ii) methods, and possibly mechanisms, for composing components.*

This generic definition allows the existence of a wide spectrum of component models, which is also happening in reality; on the market and in different research communities, there exists many component models with different characteristics. However, it makes it more difficult to properly understand the Component-Based (CB) principles. In particular, this is true since CB principles are not clearly explained and formally defined. In their diversities component models are similar to ADLs; there are similar mechanisms

Ivica Crnkovic, Séverine Sentilles and Aneta Vulgarakis are with Mälardalen University, School of Innovation, Design and Engineering, Box 883, SE-721 23 Västerås, Sweden. E-mail: {ivica.crnkovic, severine.sentilles, aneta.vulgarakis}@mdh.se.

Michel Chaudron is with Universiteit Leiden, Faculty of Mathematics and Natural Sciences, Leiden Institute of Advanced Computer Science. P.O. Box 9512, 2300 RA Leiden, The Netherlands, E-mail: chaudron@liacs.nl

Manuscript received August, 2008. Please note that all acknowledgments should be placed at the end of the paper, before the bibliography.

and principles but many variations and different implementations. For this reason there is a need for having a framework which can provide a classification and comparison between different component models in a similar manner as it was done for ADLs [4,5]. In addition, a framework can help in the selection of a particular component model or in the design of a new component model.

In this paper, we propose a classification and comparison framework for component models. Since component models and their implementations in component technologies cover a large range of different aspects of the development process, we group these aspects in several dimensions and build a multidimensional framework that counts different, yet equally important, aspects of component models. We have also analyzed a considerable number of component models, and compared their characteristics. The results of the comparison have led to some observations which are discussed in the paper.

Our research methodology was based on several iterations of (i) observations and analysis, (ii) classification, and (iii) validation; in the first iteration, based on the literature related to general principles of component-based software engineering and existing classification [1]-[13], the classification model was applied to a set of component models, and discussed with several CBSE and empirical software engineering researchers and experts from different engineering domains. The resulting analysis and discussions have led to a refinement of the framework. In the next iterations the refined framework was applied to new component models and discussed with new researchers. The process (which lasted more than one year) has been completed when in the last iteration all new component models complied well with the framework. Another important issue that we learned was related to a decision what to define as a component model and what not. This is discussed in section three.

The remainder of this paper is organized as follows. Section two motivates, explains and defines the different dimensions of the classification framework. Section three discusses the criteria for inclusion of different models/technologies into to component models survey and the classification framework. The comparison framework and observations from the comparison are presented in section four. Related work is covered in section five and section six concludes the paper. A very brief overview of the selected component models on which the classification framework has been mapped is given in appendix A.

2 THE CLASSIFICATION FRAMEWORK

The main concern of a component model is to (i) provide rules for the specification of component properties and (ii) provide rules and mechanisms for component composition, including the composition rules of component properties. These main principles hide many complex mechanisms and models, and have

significant differences in approaches, concerns and implementations. For this reason we cannot simply list all possible characteristics to compare the component models; rather we want to group particular characteristics that have similar concerns i.e. that describe the same or related aspects of component models. Starting from the definition of component models, we distinguish specification of components from specification of communication. Component specifications express component functions (typically in a form of signatures), and extra-functional properties. Most of the component models include only specification of functions, in form of interfaces. Extra-functional properties, if specified at all, are defined either in a form of extended interface or as component metadata. The functional part of an interface is directly related to interaction between components and realized through construction mechanisms using different interaction (architectural) styles. Communication between components is usually not explicitly specified, but there are different types of communications that are assumed in component models.

Finally different component models cover different phases in a component lifecycle; while some support only the modeling phase, others also provide mechanisms supporting the implementation and run-time phase.

In this paper we divide the fundamental principles and characteristics of component models into the following dimensions.

1. **Lifecycle.** *The lifecycle dimension identifies the support provided (explicitly or implicitly) by the component model, in certain points of a lifecycle of components or component-based systems.* Component-Based Development (CBD) is characterized by the separation of the development processes of individual components from the process of system development. There are some synchronization points in which a component is integrated into a system, i.e. in which the component is being bound. Beyond those points, the notion of components in the system may disappear, or components can still be recognized as parts of the system.
2. **Constructs.** *The constructs dimension identifies (i) the component interface used for the interaction with other components and external environment, and (ii) the means of component binding and communication.* In some component models, the interface comprises the specification of all component properties, including both functional and extra-functional, but in most cases, it only includes a specification of functional properties. Directly correlated to the interface are the components' interoperability mechanisms. All these concepts are parts of the "construction" dimension of CBD.
3. **Extra-Functional Properties.** *The extra-functional properties dimension identifies specifications and support that includes the provision of property values and*

means for their composition. In certain domains (for example real-time embedded systems), the ability to model and verify particular properties is equally important but more challenging than the implementation of functional properties.

4. **Domains.** *This dimension shows in which application and business domains component models are used or supposed to be used.* It indicates the specialization, or the generality of component models.

In these four dimensions, we comprise the main characteristics of component models but, of course, there are also other characteristics that can differentiate them. For example, since in many cases component models are built on a particular implementation technology, many characteristics come directly from this supporting implementation technology and are not visible in component models themselves. Still the intention with the classification and comparison model is to comprise the main characteristics of component models.

2.1 Lifecycle

While CBSE aims at covering the entire lifecycle of component-based systems, component models provide only partial lifecycle support and usually are related to the design, implementation and integration phases.

The overall component-based lifecycle is separated into several processes; building components, building systems from components, and assessing components [6]. Some component technologies provide certain support in these processes (for example maintaining component repositories, exposing interface, component deployment).

The component-based paradigm has extended the integration activities up to the run-time phase; certain component technologies provide extended support for dynamic and independent deployment of components into running systems. This support is reflected in the design of many component models. In contrast, in other component models components only exist as separate units in the development stage and become assimilated into a system when the system is built. In this case the system at run-time is monolithic. However not all component models consider this integration phase. We can clearly distinguish different component models that focus on one particular or more phases and such phases can be different for different component models. Some component technologies start in the design phase (e.g. Koala which has an explicit and dedicated design notation of components and other elements of the component model), while other component technologies focus on the implementation phase (e.g. COM, EJB). For this reason one important dimension of our component model classification lifecycle support. In our classification, we distinguish the lifecycle of components from the lifecycle of the component-based system, which are different [3,7] and are not necessary temporally related – they are ongoing in

parallel and have some synchronization points. We identify the following stages of the component lifecycle.

- (i) *Modelling stage.* The component models provide support for the modelling and the design of component-based systems and components. Models are used either for the architectural description of the systems and components (e.g. ADLs), or for the specification and the verification of particular system and component properties (e.g. statecharts, resource usage models, performance models).
- (ii) *Implementation stage.* The component model provides support for production of code. The implementation may stop with the provision of the source code, or may continue up to the generation of a binary (executable) code. The existence of executable code is a precondition for the dynamic deployment of components (during run-time).
- (iii) *Packaging stage.* Because components are the central unit in CBSE, there is a need for their storage and packaging – either in a repository or for distribution. A component package is a set of metadata and code (source or executable). Accordingly, the result of this stage can be a file, an archive, or a repository in which the packaged components reside prior to decisions about how they will be run in the target environment. For example, in Koala, components are packed into a file system-based repository, with a folder per component. The folder includes a number of files: Component Description Language (CDL) file and, a set of C and header files, test file and different documents. Another example of packaging is achieved in the EJB component model. There, packaging is done through jar archives, called `ejb-jar`. Each archive contains XML deployment descriptor, component description, component implementation and interfaces.
- (iv) *Deployment stage.* At a certain point of time, a component is integrated into a system. This activity may happen at different phases of the systems' lifecycle. In general, the components can be deployed at:
 - 1) *compilation time*, so it is no longer possible to change the way the components interact with each other. For instance, Koala components are deployed at compilation time and they use static binding by following naming conventions and generated renaming macros.
 - 2) *run time* as separate units by using means such as registers (COM) or containers (CCM,EJB). For example, CORBA components are deployed at run time in a container by using information of the deployment descriptor packed with the component implementation.

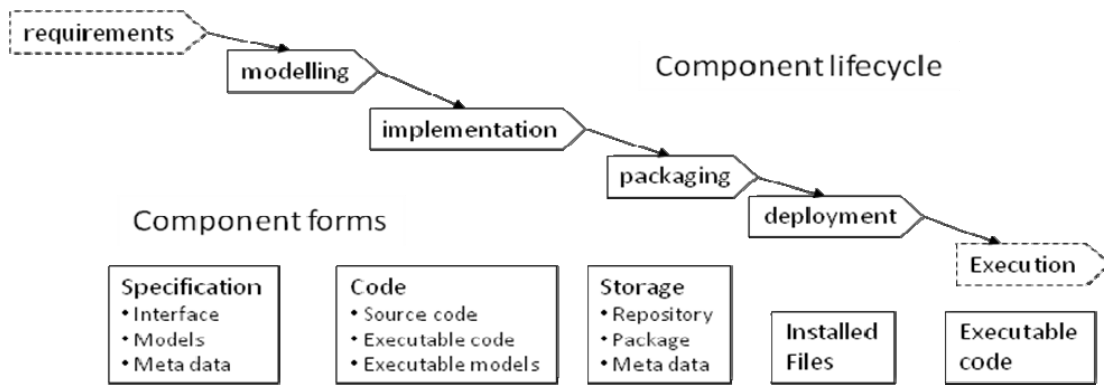


Fig. 1. Component lifecycle and component forms

Figure 1 illustrates different stages in a component lifecycle and the associated forms of the components. Through the stages some of the forms are transformed into new ones, some remains, while some disappear. In the figure the requirements and execution phase are denoted with the dashed lines which indicate that in these stages components do not necessary exist as independent units. The forms of the components will be different across phases for different component models.

2.2 The constructs

As defined in [8], the verb “construct” means “to form something by putting different things together”, so in applying this definition to the CBSE domain, we define by the “Constructs” dimension, the way components are connected together within a component model in order to provide communication means. But although this communication aspect is of primordial importance, it is not often expressed explicitly. Instead, it is reflected implicitly by some underlying mechanisms. This should be distinguished from specifications of functional and sometimes extra-functional properties in a form of component interfaces. Consequently, a component interface has a double role: It first specifies the component properties (functional and possibly extra-functional), and second, it identifies the connection points through which components are interconnected.

2.2.1 Interface

Interface specification is the characteristic “sine qua non” of a component model. Interfaces are defined either by using special languages, or elements of programming languages. Several languages exist that specify components’ interfaces and their connections: modeling languages, such as UML or different Architecture Description Languages (ADLs), particular specification languages, such as Interface Definition Languages (IDLs), programming languages such as Interface in Java, or abstraction class in C++, or some additions built directly in a programming language, such as pre-defined structs in C. In case of special languages, the interface specifications are translated to a programming language. In a few cases (e.g. COM), the interface is also defined in a binary format in order to

have a standard representation at deployment and run-time. Some mechanisms such as introspection in Java are also used to discover the interfaces of a component at run-time.

The component models that use programming languages or their extensions for component specification, also inherit properties of these languages. For example the component models that use object-oriented languages utilize the concepts of classes and (interface) inheritance. Typically a component is expressed as a class in which the interface is defined as a set of operations/functions and attributes. However there exist other types of interfaces so called “port-based” where ports are entries for receiving/sending different data types and events. Note that this concept is different from the concept in UML 2.0 [9] in which a port is defined as a set of specifications.

Some component models distinguish also the “provides”-part (i.e. the specification of the functions that the component offers) from the “requires”-part (i.e. the specification of the functions the component requires) of an interface.

In order to ensure that a component will behave as expected according to its specification and operational mode, and in order to ensure that a component is supplied with expected input and environment the notion of contract has been adjoined to interfaces. According to [10], contracts can be classified hierarchically in four levels which, if taken together, may form a global contract. We only adopt the three first levels in our classification since the last level “contractualizes” only the extra-functional properties and this is not in direct relation with interoperability

- *Syntactic level*: describes the syntactic aspect, also called signature, of an interface. This level ensures the correct utilisation of a component. That is to say that the “calling-component” must refer to the proper types, fields, methods, signals, ports and handles the exceptions raised by the “responding-component”. This is the most common and most easy agreement to certify as it relies mainly on an, either static or dynamic, type checking technique.
- *Semantic level*: reinforces the previous level of contracts in certifying that the values of the parameters as well as the persistent state variables are

within the proper range. This can be asserted by pre-conditions, post-conditions and invariants. A generalization of this level can be assumed as semantics.

- *Behaviour level*: dynamic behaviour of services. It expresses either the composition constraints (e.g., constraints on their temporal ordering) or the internal behaviour (e.g. dynamic of internal states).

Finally, the constructs dimension refers to the notions of reusability and evolvability, which are important principles of CBSE. Indeed many component models are endowed with diverse features for supporting them; one typical solution is the ability to add new interfaces to a component. This makes it possible to embody several versions or variants of functions in the component.

2.2.2 Composition of constructs

While compositions in general consider compositions of component properties, both functional and extra-functional, compositions of constructs are related to components interactions. Constructs compositions are implemented as connections of interaction channels and the process of this connection is called binding. The binding mechanism is related to the component lifecycle; it can occur at compilation time (when a compiler provides connections between components using programming language mechanisms), or at run-time, in which connection mechanisms are utilised that are provided by the underlying run-time infrastructure. Such a run-time infrastructure may consist of dedicated component middleware, and/or a component framework or of a common operating system or middleware.

A so-called “docking interface” method is commonly used when binding occurs at run-time. This docking interface does not offer any application functionality, but serves instead for managing the binding and subsequent interaction between a component and the underlying run-time infrastructure.

In many component models (e.g. CCM, EJB) the composition specification is location-transparent; the run-time location of components (placed on a local or a remote node) is specified separately from the binding information. This information about the location is used in the deployment phase.

Connectors, introduced as distinct elements in ADLs, are not common among the first class citizens in most component models. Connectors are mediators in the connections between components and have a double purpose: (i) enabling indirect composition (so called exogenous compositions), and (ii) introducing additional functionality, especially for mediation between components. In the exogenous composition information concerning the binding resides outside of the components; the components have no knowledge of who they are connected to. Exogenous composition enables more seamless evolution because it separates changes to components from changes to their bindings.

In several component technologies, connectors are implemented as special types of components, such as adaptors or proxies, either to provide additional functional or extra-functional properties, or to extend the means of intercommunication. In direct (endogenous) type of composition the components are connected directly through their interfaces. Information concerning the binding resides inside components.

The interface specification implicitly defines the type of interaction between components to comply with particular architectural styles. In most cases, a particular component models provide a single basic interaction style (for example, “request-response” or “pipe & filter”, but others, such as Fractal, Pin and BIP allow the construction of different architectural styles.

An important question related to the composability of components has concerned the research community [11]: Can the assemblies of components (by assemblies we assume a set of components mutually connected) be treated as components themselves, i.e. is the composition hierarchical? There are two kinds of assemblies supported by existing component technologies. The first is the first order assembly which is not treated as a component in the component model. This type of assembly is merely a set of components of an arbitrary form, creating an application or a part of an application. In terms of binding the component models refer to “horizontal composition” or “horizontal binding”. The second type of assembly is hierarchical which means that the assembly, created from components, again satisfies the properties that an individual component should satisfy according to the component model. In that case we refer to “hierarchical composition” or “hierarchical binding”. The criteria for vertical composition are related to constructs (interface specification and the interaction), and possibility extra-functional properties. Most of the component models support partial vertical composition. For example interfaces can be composed recursively in modeling phase, but not in the deployment phase (in particular when deployment is performed during run-time).

2.2.3 Constructs classifications

Following the observations and reasoning from above we identify the following classification characteristics for interfaces and connections in the constructs dimension.

- (i) *Interface specification*, in which different characteristics allowing the specification of interfaces are identified:
 - 1) The distinction of interface type: operation-based (e.g. methods invocations) and port-based interface (e.g. data passing).
 - 2) The distinction between the provides-part and the requires-part of an interface.
 - 3) The existence of some distinctive features appearing only in this component model (such as special type of ports, optional operations).
 - 4) The language used to specify the interface.

- 5) Interface levels which describes the levels of contractualisation of the interfaces, namely syntactic, semantic and/or behaviour level.
- (ii) *Interactions*, which comprise the following characteristics:
- 1) Interaction style which describes the main underlying architectural style used.
 - 2) Communication type which details mainly if the communication used are synchronous and/or asynchronous.
 - 3) Binding type describes the way components may be linked together through the interfaces. It is realized in two subtypes:
 - a) The exogenous/endogenous sub-category describing whether the component model includes connectors as architectural elements, and
 - b) The hierarchical sub-category expressing the possibility of having a hierarchical composition of components (horizontal composition is an intrinsic part of all component models, thus it is implicitly assumed, and not put in the classification framework).

2.3 Extra-Functional Properties

Properties are used in the most general sense as defined by standard dictionaries, e.g.: “a construct whereby objects and individuals can be distinguished” [11]. There is no unique taxonomy of properties, and consequently many property classification frameworks can exist. One commonly used classification is to distinguish functional from extra-functional properties. While functional properties describe functions or services of an object, extra-functional properties (EFPs) specify the quality, or in general a characteristic of interest, of objects. In CBSE, there is also a distinction between component properties and system properties. A property at the system level can result from the composition of the same properties of constituent components, but also from the composition of different properties. In latter case such property can exist only on a system level. Such properties are called emerging properties.

2.3.1 Composition of extra-functional properties

EFPs can be complex and abstract or, they can be tangible and concrete. Examples of abstract (and complex) properties are dependability or performance and examples of tangible properties are memory footprint or scalability. Complex properties are typically the result of the composition of several more tangible properties. An important concern of CBSE is composition of properties expressed in the following way.

For an assembly A that is composed of component C1 and C2

$$A = C1 \circ C2$$

express a property of the assembly as a composition of properties of the components

$$P(A) = P(C1) \circ P(C2).$$

Different EFPs have different characteristics and hence are specified in very different ways. Also computing the compositions of EFPs require different composition theories for different EFPs. In relation to composability, one of the challenges of CBSE is predictability. To enable analysis at the design stage and to avoid expensive, tedious and non-accurate tests and increase reusability, a lot of efforts has been made in CBSE research communities to design component models that enable predictability.

According to [11], the properties can be classified according to types of compositions in the following basic categories.

- *Directly composable properties* (example: static memory): A property of an assembly is a function of, and only of, the same property of the components involved.

$$P(A) = f(P(C1), \dots, P(Ci), \dots, P(Cn))$$

- *Architecture-related properties* (example: performance): A property of an assembly is a function of the same property of the components and of the software architecture.

$$P(A) = f(SA, \dots, P(Ci), \dots), \quad i=1..n$$

SA = software architecture

- *Derived properties* (example: response time vs. execution time): A property of an assembly depends on several different properties of the components.

$$P(A) = f(SA, \dots, P_i(C_j), \dots), \quad i=1..m, \quad j=1..n$$

P_i = component properties
 C_j = components

- *Usage-dependent properties* (example: reliability): A property of an assembly is determined by its usage profile.

$$P(A, U) = f(SA, \dots, P_i(C_j, U), \dots), \quad i=1..m, \quad j=1..n$$

U = Usage profile

- *System environment context properties* (example: safety): A property is determined by other properties and by the state of the system environment.

$$P(S, U, X) = f(SA, \dots, P_i(C_j, U, X), \dots), \quad i=1..m, \quad j=1..n$$

S = system, X = system context

This idealised classification indicates the limitations of the compositions of EFPs. Determining the

compositions of properties of components becomes feasible when restrictions are imposed on the design of individual components (by means of rules/constraints in of the component model) and system architecture. For example static memory usage of an assembly can be defined as the sum of static memory usage of involved components, but only using particular composition policies (e.g. no concurrency). In this way, we can obtain predictability of the considered property. Other properties are related to usage profile and if we cannot predict usage profile we cannot predict the system properties. Some other properties are not composable at all, and in that case we cannot predict their composition.

2.3.2 Management of extra-functional properties

Even if EFPs are not composable, they can be manageable, i.e. they can be obtained by using some solutions encapsulated in component models and standardized architectural solutions.

Different types of EFP management exist according to the way the component models handle them. We distinguish two main dimensions (Fig. 2):

- (i) A property is managed by the components (endogenous EFP management - approaches A and B), or by the system (exogenous EFP management - approaches C and D) or managed.
 - (ii) A property is managed on a system-wide scale (approaches B and D), or the property is managed on a per-collaboration basis (approaches A and C).
- *Approach A (endogenous per collaboration)*. A component model does not provide any support for EFP management, but it is expected that a component developer implements it. This approach makes it possible to include EFP management policies that

are optimized towards a specific system, and also can cater for adopting multiple policies in one system. This heterogeneity may be particularly useful when COTS components need to be integrated. On the other hand, the fact that such policies are not standardized may be a source of architectural mismatch between components. This approach can hardly manage emerging properties.

- *Approach B (endogenous systemwide)*. In this approach, there is a mechanism in the component execution platform that contains policies for managing EFPs for individual components as well as for EFPs involving multiple components. The ability to negotiate the manner in which EFPs are handled requires that the components themselves have some knowledge about how the EFPs affect their functioning. This is a form of reflection.
- *Approach C (exogenous per collaboration) and Approach D (exogenous systemwide)*. In these approaches the components are designed such that they address only functional aspects and not EFP. Consequently, in the execution environment, these components are surrounded by a container. This container contains the knowledge on how to manage EFPs. Containers can either be connected to containers of other components (approach C) or containers can interact with a mechanism in the component execution platform that manages EFPs on a system wide scale (approach D). The container approach is a way of realizing separation of concerns in which components concentrate on functional aspects and containers concentrate on extra-functional aspects. In this way, components become more generic because no modification is re-

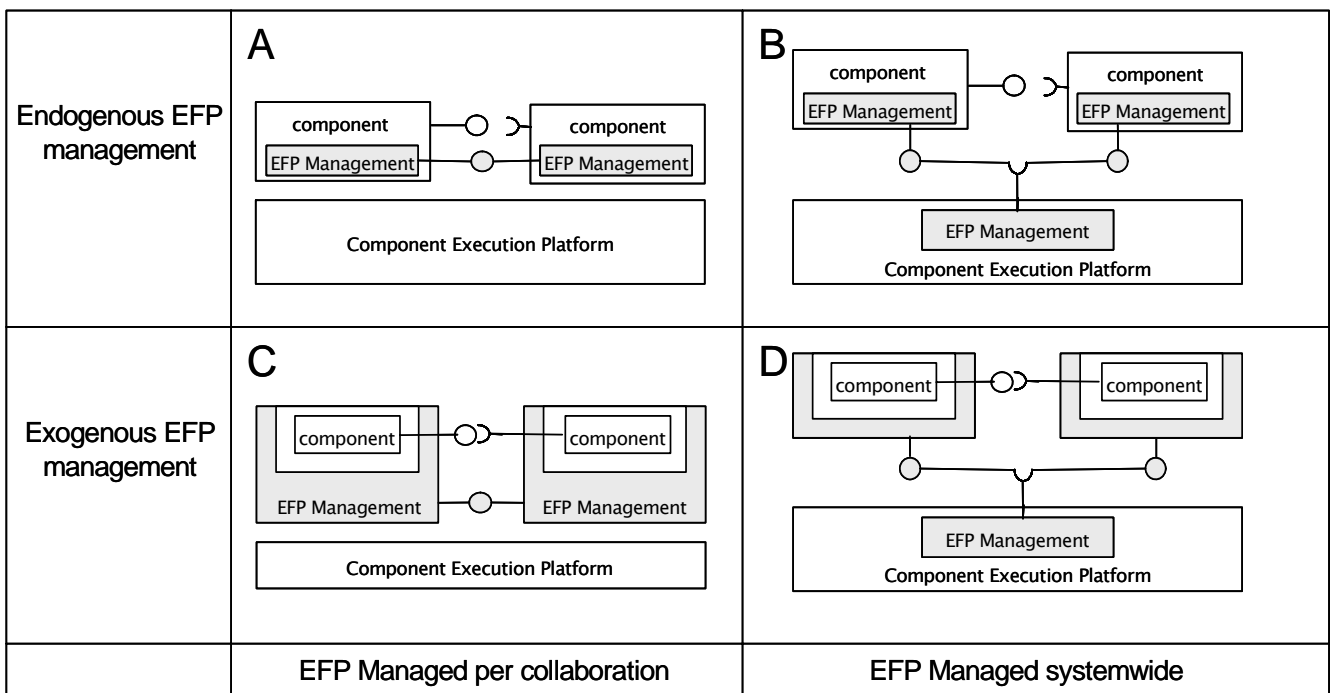


Fig. 2. Management of extra-functional properties

quired to integrate them into systems that may employ different policies for EFPs. Since these components do not address EFPs, another advantage is that they are simpler and hence cheaper to implement. A disadvantage of this approach might be a degradation of the system performance.

2.3.3 Extra-functional properties classification

For the EFPs we provide a classification in respect to the following questions:

- (i) *Management of EFPs*: Which type of management (if any) is provided by the component model?
- (ii) *EFP specification*: Does the component model contain means for specification and management of specific EFPs. If yes, which properties or which types of properties?
- (iii) *Composability of EFPs*: Does the component model provide means, methods and/or techniques for composition of certain extra-functional properties and/or what type of composition?

2.4 Domains

Some component models are aimed at specific application domains as for instance consumer electronics or information systems. In such cases, requirements from the application domain penetrate into the component model. The benefits of a domain-specific component models are that the component technology facilitates achieving certain requirements. Such component models are, as a consequence, limited in generality and will not be so easily usable in domains that are subject to different requirements.

Some component models are of general-purpose. They provide basic mechanisms for the specification and the composition of components, but do not assume any specific architecture beyond general assumptions (like interaction style, support for distributed systems, compilation or run-time deployment). A general solution that enables component models to be both generally applicable but to also cater for specific domains is through the use of optional frameworks. A framework is an extension of a component model that may be used, but is not mandatory in general.

There is a third type of component models, namely generative; they are used for instantiation of particular component models. They provide common principles, and some common parts of technologies (for example modeling), while other parts are specific (for example different implementations).

According to this, we classify the component models as

- (i) General-purpose component models;
- (ii) Specialized component models;
- (iii) Generative component models.

2.5 The Classification overview

Fig. 3 summarizes the classification framework in a graph form.

3 SURVEY OF COMPONENT MODELS

Nowadays a number of component models exist. They vary widely: in usage, in support provided, in concerns, in complexity, in formal definitions, etc.. In our classification of component models, the first question is whether a particular model (or technology, method, or similar) is a component model or not. Similar to biology in which viruses cover the border between life and non-life, there is a wide range of models, from those having many elements of component models but still not assumed as component models, via those that lack many elements of component models, but still are designated as component models, to those which are broadly accepted component models. Therefore, we identify the minimum criteria required to classify a model, or a notation as a component model. This minimum is defined by the definition of component models given in the introduction: A model that defines rules for the design and specification of components and their properties and means of their composition can be classified as a component model. It should be noted that this condition is mandatory, but not sufficient. We have identified several models that fulfill this condition, but still we have not included them in the survey. We can call them “almost” component models.

3.1 “Almost” component models

A wide range of modeling languages contains the term “component” and even (semi)formally specifies components and component compositions. For example in the classification of ADLs [5] one of the basic elements are components (and connectors as means for construction composition). UML 2.0 is even closer to component models since it provides a metamodel for components, interfaces and ports. Still we have deliberately chosen not to select them as component models, in difference to some other classifications (such as [13]). One reason is that their purpose is not component-based development but rather the specification of system architectures. ADLs and UML 2.0 are excellent language candidates for modeling component-based systems and components in the design phase, but are missing other characteristics to be declared as component models. Certain languages derived from UML, such as xUML [14] in which the component specification is translated to an executable entity, are even closer candidates for component models. However xUML and similar languages do not operate with components as first class citizens (for example components are not treated as separate development or executable entities), but components are only architectural elements.

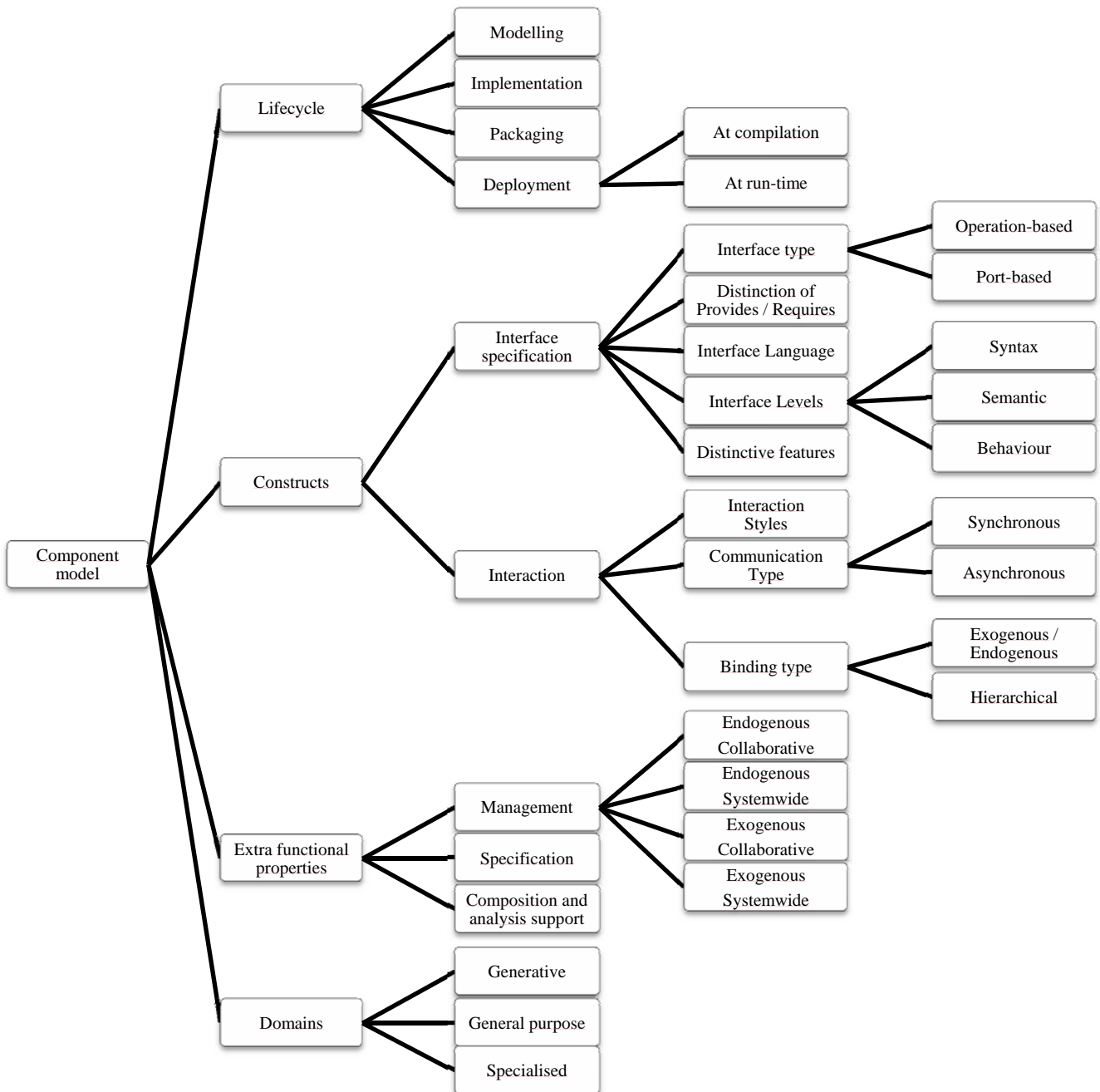


Fig. 3. The hierarchical structure of the classification framework

On the other side of the lifecycle line are services. One can argue that services are special types of components. Services are focused on run-time retrieval and run-time deployment. Similar to components, services are specified by an interface, and provide support for constructs compositions [15]. Still we have not included services in the classifications for similar reasons as for ADLs – their focus is not component-based development. In analogy to ADLs, services are not component models but rather use component models.

Further, we have not included technologies such as Unix processes and “pipe & filter” mechanisms, or modeling environments such as Simulink or Ptolemy [16], as again the components are not the primary concern in these approaches.

Finally we have not included technologies like Eclipse or Photoshop that enable the integration of plugins from third parties and in this way suit well to a part of Szyperski’s definition of components (“deployed independently and is subject to composition by third party”). However they do not provide mechanisms of compositions between components, rather mechanism between components and the underlying platform.

For these “almost component models” one can argue that they are component models or technologies, and that they could be included into the survey. Our position is that their inclusion will break the spirit of the component models as defined in this paper according to the arguments presented.

3.2 Component models

In our classification framework we have selected a number of component models that appeared in the research literature and in practice. While some of them are widely spread and proven, others are used as demonstrators or illustrations of ideas in research.

The classification framework does not show the success of particular component models, or any business model, but it is based on the technical characteristics only. The components models that we have included in the list are shortly referred to in the appendix A.

It is worth to mention that for some of the component models that we found, our selection criteria were satisfied, however because of scarcity of available documentation it was impossible to get the needed detailed information (which usually is a sign that no activity around the model is going on). In these cases, we have decided to omit them from our list.

4 THE COMPARISON FRAMEWORK

The characteristics of the component models are collected in the tables below, following the dimensions in the classification framework, namely lifecycle (Table 1), constructs (Tables 2, and 3), extra-functional properties (Table 4), and the domains (Table 5) lined in the alphabetic order. Following each table, a short discus-

sion gathering observations and their rationales is presented.

4.1 Life-cycle classification

From the observation of Table 1, one can notice that there is a group of component models that do not provide any support for modeling of components or component-based applications, but cover only implementation part (specification and deployment). All these component models belong to the state of the practice and most of them are widely used. Does that mean that the modeling of components is not supposed to be a part of a component model? Or does it mean that other tools, for example general-purpose modeling tools, such as UML or ADLs are used for modeling, while component technologies are used for the implementation? It is partially true that most of the practitioners do not model their systems using formal specification languages, but rather express their design in a non-formal way for documentation purpose only, or in a semiformal way typically using UML. In both cases neither the precise definitions of components nor their interactions are assumed to be of high priority. This is also an indicator of differences between state of the art and state of the practice; many solutions that include modeling of components or their properties from the state of the art have still not been realized or scaled up in practice.

Table 1: Lifecycle Dimension

Component Models	Modelling	Implementation	Packaging	Deployment
AUTOSAR	N/A	C	Non-formal specification of container	At compilation
BIP	A 3-layered representation: behavior, interaction, and priority	BIP Language	N/A	At compilation
BlueArX	N/A	C	N/A	At compilation
CCM	N/A	Language independent	Deployment Unit archive (JARs, DLLs)	At run-time
COMDES II	ADL-like language	C	N/A	At compilation
CompoNETS	Behaviour modeling (Petri Nets)	Language independent	Deployment Unit archive (JARs, DLLs)	At run-time
EJB	N/A	Java	EJB-Jar files	At run-time
Fractal	ADL-like language (Fractal ADL, Fractal IDL), Annotations (Fractlet)	Java (in Julia, Aokell) C/C++ (in Think) .Net lang. (in FracNet)	File system based repository	At run-time
KOALA	ADL-like languages (IDL,CDL and DDL)	C	File system based repository	At compilation
KobrA	UML Profile	Language independent	N/A	N/A
IEC 61131	Function Block Diagram (FBD) Ladder Diagram (LD) Sequential Function Chart (SFC)	Structured Text (ST) Instruction List (IL)	N/A	At compilation
IEC 61499	Function Block Diagram (FBD)	Language independent	N/A	At compilation
JavaBeans	N/A	Java	Jar packages	At compilation
MS COM	N/A	OO languages	DLL	At compilation and at run-time
OpenCOM	N/A	OO languages	DLL	At run-time
OSGi	N/A	Java	Jar-files (bundles)	At run-time and at compilation
Palladio	UML profile	Java	N/A	At run-time
PECOS	ADL-like language (CoCo)	C++ and Java	Jar packages or DLL	At compilation
Pin	ADL-like language (CCL)	C	DLL	At compilation
ProCom	ADL-like language, timed automata	C	File system based repository	At compilation
ROBOCOP	ADL-like language, resource management model	C and C++	Structures in zip files	At compilation and at run-time
RUBUS	Rubus Design Language	C	File system based repository	At compilation
SaveCCM	ADL-like (SaveComp), timed automata	C	File system based repository	At compilation
SOFA 2.0	Meta-model based specification language	Java	Repository	At run-time

The second observation from Table 1 is the fact that most of the component models use object-oriented languages for the implementations with domination of Java. Still there exist component models using other languages, for example imperative programming languages such as C.

It seems that the packaging and component repositories are not in focus of component models. In most cases, certain standard archives are used (such as DLL or JAR packages). The lack of repositories indicates a low focus of reuse, in particular of COTS components.

Deployment at compile time and run-time occurs almost equally often. Deployment at compile time limits the flexibility at run-time, but on the other hand

enables easier predictability, richer composition features (such as hierarchical composition), and more efficient reuse (such as deployment of implementation parts that will be used in the application). This might be a reason why this is the primary deployment style chosen by specialized component models (cf. Table 5).

4.2 Constructs classification

Tables 2 and 3 show interface and interaction specifications of the selected component models. Although the existence of interface is a “*conditio sine qua non*” for component models, and all selected component models identify the interface as an indispensable part of a component, Table 2 shows that interfaces can be of different types. Most interfaces are of operation type,

Table 2: Constructs – Interface Specification

Component Models	Interface type	Distinction of Provides / Requires	Distinctive features	Interface Language	Interface Levels (Syntactic, Semantic, Behaviour)
AUTOSAR	Operation-based Port-based	Yes	AUTOSAR Interface*	C header files	Syntactic
BIP	Port-based	No	Complete interfaces, Incomplete interfaces	BIP Language	Syntactic Semantic Behaviour
BlueArX	Port-based	Yes	N/A	C	Syntactic
CCM	Operation-based Port-based	Yes	Facets and receptacles Event sinks and event sources	CORBA IDL, CIDL	Syntactic
COMDES II	Port-based	Yes	N/A	C header files State charts diagrams	Syntactic Behaviour
CompoNETS	Operation-based Port-based	Yes	Facets and receptacles Event sinks and event sources	CORBA IDL, CIDL, Petri nets	Syntactic Behaviour
EJB	Operation-based	No	N/A	Java Programming Language + Annotations	Syntactic
Fractal	Operation-based	Yes	Component Interface, Control Interface	IDL, Fractal ADL, or Java or C, Behavioural Protocol	Syntactic Behaviour
KOALA	Operation-based	Yes	Diversity Interface, Optional Interface	IDL, CDL	Syntactic
KobrA	Operation-based	N/A	N/A	UML	Syntactic
IEC 61131	Port-based	Yes	N/A	N/A	Syntactic
IEC 61499	Port-based	Yes	Event input and event output Data input and data output	N/A	Syntactic
JavaBeans	Operation-based	Yes	N/A	Java	Syntactic
MS COM	Operation-based	No	Ability to extend interface	Microsoft IDL	Syntactic
OpenCOM	Operation-based	No	Interfaces additional to COM-interface managing lifecycle, introspections, etc.	Microsoft IDL	Syntactic
OSGI	Operation-based	Yes	Dynamic Interfaces	Java	Syntactic
Palladio	Operation-based	Yes	Possibility to annotate interface	UML	Syntactic Behaviour
PECOS	Port-based	Yes	Ability to extend interface	Coco language, Prolog query Petri nets	Syntactic Semantic Behaviour
Pin	Port-based	Yes	N/A	Component Composition Language (CCL), UML statechart	Syntactic Behaviour
ProCom	Port-based	Yes	Data and trigger ports	XML based, Timed Automata	Syntactic Behaviour
Robocop	Port-based	Yes	Ability to extend different types of interface/annotations	Robocop IDL (RIDL), Protocol specification	Syntactic Behaviour
RUBUS	Port-based	Yes	Data and trigger ports	C header files	Syntactic
SaveCCM	Port-based	Yes	Data, trigger, and data-trigger ports	SaveComp (XML- based), Timed Automata	Syntactic Behaviour
SOFA 2.0	Operation-based	Yes	Utility Interface, Possibility to annotate interface and to control evolution	Java, SPC algebra	Syntactic Behaviour

*) Due to a limited space we do not provide a detailed description of specifics of component models. Some of the terms are referred to in the appendix in the corresponding component models, but a more detailed descriptions can be found in the referred to papers.

thus using functions and parameters for defining elements of services the component provides and requires. Still, many component models use ports as interface elements using them for passing data. Such component models are typically used in embedded systems and have their grounds from the concept of hardware components. Some component models do not distinguish between required and provided interface, but the interface is identified with the provided interface, similar to the object-oriented approach. In port-based interfaces, input and output interfaces consisting of ports that receive and send data (often designated as sink and source) are distinguished, which corresponds to provided and required interface..

Since interfaces are an obligatory part of the component specification, all component models provide at least the first level, i.e. syntactic specification. A considerable number of component models also have behavior specifications, in most cases specified by a particular form of finite state machines (state charts, timed automata). Rather few of the component models identify semantic of the interfaces. If semantics are defined, then mostly pre- and post-conditions are used for this. It is worth to mention that interface semantics should not be mixed with other types of semantics that some component models can have (e.g. SaveCCM has execu-

tion semantics which defines the process of the component execution in respect to time).

In line with the type of an interface (operation vs. ports), from the information provided in Table 3 one can conclude that the dominating interaction styles in the component models are "request response" (typically used in client/server architectures), and dataflow and pipe & filter. Some component models have specific additions to interaction styles - event-driven, broadcast or rendez-vous.

Table 4 shows that the dominant communication type in component models is synchronous. Component models that provide support for asynchronous type of communication also support synchronous communication. This indicates that component models are not concerned about architecture (architectural design), but rather targeting detailed design. This fact is also reflected in the use of connectors. Quite a few of the component models have connectors as first class entities, which indicates that components in many component models are implicitly assumed as fine-grained entities, in contrast to architectural components.

Finally, one can observe that many component models do not support vertical binding, i.e. the means for hierarchical composition. V composition of ertical

Table 3: Constructs - Interaction

Component Models	Interaction Styles	Communication Type	Binding Type	
			Exogenous	Hierarchical
AUTOSAR	Request response, Messages passing	Synchronous, Asynchronous	No	Delegation
BIP	Triggering Rendez-vous, Broadcast	Synchronous, Asynchronous	No	Delegation
BlueArX	Pipe&filter	Synchronous	No	Delegation
CCM	Request response, Triggering	Synchronous, Asynchronous	No	No
COMDES II	Pipe&filter	Synchronous	No	No
CompoNETS	Request response	Synchronous, Asynchronous	No	No
EJB	Request response	Synchronous, Asynchronous	No	No
Fractal	Multiple interaction styles	Synchronous, Asynchronous	Yes	Delegation, Aggregation
KOALA	Request response	Synchronous	No	Delegation, Aggregation
KobrA	Request response	Synchronous	No	Delegation, Aggregation
IEC 61131	Pipe&filter	Synchronous	No	Delegation
IEC 61499	Event-driven, Pipe&filter	Synchronous	No	Delegation
JavaBeans	Request response, Triggering	Synchronous	No	No
MS COM	Request response	Synchronous	No	Delegation, Aggregation
OpenCOM	Request response	Synchronous	No	Delegation, Aggregation
OSGi	Request response, Triggering	Synchronous	No	No
Palladio	Request response	Synchronous	No	No
PECOS	Pipe&filter	Synchronous	No	Delegation
Pin	Request response, Message passing, Triggering	Synchronous, Asynchronous	No	No
ProCom	Pipe&filter, Message passing	Synchronous, Asynchronous	Yes	Delegation
Robocop	Request response	Synchronous, Asynchronous	No	No
Rubus	Pipe&filter	Synchronous	No	No
SaveCCM	Pipe&filter	Synchronous	No	Delegation, Aggregation
SOFA 2.0	Multiple interaction styles	Synchronous, Asynchronous	Yes	Delegation

binding is implemented either through delegated interfaces (i.e. selected interfaces from sub-components build up the interface of the composite components) or as aggregation in which the composite component (or in this case just an assembly) include all interfaces of the aggregated components.

4.3 Extra-functional properties classification

From Table 4 an interesting observation can be found: Many components provide certain support for management of EFPs, either system-wide or per container. However a significantly smaller number of component models have formalisms for EFPs specifications. Even smaller number provides means for composition of EFPs. This is particularly true for commercial component models. This is not surprising since many EFPs are either not formally defined, or are considered too complex.

Some of the component models provide architectural solutions (for example redundancy or authentication) which in general improve the quality of systems. These solutions have an impact on different properties (for example reliability and availability). The solutions are usually not part of components themselves but are built into the underlying platform, and added as additional service used in some particular domains (for example COM+ used in MS COM and .NET technologies). While these component models provide support for increasing quality, they still do not support EFP compositions and by this do not obtain “predictability by construction”. Clearly, composition of EFPs still

belongs to research challenges. A vast majority of EFPs that are explicitly managed (specified and composed) belong to resource usage and timing properties.

4.4 Domains classification

From Table 5 we see that the distribution between general-purpose component models and specialized component models is equal. We could expect more specialized; Probably in practice there are more specialized proprietary and not published component models. We have also observed a migration of certain component models. For example OSGI was originally designed for embedded systems, but later has been used as general-purpose component model in different domains. There is also an opposite trend to this. General-purpose component models have been adapted for particular domains by a combination of addition of new features and restriction of some functions. Such examples are CompoNETS and OpenCOM.

Specialized component models belong to two domains: a) embedded systems, and b) information systems. The component models from the embedded systems domain have some common characteristics: the use of the “Pipe & Filter/ dataflow” architectural style, components are usually deployable at compilation time, components are resource-aware and often there is support for management of timing properties. These component models are significantly different from general-purpose component models.

The component models from the information systems domains are significantly more similar to general-

Table 4: Extra-Functional Properties

Component Models	Management of EFP	Properties specification	Composition and analysis support
AUTOSAR	Endogenous per collaboration (A)	N/A	N/A
BIP	Endogenous system wide (B)	Timing properties	Behaviour compositions
BlueArX	Endogenous per collaboration (A)	Resource usage, Timing properties	N/A
CCM	Exogenous system wide (D)	N/A	N/A
COMDES II	Endogenous system wide (B)	Timing properties	N/A
CompoNETS	Endogenous per collaboration (A)	N/A	N/A
EJB 3.0	Exogenous system wide (D)	N/A	N/A
Fractal	Exogenous per collaboration (C)	Ability to add properties (by adding “property” controllers)	N/A
KOALA	Endogenous system wide (B)	Resource usage	Compile time checks of resources
KobrA	Endogenous per collaboration (A)	N/A	N/A
IEC 61131	Endogenous per collaboration (A)	N/A	N/A
IEC 61499	Endogenous per collaboration (A)	N/A	N/A
JavaBeans	Endogenous per collaboration (A)	N/A	N/A
MS COM	Endogenous per collaboration (A)	N/A	N/A
OpenCOM	Endogenous per collaboration (A)	N/A	N/A
OSGi	Endogenous per collaboration (A)	N/A	N/A
Palladio	Endogenous system wide (B)	Performance properties specification	Performance properties
PECOS	Endogenous system wide (B)	Timing properties, generic specification of other properties	N/A
Pin	Exogenous system wide (D)	Analytic interface, timing properties	Different EFP composition theories, example latency
ProCom	Endogenous system wide (B)	Timing and resources	Timing and resources at design and compile time
Robocop	Endogenous system wide (B)	Memory consumption, Timing properties, reliability, ability to add other properties	Memory consumption and timing properties at deployment
Rubus	Endogenous system wide (B)	Timing	Timing properties at design time
SaveCCM	Endogenous system wide (B)	Timing properties, generic specification of other properties	Timing properties at design time
SOFA 2.0	Endogenous system wide (B)	Behavioural (protocols)	Composition at design

Table 5: Domains

Domain	AUTOSAR	BIP	BlueArX	CCM	COMDES II	CompoNETS	EJB	Fractal	KOALA	KobrA	IEC 61131	IEC 61499	JavaBeans	MS COM	OpenCOM	OSGi	Palladio	PECOS	Pin	ProCom	Robocop	RUBUS	SaveCCM	SOFA 2.0
General-purpose				x		x	x	x		x			x	x	x		x		x					x
Specialised	x	x	x		x				x		x	x				x		x		x	x	x	x	
Generative								x													x			x

purpose component models. Typically they have similar characteristics as general-purpose component models, such as use of “request response” interaction, support for run-time run-time deployment, expandable interface, implementation in object-oriented language but they can be distinguished from general purpose component models through specific support for distributed components, data transaction support, interoperability with databases, and some architectural solutions such as redundancy or location transparency.

5 RELATED WORK

Over the last decade, several attempts to identify key features of software components and component models have been proposed: classification or studies of components and interfaces ([17], [18]), interfaces, extra-functional properties ([11]), ADLs ([5]), component models ([13]), characteristics of component models for particular business domains ([12]), among others.

The models presented in [17] and [18] do not consider any component model but rather focus on practical issues of component utilization and reutilization. In [17], the interface classification is split into two categories: application interfaces and platform interfaces. Application interfaces describe the information about the interaction with other components (messages protocol, timing issues to requests) whereas the platform aspect is concentrates on the interaction between components and the executing platform. Similarly in [18] a model for characterizing components is proposed which reuses the classification model of interfaces from [17]. A component is there regarded as the description of three main items (informal description, externals and internals) each of them split into several subelements. The informal description is connected with a set of “human-related” features which can influence on the selection of a component such as its age, its provenance, its level of reuse, its context, its intent and if there is any related component solving a similar problem. The externals are concerned with interaction mechanisms both with other application artifacts and with the platform (application interfaces, platform interfaces, role, integration phase, integration frameworks, technology and non-functional features). Finally the internals are concerned with elements related to the potential information needed during the development process of a system (nature, granularity, encapsulation, structural aspects, behavioural aspects, accessibility to source code).

Similar to our work to some extent, a classification framework to classify each of the proposed models,

frameworks, or standards is proposed in [19], trying to determine what the core features of a software component are. The classification approach is different from ours; it includes identification of a component by a set of elements/characteristics (unit of composition, reuse, interface, interoperability, granularity, hierarchy, visibility, composition, state, extensibility, marketability, and support for OO). The classification includes only business components and business solutions. One of the problems with this classification is the non-orthogonality of some of the characterized items.

In [5], in which ADLs are classified, components are defined as basic elements of ADLs. The components are distinguished by the following features: interface, types, semantics, constraints, evolution, and nonfunctional properties.

In [12], a classification model is proposed to structure the CBSE body of knowledge. All research results are characterized according to several aspects (concepts, processes, roles, product concerns and business concerns, technology, off-the-shelf components and related development paradigms). Here, the component model is only considered as one of the fifty elements in the CBSE items. However, in this work, a more precise taxonomy of application domains is proposed. The paper identifies the following application domains in which component-based approaches are utilized: avionics, command and control, embedded systems, electronic commerce, finance, healthcare, real-time, simulation, telecommunications and, utilities.

In [7], several component models (JB, COM, MTS, CCM, .NET and OSGi) are mainly described according to the following criteria: Interfaces and Assembly using ACME notation, Implementation, and Lifecycle. The models are not compared or valued, but rather these characteristics are described for each component model.

In [13], a study of several component models is presented that considers the following aspects: syntax, semantics and composition through an idealized component-based development lifecycle. A smaller number of component models are considered (also UML and ADLs are included). Based on this study, a taxonomy centered on the composition criterion is proposed, which clarifies at which steps of the development process of a given component model, components can be composed and whether they can be retrieved from a repository to be composed. Further the different types of bindings (compositions) of some of the component models are discussed in more details. This taxonomy does not consider EFPs.

6. CONCLUSION

In this survey, we have presented a framework for the classification and comparison of component models, which identifies issues related to component-based development. This survey indicates that many principles comprised in the component-based approach are not always included in every component model. Many of these principles are taken and further developed from other approaches (OO development, modeling using ADLs) which also contributes to an unclear understanding of component-based development.

The intention of this work is to increase the understanding of component-based approach by identifying the main concerns, common characteristics and differences of component models.

The proposed framework does not include all the elements of all component models since many of them have specific solutions – some related to models, some related to particular technology solutions. Further we have not characterized the component themselves (like implementation, internal behavior, whether components are active or passive, and similar). The framework however identifies the minimal criteria for assuming a model to be a component model and it groups the basic characteristics of the models.

From the results we can recognize some recurrent patterns, such as: general-purpose component models utilize the “request response” style, while in the specialized domains (mostly embedded systems) “pipe & filter/dataflow” is the predominate style. We can also observe that support for composition of extra-functional properties is rather scarce. There are many reasons for that: in practice explicit reasoning and predictability of EFPs is still not widespread, there are unlimited number of different EFPs, and finally the compositions of many EFPs are not only the results of component properties, but also a matter outside component models – for example of system architectures, which makes EFP an aspect that is difficult to handle at the level of traditional implementation languages.

In similarity with other technologies we could expect a convergence of the main characteristics of component models, i.e. becomes more standardized, using more commonly accepted concepts and terminology, even if the number of different component models will not necessary decreased. The aim of this work is to provide a help in this convergence process.

APPENDIX A - SURVEY OF COMPONENT MODELS

In this appendix, we provide a brief overview of component models taken in the survey and their main characteristics. The component models are listed in the alphabetic order. The list should be understood as a provision of some characteristic examples, or examples of widely used component models in Software Engineering.

Note that when listing the component models we have not provided their product name with edition number except for cases in which the edition numbers

are part of the name or indicate significant difference from the previous version.

AUTOSAR (AUTomotive Open System ARchitecture) [20], the new standard in automotive industry is the result of the partnership between several manufacturers and suppliers from the automotive field. The main focus of AUTOSAR is standardization of architecture, architectural components and their interoperability, which allows a separation of development of component-based applications from the underlying platform. AUTOSAR supports both the client-server and sender-receiver communication types. An AUTOSAR software component instance is only assigned to one computer node - Electronic Control Unit (ECU). The AUTOSAR software components are implemented in C. The main focus of AUTOSAR is the architecture not the component model itself.

BIP (Behavior, Interaction, Priority) [21] framework developed at Verimag is used for modelling heterogeneous real-time components. This heterogeneity is considered for components having different synchronization mechanisms (broadcast/rendez-vous), timed components or non-timed components. BIP focuses on component behaviour through a model with a three-layer structure of the components (Behaviour, Interaction and Priority); a component can be seen as a point in this three-dimensional space constituted by each layer. In this model, compound components, i.e. components created from already existing ones, and systems are obtained by a sequence of formal transformations in each of the dimension. BIP comes up with its own programming language but targets C/C++ execution. Some connections to the analysis tools of the IF-toolset [22] and the PROMETHEUS tools [23] are also provided.

BlueArX [24][25] is a component model developed and used by Bosch for the automotive control domain. BlueArX defines a hierarchical component model with focus on design-time, which does not require additional run-time or memory resources on the target hardware. A BlueArX component consists of specification, documentation and implementation (as object or C source code). BlueArX components and interfaces are specified using MSRSW (Manufacturer Supplier Relationship SoftWare), a standardized XML format. Components communicate using client-server and sender-receiver interfaces. Besides name and type the interfaces specification lists additional details (e.g. mapping between internal and physical representation, value range, and physical unit). Other interfaces address component configuration (variation points), calibration data and extra-functional properties, like timing, memory usage or generic specification of other properties.

COMDES II [26], developed at University of Southern Denmark, defines various types of components to address both architectural and behavioral properties of control software systems. It employs a two-level model to specify system architecture. At the first (system)

level a distributed control application is conceived as a network of communicating actors and at the second (actor) level an actor is specified as a software artifact containing a single actor task and multiple I/O drivers. The functional behavior is specified by a composition of different function block instances which implement concrete computation or control algorithms. COMDES II defines four kinds of functional blocks: basic, composite, modal and state machine. The former two can be used to model continuous behavior (data flow) and the later two describe the sequential behavior (control flow). All non-functional information such as physicality, real-time and concurrency is specified with respect to actors.

CompoNETS [26], developed at Université Toulouse 1, is based on CCM where additionally the internal behavior of a software component and inter-component communication are specified by Petri Nets. Accordingly, a mapping from the constructs of the component models (e.g. facets, receptacles, event sources and sinks) to the constructs of Petri-net based behavioral formalism (e.g. places, transitions etc.) is defined. Other characteristics are the same (or very similar) to CCM.

CCM (CORBA Component Model) [28] evolved from Corba object model and it was introduced as a basic model of the OMG's component specification. The CCM specification defines an abstract model, a programming model, a packaging model, a deployment model, an execution model and a metamodel. The metamodel defines the concepts and the relationships of the other models. CORBA components communicate with outside world through ports. CCM uses a separate language for the component specification: Interface Definition Language (IDL). CCM provides a Component Implementation Framework (CIF) which relies on Component Implementation Definition Language (CIDL) and describes how functional and non-functional part of a component should interact with each other. In addition, CCM uses XML descriptors for specifying information about packaging and deployment. Furthermore, CCM has an assembly descriptor which contains metadata about how two or more components can be composed together.

EJB (Enterprise JavaBeans) [29], developed by Sun Microsystems envisions the construction of object-oriented and distributed business applications. It provides a set of services, such as transactions, persistence, concurrency, interoperability. EJB differs three different types of components (The EntityBeans the SessionBean and the MessageDrivenBeans). Each of these beans is deployed in an EJB Container which is in charge of their management at runtime (start, stop, passivation or activation) and EFPs (such as security, reliability, performance). EJB is heavily related to the Java programming language.

Fractal [30] is a component model developed by France Telecom R&D and INRIA. It intends to cover the whole development lifecycle (design, implementation, deployment and maintenance/management) of

complex software systems. It includes several features, such as nesting, sharing of components and reflexivity in that sense that a component may respectively be created from other components, be shared between components and can expose its internals to other components. The main purpose of Fractal is to provide an extensible, open and general component model that can be tuned to fit a large variety of applications and domains. Fractal includes different instantiations and implementations: a C-implementation called Think, which targets especially the embedded systems and a reference implementation, called Julia and written in Java.

Koala [31] is a component model developed by Philips for building software for consumer electronics. Koala components are units of design, development and reuse. Koala has a set of modeling languages: Koala IDL is used to specify Koala component interfaces, its Component Definition Language (CDL) is used to define Koala components, and Koala Data Definition Language (DDL) is used to specify local data of components. Koala components communicate with their environment or other components only through explicit interfaces statically connected at design time. Koala targets C as implementation language and uses source code components with simple interaction model. Koala pays special attention to resource usage such as static memory consumption.

KobrA (KOMPONENTENBASIERTE ANWENDUNGSENTWICKLUNG) [32] is a hierarchical component model that supports a model-driven, UML-based representation of components. In KobrA components are not physical components like in the contemporary physical technologies (e.g. CORBA, EJB, .NET) but logical building blocks of the software system. The components can be constructed in any UML modeling tool and deposited into a file system. They can be compared to subsystems in UML with additional behavior. KobrA uses UML class diagrams to specify structure, functional model to describe functionality and finally the behavioral model describes the component behavior. Composition of components is done in the design phase by direct method calls.

IEC 61131 [33] is a standard for the design of Programmable Logic Controllers approved by the International Electrotechnical Commission (IEC). In this standard, the software units are called function blocks and based on incoming events, they execute some algorithms to update the internal variables. This standard has been further extended to IEC 61499 [34] which provides distribution in the runtime environment through high-level abstraction of communication primitives. IEC 61499 is an open communication standard for distributed control systems

JB (Java Beans) [35] developed by Sun Microsystems is based on Java programming language. In the JavaBeans specification a bean is a reusable software component that can be visually composed into applets, applications, servlets, and composite components, using visual application builder tools. Programming a

Java component requires definition of three sets of data: i) properties (similar to the attributes of a class); ii) methods; and iii) events which are an alternative to method invocation for sending data. JavaBeans was primarily designed for the construction of graphical user interface. The model defines three types of interaction points, referred to as ports: (i) methods, as in Java, (ii) properties, used to parameterize the component at composition time, (iii) event sources, and event sinks (called listeners) for event-based communication.

COM (Microsoft Component Object Model) [36] is one of the most commonly used software component models for desktop and server side applications. A key principle of COM is that interfaces are specified separately from both the components that implement them and those that use them. COM defines a dialect of the Interface Definition Language (IDL) that is used to specify object-oriented interfaces. Interfaces are object-oriented in the sense that their operations are to be implemented by a class and passed a reference to a particular instance of that class when invoked. A concept known as interface navigation makes it possible for the user to obtain a pointer to every interface supported by the object. This is based on VTable. Although COM is primarily used as a general-purpose component model it has been ported for development of embedded software and extended for distributed information systems

OpenCOM [37] is a lightweight component model developed at Lancaster University which aims at exploiting component-based techniques within middleware platforms. It is built atop a subset of Microsoft's COM. These include the binary level interoperability standard, Microsoft's IDL, COM's globally unique identifiers and the IUnknown interface. The higher-level features of COM such as distribution, persistence, transactions and security are not used. The key concepts of OpenCOM are capsules, components, interfaces, receptacles and connections. Capsules are runtime containers and they host components. Each component implements a set of custom receptacles and interfaces. A receptacle describes a unit of service requirement, an interface expresses a unit of service provision, and a connection is the binding between an interface and a receptacle of the same type.

OSGi (Open Services Gateway Initiative) [38] is a consortium of numerous industrial partners working together to define a service-oriented framework with an "open specifications for the delivery of multiple services over wide area networks to local networks and devices". Contrary to most component definitions, OSGi emphasizes the distinction between a unit of composition and a unit of deployment in calling a component respectively service or bundle. It offers also, at contrary to most component models, a flexible architecture of systems that can dynamically evolve during execution time. This implies that in the system, any components can be added, removed or modified at run-time. In relying on Java, OSGi is platform independent. There exists several additions of OSGi that

provides additional characteristics.

Palladio Component Model [39], developed at University of Oldenburg and University of Karlsruhe, provides a domain specific modeling language for component-based software architectures, which is tuned to enable early life-cycle performance predictions. Palladio defines its own metamodel specified in EMF/Ecore and divided into several domain specific languages for each developer role (i.e. component developers, software architects, system deployers and domain experts). All specifications can be combined to derive a full Palladio component model instance. As a starting point for implementing the system's business logic, the instance can be converted into Java code skeletons via Model2Text transformation. Components are specified via provided and required interfaces which consist of a list of service signatures. In order to allow accurate performance prediction, a so called resource demanding service effect specification can be added to each provided service to describe the sequence of called required services, resource usage, transition probabilities, loop iteration numbers, and parameter dependencies. Components and their roles can be connected via assembly connectors to build an assembly.

Pecos [40] is a joined project between ABB Corporate Research and Bern University. Its goal is to provide an environment that supports specification, composition, configuration checking and deployment for reactive embedded systems built from software components. There are two types of components, leaf components and composite components. The inputs and outputs of a component are represented as ports. At design phase composite components are made by linking their ports with connectors. Pecos targets C++ or Java as implementation language, so the run-time environment in the deployment phase is the one for Java or C++. Pecos enables specification of EFPs such as timing and memory usage in order to investigate in prediction of the behaviour of embedded systems.

Pin [41] component model developed at Carnegie Mellon Software Engineering Institute (SEI) is used as a basis in prediction-enabled component technologies (PECTs). By using principles from PECT it aims at achieving predictability by construction i.e. constraining the design and the implementation to analyzable patterns. To achieve predictability of a particular property PECT proposes a building of a reasoning framework that includes a component technology powered by analytical interface used for a specification of a property of interest and analysis theory used in provision of the system property composed from component properties. Accordingly, in order to perform analysis, proper analysis theories must be found and implemented in a suitable underlying component technology. PECT currently supports three reasoning frameworks from Pin Component model: λ_{ABA} - for predicting average latency in assemblies with periodic tasks, λ_{SS} - for predicting average latency in stochastic tasks managed by a sporadic server and ComFoRT -

for formal verification of temporal safety and liveness. Pin Components are defined in an ADL-like language, in the “component and connector style”, so called Construction and Composition Language (CCL). Pin components are fully encapsulated, so the only communication channels from a component to its environment and back are sink and source pins. Composition of components is obtained by connecting source and sink pins and the behavior of the interaction, which is specified as executable state machines.

ProCom [42] is a component model for control-intensive distributed embedded systems being developed at PROGRESS Strategic Research Center at Mälardalen University, Sweden. ProCom consists of two layers, in order to address different concerns that exist at different levels of a distributed embedded system. The upper layer, ProSys, focuses on modeling of the whole system or large subsystems. It considers complex active subsystems as components and captures the message flow between them. The lower layer, ProSave, serves for modeling of ProSys components on a detailed level. It explicitly captures the data transfer and control-flow between the components using a rich set of connectors which makes a platform for modeling control loops in a way that allows them to be easily analyzed and synthesized. The analysis is facilitated by the explicit control-flow and by the abstraction provided by components (read-execute-write semantics, encapsulation). The model provides support for different types of analysis by making possible to attach various models (behaviour, timing, resource utilization, etc.) to different architectural elements such as components, connections, subsystems, etc. Further, it considers deployment as a specific activity which includes components allocations, transformation of components to the entities complied with the execution model, and synthesis, i.e. creation of a glue code.

Robocop [43] is a component model developed by the consortium of the Robocop ITEA project, inspired by COM, CORBA and Koala component models. It aims at covering all the aspects of the component-based development process for the high-volume consumer device domain. Robocop component is a set of possibly related models and each model provides particular type of information about the component. The functional model describes the functionality of the component, whereas the extra-functional models include modeling of timeliness, reliability, safety, security, and memory consumption. Robocop components offer functionality through a set of ‘services’ and each service may define several interfaces. Interface definitions are specified in a Robocop Interface Definition Language (RIDL). The components can be composed of several models, and a composition of components is called an application. The Robocop component model is a major source of for ISO standard ISO/IEC 23004-1:2007 Information technology - Multimedia Middleware.

Rubus [44] component was developed as a joint project between Arcticus Systems AB and Mälardalen

University. The Rubus component model runs on top of the Rubus real-time operating system. It focuses on the real-time properties and is intended for small resource constrained embedded systems. Components are implemented as C functions performed as tasks. A component specifies a set of input and output ports, persistent states, timing requirements such as release-time, deadline. Components can be combined to form a larger component which is a logical composition of one or more components.

SaveCCM [45], developed within the SAVE project by several Swedish universities, is a component model specifically designed for embedded control applications in the automotive domain with the main objective of providing predictable vehicular systems. SaveCCM is a simple model that constrains the flexibility of the system in order to improve the analysability of the dependability and of the real-time properties. The model takes into consideration the resource usage, and provides a lightweight run-time framework. For component and system specification SaveCCM uses “SaveCCM language” which is based on a textual XML-syntax and on a subset of UML2.0 component diagrams.

SOFA (Software Appliances) [46] is a component model developed at Charles University in Prague. A SOFA component is specified by its frame and architecture. The frame can be viewed as a black box and it defines the provided and required interfaces and its properties. However a framework can also be an assembly of components in a composite component. The architecture is defined a grey-box view of a component, as it describes the structure of a component until the first level of nesting in the component hierarchy. SOFA components and systems are specified by an ADL-like language, Component Description Language (CDL). The resulting CDL is compiled by a SOFA CDL compiler to their implementation in a programming language C++ or Java. SOFA components can be composed by method calls through connectors. The SOFA 2.0 component model is an extension of the SOFA component model with several new services: dynamic reconfiguration, control interfaces and multiple communication styles between the components.

ACKNOWLEDGMENT

This work was partially supported by the Swedish Foundation for Strategic Research via the strategic research centre PROGRESS. We would also like to thank a number of researchers and practitioners, in particular Colin Atkinson (Mannheim University), Jan Carlson (Mälardalen University), Kung-Kiu Lau (Manchester University), Herman Martin (Robert Bosch), Ralf Reussner (Karlsruhe University), Heinz Schmidt (RMIT University), Christian Zeidler (ABB Research) for their valuable comments, and several anonymous reviewers who have read previous versions of this paper and provided constructive output.

REFERENCES

- [1] C. Szyperski, *Component Software*, Addison-Wesley Professional; 2002
- [2] G. T. Heineman and W. T. Councill, *Component-based Software Engineering: Putting the Pieces Together*, Addison-Wesley, 2001.
- [3] M. Chaudorn and Ivica Crnkovic, Component-based software engineering, *chapter 18 in H. van Vliet, Software Engineering: Principles and Practice*, Wiley, 2008
- [4] N. Medvidovic, E. M. Dashofy, R. N. Taylor, Moving architectural description from under the technology lamppost, *Information and Software Technology*, vol. 49, Iss.1, 2007
- [5] N. Medvidovic and R. N. Taylor, A Classification and Comparison Framework for Software Architecture Description Languages, *IEEE Transactions on Software Engineering*, Vol. 26, No. 1, January, 2000
- [6] I. Crnkovic, M. Chaudron, S. Larsson Component-based Development Process and Component Lifecycle, *Journal of Computing and Information Technology*, vol 13, nr 4, 2005
- [7] I. Crnkovic, M. Larsson, *Building Reliable Component-Based Software Systems*, Artech House, 2002
- [8] Oxford Advanced Learner's Dictionary, Available http://www.oup.com/oald-bin/web_getald7index1a.pl
- [9] The Object Management Group, UML Superstructure Specification v2.1, April 2006, Available at <http://www.omg.org/docs/ptc/06-04-02.pdf>
- [10] A. Beugnard, J.-M. Jézéquel, and N. Plouzeau. Making components contract aware, *IEEE Computer*, 32(7):38-45, 1999.
- [11] I. Crnkovic, M. Larsson, O. Preiss, Concerning Predictability in Dependable Component-Based Systems: Classification of Quality Attributes, *Architecting Dependable Systems III*, p pp. 257 – 278, Springer, LNCS 3549, 2005
- [12] G. Kotonya, I. Sommerville and S. Hall, Towards A Classification Model for Component-Based Software Engineering Research, *Proc. of the IEEE 29th EUROMICRO Conference*, September 2003
- [13] Kung-Kiu Lau and Zheng Wang, Software Component Models, *IEEE Transaction on Software Engineering*, Vol. 33, No. 10, October 2007
- [14] Stephen J. Mellor, Marc J. Balcer, *Executable UML: A Foundation for Model-Driven Architecture*, Addison-Wesley Professional, 2002
- [15] Hongyu Pei Breivold, Magnus Larsson, Component-Based and Service-Oriented Software Engineering: Key Concepts and Principles. *Proc. of the 33rd IEEE EUROMICRO conference, SEAA 2007*, pages, 13-20
- [16] H. J. Reekie, S. Neuendorffer, C. Hylands, and E. A. Lee. Software practice in the Ptolemy project. *Technical Report GSRC-TR-1999-01*, Gigascale Silicon Research Center, April 1999.
- [17] Sherif Yacoub, Hany Ammar, Ali Mili, A Model for Classifying Component Interfaces, *Second International Workshop on Component-Based Software Engineering, May 17th-18th, 1999*, Los Angeles, CA, USA
- [18] Sherif Yacoub, Hany Ammar, and Ali Mili, Characterizing a Software Component, *Second International Workshop on Component-Based Software Engineering, May 17th-18th, 1999*, Los Angeles, CA, USA
- [19] Klement J. Fellner and Klaus Turowski, Classification Framework for Business Components, *Proc. of the 33rd Hawaii International Conference on System Sciences*, 2000
- [20] AUTOSAR Development Partnership, AUTOSAR – Technical Overview v2.0.1, 27/06/2006, Available at http://www.autosar.org/download/AUTOSAR_TechnicalOverview.pdf
- [21] Ananda Basu, Marius Bozga and Joseph Sifakis, Modeling Heterogeneous Real-time Components in BIP, *Proc. of the 4th IEEE International Conference on Software Engineering and Formal Methods (SEFM06)*, Invited talk, September 11-15, 2006, Pune, pp 3-1
- [22] M. Bozga, S. Graf, I. Ober, I. Ober, and J. Sifakis, The IF toolset, in *SFM-RT 2004: international school on formal methods for the design of computer, communication and software systems*, Bertinoro, Italy 2004, vol. 3185, pp. 237-267, [Note(s) : VI, 293
- [23] G. Gößler, PROMETHEUS – a compositional modelling tool for real-time systems, *Proc. Workshop RT-TOOLS 2001, Technical report 2001-014*, Uppsala University
- [24] Bernhard Weichel, Martin Herrmann, A Backbone in Automotive Software Development Based on XML and ASAM/MSR, *SAE World Congress 2004*, Nr. 2004-01-295
- [25] Ji Eun Kim, Rahul Kapoor, Martin Herrmann, Jochen Haerdlein, Franz Grzeschniok, Peter Lutz, Software Behavior Description of Real-Time Embedded Systems in Component Based Software Development, *11th IEEE Symposium on Object Oriented Real-Time Distributed Computing (ISORC)*, 2008
- [26] X. Ke, K.Sierszecki and C. Angelov COMDES-II: A Component-Based Framework for Generative Development of Distributed Real-Time Control Systems, *13th IEEE International Conference on Embedded and Real-Time Computing Systems and Applications (RTCSA 2007)*
- [27] Rémi Bastide, Eric Barboni, Amélie Schyn, “Component-Based Behavioural Modelling with High-Level Petri Nets”, *Third Workshop on Modelling of Objects, Components and Agents (MOCA'04)*, Aarhus, Denmark, 2004
- [28] OMG CORBA v4.0, Available <http://www.omg.org/docs/formal/06-04-01.pdf>
- [29] EJB 3.0 Expert Group, JSR 220: Enterprise JavaBeans™, Version 3.0 EJB Core Contracts and Requirements Version 3.0, Final Release, May 2, 2006.
- [30] E. Bruneton, T. Coupaye & J.B. Stefani, The Fractal Component Model, February 5, 2004. Available <http://fractal.objectweb.org/specification/index.html>
- [31] R. van Ommering, F. van der Linden, and J. Kramer. “The koala component model for consumer electronics software”, *IEEE Computer*, pages 78–85. IEEE, March 2000.
- [32] C. Atkinson, J. Bayer, C. Bunse, E. Kamsties, O. Laitenberger, R. Laqua, D. Muthig, B. Paech, J. Wüst and J. Zettel, *Component-based Product Line Engineering with UML*, Addison-Wesley, 2002
- [33] IEC, *Application and Implementation of IEC 61131-3*, IEC, 1995.
- [34] IEC, IEC 61499 Function Blocks for Embedded and Distributed Control Systems Design, IEC, 2005
- [35] JavaBeans specification, Sun Microsystems, 1997, Available <http://java.sun.com/javase/technologies/desktop/javabeans/docs/spec.html>
- [36] D. Box, *Essential COM*, Addison-Wesley Professional, 1997
- [37] M. Clarke, G. Coulson, G. Blair and N. Parlavantzas “An Efficient Component Model for the Construction of Adaptive Middleware”. *Proc. of Middleware 2001*
- [38] OSGi Alliance, 15/02/2007, Available <http://www.osgi.org/>
- [39] S. Becker, H. Koziolok and R. Reussner, Model-Based Performance Prediction with the Palladio Component Model, *Proc. of the 6th International Workshop on Software and Performance*, Buenos Aires, Argentina, 2007
- [40] M. Winter, C. Zeidler, C. Stich, “The PECOS Software Process”, *Workshop on Component-based Software Development Processes*, ICSR 7 2002.
- [41] S. Hissam, J. Ivers, D. Plakosh, K. Wallnau, Pin Component Technology (V1.0) and Its C Interface. CMU Technical Report, CMU/SEI-2005-TN-001
- [42] Séverine Sentilles, Aneta Vulgarakis, Tomáš Bureš, Jan Carlson, Ivica Crnković, A Component Model for Control-Intensive Distributed Embedded Systems, *Proc. of 11th Symposium on Component-based Software Engineering (CBSE)*, LNCS Springer 2008, vol. 5282.

- [43] H. Maaskant; *A Robust Component Model for Consumer Electronic Products*, Philips Research Book Series Volume3, p167-192
- [44] Arcticus Systems, Available Rubus component model, <http://www.arcticus-systems.com>
- [45] M. Åkerholm et al., The SAVE approach to component-based development of vehicular systems, *Journal of Systems and Software*, Elsevier, May, 2006
- [46] T. Bureš, P. Hnětynkal and F. Plášil, SOFA 2.0: Balancing Advanced Features in a Hierarchical Component Model, *Proc. of SERA 2006, Seattle, USA, IEEE CS, Aug 2006*

Ivica Crnkovic received M.Sc. ('81) and Ph.D. ('98) in computer science in 1991, and M.Sc. in Theoretical Physics ('84) all from the University of Zagreb, Croatia. After 15 years of work in industry, he moved to academia 1999. He is a professor of industrial software engineering and chair of Software Engineering Division at Mälardalen University, Sweden. He is a co-author of two books, and co-editor of five books, the co-author of more than 100 refereed articles and papers on software engineering topics. He has co-organized several conferences and workshops and related to software engineering. His research interests include component-based software engineering, software architecture, software configuration management, software development environments and tools, as well as software engineering in general.

Séverine Sentilles received a M.Sc. ('06) degree in Computer Science from the "Université de Pau et des pays de l'Adour" in France and she is currently PhD student at Mälardalen University in Sweden. Her main research interests are centered on Component-based Software Engineering, Model-Driven Engineering and Model-Driven Engineering software development environments and tools.

Aneta Vulgarakis received a M.Sc. in July 2006 at the Faculty of Electrical Engineering, Skopje (Macedonia) with professional specialization in Computer Science, Information Technology and Automation. Her main research interest is Component-based Software Engineering, formal models and verification techniques for constructing correct and predictable real-time embedded systems.

Michel R.V. Chaudron received the M.Sc. ('92) and Ph.D. ('98) degrees in Computer Science from Leiden University in The Netherlands. He spent a couple of years working in IT-industry after which he worked at the TU Eindhoven. Currently, he is associate professor at the Leiden Institute of Advanced Computer Science where he co-heads the ICT in Business program. His main research interests are: Software Architecture, Component-based Software Engineering, UML, and empirical research in software engineering. He has published more than 80 refereed papers in these areas and is an active member of conferences in these areas.