A Classification Framework for Component Models

Ivica Crnkovic
Mälardalen University, Department of Computer Science and Electronics
Box 883, SE-721 23 Västerås, Sweden
ivica.crnkovic@mdh.se

Michel Chaudron
Technical University Eindhoven, Dept. of Mathematics and Computing Science,
P.O. Box 513, 5600 MB Eindhoven, The Netherlands
m.r.v.chaudron@TUE.nl

Séverine Sentilles
Mälardalen University, Department of Computer Science and Electronics
Box 883, SE-721 23 Västerås, Sweden
severine.sentilles@mdh.se

Aneta Vulgarakis
Mälardalen University, Department of Computer Science and Electronics
Box 883, SE-721 23 Västerås, Sweden
aneta.vulgarakis@mdh.se

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 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 some similarities, but also principal differences, and in many 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 quantifies basic principles of component models. Further, the paper classifies a certain number of component models using this framework.

Categories and Subject Descriptors

D.2.2 Design Tools and Techniques

General Terms

Design, component-based software engineering.

Keywords

Component models, taxonomy.

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, and Software Architecture in which a system is seen as a structure with clearly identified components and connectors. The techniques and technologies that form the basis for component models originate mostly from object-oriented programming and Architecture Description 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 that 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 the most commonly used definition coming from Szyperski [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 a software component. In spite of its generally 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 Councill [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 butthe component model itself is not specified. This definition of a component can be even more pushed further in the generalization, but on the contrary the definition of a component model can be expressed more precisely [3]:

Definition I: A *Software Component* is a software building block that conforms to a component model.

Definition II: 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; there exist many component models with many different characteristics on the market and in different research communities. This diversity makes it more difficult to properly understand the Component-Based (CB) principles, and to properly select a component model of interest, or to compare models. 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 and principles but very different implementations. For this reason there is a need for providing 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 this paper, we thus 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 of the framework - for certain component models we will say that they are similar in one dimension, but different in another. Several different taxonomies of component models already exist, an example is [6] in which taxonomy is described in respect to component life cycles in particular composition is presented, another example is [7] in which domains, business perspectives are shown as well as a focus on CBSE concepts such as reuse. Our comparison framework has the goal to provide a multidimensional framework, that counts different, yet equality important aspects of component models.

The remainder of this paper is as follows. Section two motivates, explains and defines the different dimensions of the classification framework. Section three gives a very brief overview of selected component models, and section four provides a short description of component model characteristics in the comparison framework, for each dimension.

2. The Classification Framework

The main concern of a component model is to (i) provide the rules for the specification of component properties and (ii) provide the rules and mechanisms for the 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. The fundamental principles can be divided into the following categories:

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 that point, 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.* Interface specification is the characteristic “sine qua non” of a component model. In some component models, the interface comprises the specification of all component properties, but in most cases, it only includes a specification of properties through which the communication with the environment should be realized. 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 themselves.

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

In these four dimensions, we comprise the main characteristics of component models but, of course, there are 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 that are not visible in component models themselves.

2.1 Lifecycle

In the software development lifecycle, a number of methods and technologies specifying and supporting particular phases of the cycle exist. While CBSE aims at covering the entire lifecycle of component-based systems, component models provide only partial lifecycle support and are usually related to 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, and similar).

The component-based paradigm (i.e. composability and reusability) has extended the integration activities in the run-time phase; certain component technologies provide extended support for dynamic and independent deployment of components into the systems. This support reflects the design of many component models. Accordingly, some of the components are only available at development stage and at run-time the system is monolithic. However not all component models consider the integration phase. We can clearly distinguish different component models that are related to a particular phase and such phase can be different for different component models. Some component technologies start in the design stage (e.g. Koala which has an explicit and dedicated design notation). Many other component technologies focus on the implementation phase (e.g. COM, EJB). For this reason one important dimension of the component model classification is the lifecycle support dimension. In such classification, we must consider both component lifecycle and component-based system lifecycle, which are somewhat different [3, 9] and are not necessary temporally related – they are ongoing in parallel and have some synchronization points. Here we identify characteristic “points” of both lifecycles that are concerns in component models:

(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).

(ii) Implementation stage. The component model provides support for generating and maintaining code. The implementation can

stop with the provision of the source code, or can continue up to the generation of a binary (executable) code. The existence of executable code is an assumption for the dynamic deployment of the components (i.e. the deployment of the components in the system run-time).

(iii) Packaging stage. Since components can be developed separately from systems, and the primary idea of the component-based approach is to reuse existing components, 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 compiled code modules that contain implementations of a component interface. Accordingly, the result of this stage can be a file, archive, or a repository where the packaged components are residing 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, in which a directory exists for each component. The directory contains a Component Description Language (CDL) file and a set of C and header files. Additionally, it can also contain interface definition files and/or data definition files. 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 i.e. bound to the execution platform. This activity happens at different points of development or maintenance phase. However, each of the component technologies that exist today solves the deployment issues in their own particular way. In general, the components can be deployed at compilation time (static binding) as part of the system, making it no longer possible to change how the components interact with each other, or at run time as separate units by using means such as registers (COM) or containers (CCM, EJB). For instance, Koala components are deployed at compilation time and they use static binding by following naming conventions and generated renaming macros. In opposition, CORBA components are deployed at run time in a container by using the information of the deployment descriptor packed with the component implementation.

2.2 Constructs

As mentioned in [30], the verb "construct" means "to form something by putting different things together", so in applying this definition to the CBSE domain, we define under this "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 is at contrary to functional – and sometimes extra-functional – properties of a component which are clearly stated in component interfaces. Consequently, a component interface has a double role: it first specifies the component properties (functional and possibly extra-functional), and second, it defines the actions through which components may be interconnected. Some of the component models distinguish also the "provides"-part (i.e. the specification of the functions that the component offer) from "requires"-part (i.e. the specification of the functions the component require) of an interface.

Besides coming along with the massive emergence of component models, several languages exist nowadays for specifying an interface: modelling languages (such as UML or different ADLs), particular specification languages (Interface Definition Languages), programming languages (such as interfaces in Java) or some additions built directly in a programming language. Similarly, the interaction can also be of different types: port-based where ports are the channels for communication of different data types and events; functions in programming languages defining input and output parameters; interfaces or classes in Object Oriented programming languages.

However, an interface remains most of time a very succinct description of the services a component proposes or needs. So in order to ensure that a component will behave as expected according to its specification and operational mode, 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 "client-component" must refer to the proper types, fields, methods, signals, ports and handles the exceptions raised by the "server-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 proper ranges. 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. dynamics 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 but one typical solution is directly related to the existence of interfaces and therefore to our constructs dimension. This solution offers the ability to add new interfaces to a component which makes possible to embody several versions or variants of functions in the component.

Besides, compositions in constructs are implemented as connections of interaction channels and the process of connecting is called binding. As mentioned before, the binding mechanismis related to the component lifecycle; it can occur at compilation time (when a compiler provides a direct connection between components using programming language mechanisms), or at run-time, in which the connection mechanisms are utilised either by the services of the underlying operation system, or are implemented in the component middleware or the component framework. A so-called "docking interface" method is utilized when the binding is provided at the run-time. This interface does not offer any application functionality, but serves instead for interaction between a component and the underlying system.

Another type of binding is also realised through connectors. 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 composition, in regards to direct or endogenous composition), (ii) introducing additional functionality. Exogenous composition enables more seamless evolution since it allows independent changes of components. In addition, in several component technologies, connectors act as specialised components, such as adaptors or proxies, either to provide additional functional or extra-functional properties, or to extend the means of intercommunication.

The interface specification implicitly defines the type of interaction between components to comply with particular architectural styles. In most cases, particular component models provide a single basic interaction mechanism, but others, such as Fractal for example, allow the construction of different architectural styles.

For the constructs dimension of this classification framework, we distinguish consequently the following points.

(i) *Interface specification,* in which different characteristics allowing the specification of interfaces are identified:
(1) The distinction between the notions of interface and port. Although a port is generally seen as a part of an interface, in some component models a port is actually the only mean of communication. In these cases, the binding is done in a wiring manner such as in the pipe and filter architectural style. On the contrary, interfaces may involve many different ways of binding..
(2) The distinction between the provides-part and requires-part of an interface.
(3) The existence of some distinctive features appearing only in this component model. And,
(4) The language used to specify those interfaces.

(ii) *Interface levels* which describe the levels of contractualisation of the interfaces, namely syntactic, semantic and/or behaviour level

(iii) *Standard Architectural Style* which aims at identifying the recurrent patterns of interaction among components. Some of them are for example pipe&filter, client/server or Event-driven.

(iv) *Communication type* which details mainly if the communication used is synchronous and/or asynchronous. An extension of this could be to consider also the number of receivers (unicast, multicast or broadcast).

(v) *Binding type* describes the way components may be linked together through the interfaces.
(1) The exogenous sub-category depicts if the component model allows using some connectors. And,
(2) The vertical sub-category expressing the possibility of having a hierarchical composition of components
We assume here that the "endogenous" composition and the "horizontal" binding are the default mechanism of any component model, i.e. a "direct" connection between two components.

2.3 Extra-Functional Properties

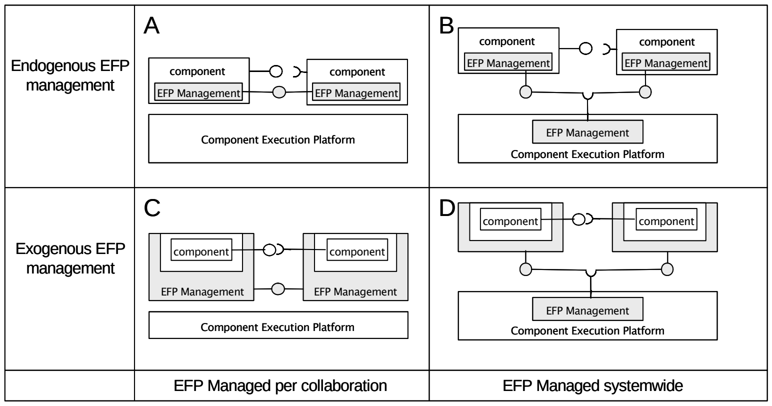
Properties (also designated as attributes) are used in the most general sense as defined by standard dictionaries, e.g.: "a construct whereby objects and individuals can be distinguished" [11]. There is no unique taxonomy of properties, and consequently there can exist many property classification frameworks. One commonly used classification is to distinguish functional from extra-functional properties. While functional properties describe functions or services of an object (individual or thing), extra-functional properties (EFP) specify the quality (in a broader sense) of objects. In CBSE, there is a distinction between component properties and system properties. The system properties can be the result of the composition of the same properties of components, but also of a composition of different properties [12]. Important concerns of CBSE are how to provide relevant parameters from components which will be used in a provision of the system properties.

The two main dimensions in which component models differ in the way they manage EFP are the following:

– A property is managed by the system (exogenous EFP management) or managed by components (endogenous EFP management). This corresponds to wonder which actor manages a property;

– A property is managed on a system-wide scale or the property is managed on a per-collaboration basis (i.e. what is the scope of management of a property).

The different types of approaches are characterized by the reference architectures shown in Figure 1


Endogenous EFP management
Exogenous EFP management
A
B
C
D
component
EFP Management
Component Execution Platform
EFP Managed per collaboration
EFP Managed systemwide
Figure 1. Management of extra-functional properties

Many component models provide no specific facilities for managing extra-functional properties. The way a property is handled is left to the designers of the system, and as a result a property may not be managed at all (approach A). 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.

The compatibility of components can be improved if the component model provides standardized facilities for managing EFP (approach B in Figure 1). In this approach, there is a mechanism in the component execution platform that contains policies for managing EFP for individual components as well as for EFP involving multiple components. The ability to negotiate the manner in which EFP are handled requires that the components themselves have some knowledge about how the EFP affects their functioning. This is a form of reflection.

A third approach is that the components should be 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 EFP. 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 EFP on a system wide scale (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 required to integrate them into systems that may employ different policies for EFP. Since these components do not address EFP, another advantage is that they are simpler and smaller and hence they are cheaper to implement.

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

(i) *Extra-functional properties support*: does the component model provide general principles, means and/or support for specification and reasoning about extra-functional properties?

(ii) *Extra-functional properties specification*: Does the component model contain means for specification and reasoning of specific extra-functional properties. If yes, which types and/or which properties?

(iii) *Composability of extra-functional properties*: Does the component model provide means, methods and/or techniques for composition of certain extra-functional properties. If yes, which 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 automotive. In such cases, requirements from the application domain penetrate into the component model. As a result, the component model provides a natural fit for systems in that particular domain. 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 production and the composition of components, but on the other hand, provide no guidance, nor support for any specific architecture. A general solution that enables component models to be both generally applicable but 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 modelling), while other parts are specific (for example different implementations).

3. SURVEY OF COMPONENT MODELS

Nowadays there are numerous component models which can vary widely in many possible aspects: In usage, in support provided, in concerns, in complexity, in formal definitions and similar. In our classification of component models, the first question is whether a 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 are still not assumed as component models, via those that lack many elements of component models, but are still called component models, through to those which are assumed as being 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 Definition I and Definition II: A model that explicitly or implicitly identifies components and defines rules for specification of component properties and means of their composition can be classified as a component model.

In the next section, we provide a very brief overview of some component models and their main characteristics. The list is not complete, and can be increased by time. It should be understood as a provision of some characteristic examples, or examples of widely used component models in Software Engineering.

The AUTomotive Open System Architecture (AUTOSAR) [14], is the result of the partnership between several manufacturers and suppliers from the automotive field. It envisions the conception of an open standardized architecture aiming at improving the exchangeability of diverse elements between vehicle platforms, manufacturer's applications and supplier's solutions. Those objectives rely upon the utilisation of both a component-based approach for the application and standardized layered architecture. This allows separating the component-based application from the underlying platform. AUTOSAR support both the client-server and Sender-Receiver communication paradigms and each AUTOSAR Software Component instance from a vehicle platform is only assigned to one Electronic Control Unit (ECU). The AUTOSAR Software Components, as well as all the modules in an ECU, are implemented in C.

BIP [14] framework designed at Verimag 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. Moreover, BIP focuses more on component behaviours than others component models thanks to a three-layer structure of the components (Behaviour, Interaction, Priority); a component can be seen as a point in this three-dimensional space constituted by each layer. This also sets up the basis for a clear separation between behaviour and structure. 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 [16] and the PROMETHEUS tools [17] are also provided.

CORBA Component Model (CCM) [18] evolved from Corba object model and it was introduced as a basic model of the OMG's component specification i.e CORBA 3 in 2002. 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. Component is a new CORBA metatype. 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.

The Entreprise JavaBeans (EJB) [19], developed by Sun MicroSystems envisions the construction of object-oriented and distributed business applications in trying to hide to developers the underlying complexity, such as transactions, persistence, concurrency, interoperability. It also aims at the improvement of component reusability in providing different utilities, such as means, so called EJB-jars to package components, called beans. Three different types of components coexist to match the specific needs of different applications (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). In order to achieve this, EJB technology use the Java programming language.

Fractal [20] 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 comes up with 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 describes its own behaviour. 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. Consequently, nothing is fixed in Fractal; On the contrary, it even provides means to facilitate adaptation in notably having different implementations to fit the specific needs of a domain as for example its C-implementation called Think, which targets especially the embedded systems. A reference implementation, called Julia and written in Java, is also provided. Fractal can also be seen as a generic component model which intends to encompass other component models.

Koala [21] is a component model developed by Philips for building consumer electronics. Koala components are units of design, development and reuse. Semantically, components in Koala are defined in a ADL-like language. 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.

Microsoft Component Object Model (COM) [22] 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 embedded software development.

The Open Services Gateway Initiative (OSGi) [23] 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 emphasis the distinction between a unit a composition and a unit of deployment in calling a component respectively service or bundle. It offers also, 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. Thus, there is no guaranty that a service provided at a certain time will be still provided later. Being built on Java, OSGI is platform independent.

Pecos [24] is a joint project between ABB Corporate Research and academia. Their goal is to provide environment that supports specification, composition, configuration checking and deployment for reactive embedded systems built from software components. Component specification and component composition are done in an ADL-like language called CoCo. 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 EFP properties such as timing and memory usage in order to investigate in prediction of the behaviour of embedded systems.

Pin [25] component model is based on an earlier component technology developed by Carnegie Mellon Software Engineering Institute (SEI), for use in prediction-enabled component technologies (PECTs). It is aimed for building embedded software applications. By using principles from PECT it aims at achieving predictability by construction. Components are defined in an ADL-like language, in the "component and connector style", so called Construction and Composition Language (CCL). Furthermore, Pin components are fully encapsulated, so the only communication channels from a component to its environment and back are its pins.

Robocop [26] 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. A 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 modelling of timeliness, reliability, safety, security, memory consumption, etc. 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 for ISO standard ISO/IEC 23004 for multimedia middleware.

Rubus [27] component was developed as a joint project between Arcticus Systems AB and the Department of Computer Engineering at 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, behaviour and a persistent state, 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 [28], developed within the SAVE project and 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.

The SOFA (Software Appliances) [29] is 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, i.e a composite component. The architecture is defined as 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.

4. COMPONENT MODEL CLASSIFICATION

In order to illustrate the utilisation of our classification framework, we categorize here the component models listed above with respect to the corresponding dimensions. The reference documentation of each component models has generally been used to fill those tables. However, some of the information presented here are not mentioned explicitly in the reference documentation and are subject to the reader's point of view.

Table 1: Lifecycle Dimension

Component Models	Modelling	Implementation	Packaging	Deployment
AUTOSAR	N/A	C	N/A	At compilation
BIP	A 3-layered representation: statemachine diagram, priority and interaction expression or a statemachine with ports	BIP language	N/A	At compilation
CCM	Abtstract model:OMG-IDL, Programming model: CIDL	Language independent.	Deployment Unit archive (JARs,DLLs)	At run-time
Fractal	FractalGui, ADL-like language (Fractal ADL, Fractal IDL), Annotations (Fractlet)	Julia, Aokell(Java) Think(C/C++) FracNet(.Net) …	File system based repository	At run-time
KOALA	ADL-like language (IDL,CDL and DDL)	C	File system based repository	At compilation
EJB	N/A	Java, Java binary code	EJB-Jar files	At run-time

Component Models	Modelling	Implementation	Packaging	Deployment
MS COM	Microsoft IDL	Different languages, Binary standard	DLL	At run-time
OSGi	N/A	Java	Jar-files (bundles)	At run-time
PIN	ADL-like language (CCL)	C	DLL	At compilation
PECOS	ADL-like language (CoCo)	C++, Java	Jar packages	At compilation
ROBOCOP	IDL for the interface model. Several different models	C, C++	zip files	At compilation At run-time
RUBUS	N/A	C	File system based repository	At compilation
SaveCCM	ADL	C	N/A	At compilation
SOFA 2.0	Meta-model based definition	Java	Repository	At run-time

Table 2: Constructs

Component Models	Interface Specification				Interface Levels	Standard Architecture Styles	Communication Type	Binding Type	
	Interfaces/ Ports/Both	Distinction Provides / Requires	Distinctive feature	Interface Language				Exogenous	Vertical
AUTOSAR	Both	Yes	Classified within 3 types: – AUTOSAR Interface, – Standardized AUTOSAR Interface, – Standardized Interface	C header files	Syntactic No semantic No behaviour	Client/Server Data-centered	Synchronous Asynchronous	No	Delegation
BIP	Port	No	Existence of: Complete interfaces, Incomplete interfaces	BIP Language	No syntactic Semantic Behaviour	Event-driven	Synchronous Asynchronous (Rendez-vous, Broadcast)	No	Delegation
CCM	Both	Yes	Classified within 2 types: – Facets and receptacles – Event sinks and event sources	CORBA IDL	Syntactic No semantic No behaviour	Blackboard	Synchronous Asynchronous	No	No
EJB 3.0	Interface	No	N/A	Java Programming Language + Annotations	Syntactic No semantic No behaviour	Client/Server (JDBC) Blackboard (JMS)	Synchronous Asynchronous	No	No
Fractal	Interface	Yes	Existence of: Control Interface	Any programming language, IDL, Fractal ADL	Syntactic No semantic Behaviour	Multiple architectural styles	Multiple communication styles	Yes	Aggregation
KOALA	Interface	Yes	Existence of: – Diversity Interface, – Optional Interface	IDL	Syntactic No semantic No behaviour	Pipe&filter	Synchronous	Yes	Aggregation
MS COM	Interface	No	All interfaces derived from a IUnknown	Microsoft IDL	Syntactic No semantic No behaviour	Multiple architectural styles	Synchronous	No	Delegation Aggregation
OSGi	Interface	Yes	Existence of: Dynamic Interfaces	Java programming language	Syntactic No semantic No behaviour	Event-driven	Synchronous	No	No
PECOS	Port	Yes	N/A	Coco composition language	Syntactic Semantic Behaviour	Pipe&filter (with blackboard) Event-driven	Synchronous	No	Delegation
Pin	Port	Yes	N/A	Component Composition Language	Syntactic No semantic No behaviour	Pipe&filter Event-driven	Synchronous Asynchronous	Yes	No

Rubus	Port	Yes	Classified within 2 types: – data – triggered	C header file	Syntactic No semantic No behaviour	Pipe&filter	Synchronous	No	Delegation
Robocop	Port	Yes	N/A	Robocop IDL (RIDL)	Syntactic Semantic (Behaviour)	Client/Server	Synchronous Asynchronous	No	No
SaveCCM	Port	Yes	Classified within 3 types: – data – triggered – data& triggered	SaveComp (XML-based)	Syntactic No semantic Behaviour	Pipe&filter	Synchronous	No	Delegation
SOFA 2.0	Interface	Yes	Existence of: Utility Interface, Possibility to annotate interface and to control evolution	Java programming language	Syntactic No semantic Behaviour	Multiple architectural styles	Multiple communication styles	Yes	Delegation

Table 3: Extra-Functional Properties

Component Models	General support for properties	Properties specification	Composition support
AUTOSAR	N/A		
BIP	Behaviour compositions, endogenous EFP management	times properties	
CCM	Support for mechanism to influence of some EFP, exogenous EFP management	N/A	Run time support for some EFP
Fractal	Interceptor and controller in Julia, extension possibilities for different EFP, endogenous EFP management	Extension with a new controller	
KOALA	Extensions in interface specification and the compiler	Resource usage but no timing and memory consumption	Compile time checks
EJB	Support for mechanism to influence of some EFP, container maintaining EFP, exogenous EFP management	N/A	Run time support for some EFP
MS COM	Endogenous EFP management	N/A	N/A
OSGi	endogenous EFP management	N/A	N/A
PIN	Analytic interface for EFP, endogenous EFP management	Timing properties	Different EFP composition theories
PECOS	endogenous EFP management	Timing properties (WCET, periods), memory consumption	N/A
ROBOCOP	CEP provides manager for resource budgets, exogenous EFP management	Memory consumption, WCET, cycle time, priority, reliability	Some EFP checked at deployment and monitored during execution.
RUBUS	Exogenous EFP management	Timing, resource usage, QoS	Design time
SaveCCM	Interface extension endogenous EFP management	Real-time attributes	Composition of RT EFP
SOFA 2.0	Behaviour EFP specification, endogenous EFP management	Behavioural (protocols)	Composition at design

Table 4: Domains

Domain	AUTO-SAR	BIP	CCM	Fractal	KOALA	EJB	MS COM	OSGi	PIN	PECOS	ROBOCOP	RUBUS	SaveCCM	SOFA 2.0
Generative				x							x			x
General			x	x		x	x	x	x					x
Specialised	x	x			x			x		x	x	x	x	

5. CONCLUSION

In this short survey we have presented a framework for classification of component models. Such classification is not simple, since it does not cover all aspects of component models. It 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 client-server style, while in the specialized domains (mostly embedded systems) pipe & filter 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 EFP is still not widespread, there is an unlimited number of different EFP, and finally the compositions of many EFP are not only the results of component properties. Their compositions are considerably complex and (also) a matter of system architectures [12]. This taxonomy can be further analyzed and refined, which is our intention: on one side enlarge the list with the new component models, on other side refine the taxonomy by introducing some comparative values and by introducing subtypes of the points in the framework dimension.

6. 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. R. V. Chaudron, *Lecture notes on CBSE* Technische Universiteit Eindhoven, 2006

[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 Ri. N. Taylor, *A Classification and Comparison Framework for Software Architecture Description Languages*, IEEE Transactions on Software Engineering, Vol. 26, No. 1, January, 2000

[6] K.-K. Lau and Z. Wang. *A taxonomy of software component models*. In Proc. 31st Euromicro, Conference, pages 88–95. IEEE Computer Society Press, 2005.

[7] 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

[8] I. Crnkovic, M. Chaudron, S. Larsson *Component-based Development Process and Component Lifecycle,* Journal of Computing and Information Technology, vol 13, nr 4, 2005

[9] I. Crnkovic, M. Larsson, *Building Reliable Component-Based Software* Systems Artech House, 2002

[10] A. Beugnard, J.-M. Jézéquel, and N. Plouzeau. *Making components contract aware*. IEEE Computer, 32(7):38-45, 1999.

[11] Miller, G. A. (2002). *WordNet®*. Cognitive Science Laboratory, Princeton University Available: http://www.cogsci.princeton.edu/~wn/

[12] 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

[13] The Object Management Group, *UML Superstructure Specification* v2.1, April 2006. Available at http://www.omg.org/docs/ptc/06-04-02.pdf

[14] AUTOSAR Development Partnership, *AUTOSAR – Technical Overview v2.0.1*, 27/06/2006, Available at http://www.autosar.org/download/AUTOSAR_TechnicalOverview.pdf

[15] Ananda Basu, Marius Bozga and Joseph Sifakis, *Modeling Heterogeneous Real-time Components in BIP*, 4th IEEE International Conference on Software Engineering and Formal Methods (SEFM06), Invited talk, September 11-15, 2006, Pune, pp 3-1

[16] M. Bozga, S. Graf, I. Ober, I. Ober, and J. Sifakis, *The IF toolset*, in SFM, 2004.

[17] G. Gößler, *PROMETHEUS — a compositional modelling tool for real-time systems*, Proc. Workshop RT-TOOLS 2001, Technical report 2001-014, Uppsala University

[18] OMG CORBA v 4.0, http://www.omg.org/docs/formal/06-04-01.pdf

[19] EJB 3.0 Expert Group, JSR 220: Enterprise JavaBeansTM,Version 3.0 EJB Core Contracts and Requirements Version 3.0, Final Release, May 2, 2006.,

[20] E. Bruneton, T. Coupaye & J.B. Stefani, *The Fractal Component Model*, February 5, 2004. http://fractal.objectweb.org/specification/index.html

[21] R. van Ommering, F. van der Linden, and J. Kramer. *"The koala component model for consumer electronics software"*, In IEEE Computer, pages 78–85. IEEE, March 2000.

[22] D. Box, Essential COM, Addison-Wesley Professional, 1997

[23] OSGi Alliance, 15/02/2007, http://www.osgi.org/

[24] M. Winter, C. Zeidler, C. Stich, *"The PECOS Software Process"*, Workshop on Components-based Software Development Processes, ICSR 7 2002.

[25] 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

[26] H. Maaskant; "A Robust Component Model for Consumer Electronic Products", Philips Research Book Series Volume3, p167-192

[27] Arcticus Systems, *Rubus component model*, http://www.arcticus-systems.com

[28] M. Åkerholm et al., *The SAVE approach to component-based development of vehicular systems*, Journal of Systems and Software, Elsevier, May, 2006

[29] 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

[30] Oxford Advanced Learner's Dictionary, http://www.oup.com/oald-bin/web_getald7index1a.pl