A Classification Framework for Software Component Models

Ivica Crnković, *Member*, *IEEE*, Séverine Sentilles, *Student Member*, *IEEE*, Aneta Vulgarakis, *Student Member*, *IEEE*, and Michel R.V. Chaudron

Abstract—In the last decade, a large number of different software component models have been developed, with different aims and using different principles and technologies. This has resulted in a number of models which have many similarities, but also principal differences, and in many cases unclear concepts. Component-based development has not succeeded in providing standard principles, as has, for example, object-oriented development. In order to increase the understanding of the concepts and to differentiate component models more easily, this paper identifies, discusses, and characterizes fundamental principles of component models and provides a Component Model Classification Framework based on these principles. Further, the paper classifies a large number of component models using this framework.

Index Terms—Software components, software component models, component lifecycle, extra-functional properties, component composition.

1 Introduction

Component-based software engineering (CBSE) is an established area of software engineering. The techniques and technologies that form the basis for CBSE originate from object-oriented design, software architectures, and Architecture Definition Languages (ADLs), middleware, and certain older approaches such as structural and modular development. The inspiration for "building systems from components" comes from other engineering disciplines, such as civil or electrical engineering. However, because software is, in its nature, different from physical products, a direct translation of principles from the classical engineering disciplines into software engineering is not possible. 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 components. There has been much debate on the notion of software component, as, for example, in [1].

From the beginning, CBSE struggled with the problem of obtaining a common and sufficiently precise definition of the concept of software component. An early and commonly used definition from Szyperski [2] states that

"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."

In spite of its generality, this definition does not capture the entire realm of component-based approaches. The ongoing debates have led to another definition by Heineman and Councill [3]: *"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."*

Here, the emphasis has been shifted from component to *component model*[1] (a component model determines what is and what is not a component). Heineman and Councill [3] define a component model as follows: *"A component model defines a set of standards for component implementation, naming, interoperability, customization, composition, evolution, and deployment."*

This definition points out that a component model covers multiple facets of the development process, dealing with 1) rules for the construction of individual components and 2) rules for the assembly of these components into a system.

Currently, there exist many component models. Some component models target specific application domains, ranging from embedded systems (automotive software, consumer electronics) to particular business domains (finance, telecommunications, healthcare, and transportation). Other component models are based on certain technological platforms (Enterprise Java Beans, DCOM). All component models are based on some often implicit assumptions about the architecture of the types of systems they are targeting.

The architectural assumptions made by the component models often relate to the way in which system level properties (for example, real-time properties, safety, and low resource use) can be achieved through the composition

• I. Crnković, S. Sentilles, and A. Vulgarakis are with the School of Innovation, Design and Engineering, Mälardalen University, IDT, PO Box 883, Västerås S-721 23, Sweden.
E-mail: {ivica.crnkovic, severine.sentilles, aneta.vulgarakis}@mdh.se.
• M.R.V. Chaudron is with the Faculty of Science, Leiden Institute of Advanced Computer Science, Universiteit Leiden, PO Box 9512, RA 2300, Leiden, The Netherlands. E-mail: chaudron@liacs.nl.

Manuscript received 16 Aug. 2008; revised 31 Dec. 2009; accepted 15 June 2010; published online 7 Sept. 2010.
Recommended for acceptance by W. Emmerich.
For information on obtaining reprints of this article, please send e-mail to: tse@computer.org, and reference IEEECS Log Number TSE-2008-08-0248.
Digital Object Identifier no. 10.1109/TSE.2010.83.

1. In the remainder of the paper, we abbreviate the term "software component model" to "component model" and "software component" to "component."

of components. In some cases, component models take a strict approach and enforce rules that guarantee that a system level property is achieved. Other component models offer flexibility at the implementation level, but require the disciplined design of components for achieving system level properties. The fact that CBSE is applied to a large spectrum of application domains, each with their own set of architectural requirements, explains why many component models exist today. This large variety reinforces the difficulty in understanding the common principles of component models.

The diversity found in component models is similar to the one that exists in the area of ADLs: At a high level of abstraction, there are many similar mechanisms and principles, but there are also many variations and different implementations. For this reason, in a manner similar to what was done for ADLs [4], [5], we propose a framework which provides a classification and comparison between different component models. A classification framework can aid in the understanding of software component models through its explanation of their key principles. The framework can help in assessing the suitability of a component model for a particular set of requirements or for a particular application domain. Furthermore, the framework can help in the design of new component models through the list of important characteristics a component model must provide.

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

Our research methodology followed an empirical approach consisting of the successive iterations of the steps of: 1) observations and analysis, 2) classification, and 3) validation. The observations and analysis included the study of a number of component models and the literature related to the general principles of CBSE [2], [3], [4], [6], [7], [8], [9], and related classifications [5], [10], [11], [12]. In addition, we utilized our own experience gained from the development of the SAVE Components Component Model (SaveCCM) [13], PROGRESS Component Model (ProCom) [14], and Robust Open Component Based Software Architecture for Configurable Devices Project (Robocop) [15] component models, and our tight cooperation with industry that used some component technologies in their development (ABB (COM, Pin), Ericsson (Service-oriented architecture), Philips (Koala, Robocop), Volvo (AUTomotive Open System ARchitecture (AUTOSAR), Rubus), Arcticus (Rubus)). Based on this, our classification framework was built, incrementally populated, and refined with a set of component models. The validation consisted of trying to fit, at each iteration, a larger set of component models into the framework. Further validation was performed by discussing the framework with several CBSE experts from industry and academia and with researchers in the broader field of software engineering. For several component models, we contacted their developers and obtained feedback on the classification we proposed for "their" component models. The resulting analysis and discussions have also led to a refinement of the framework.

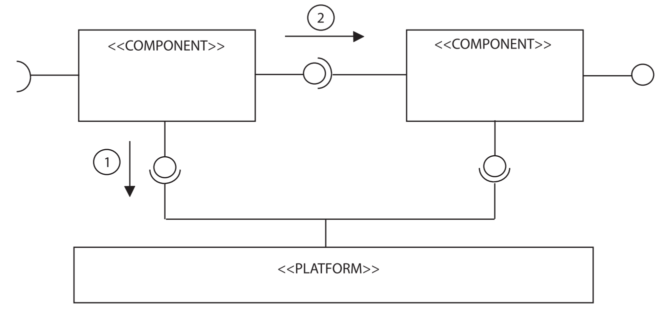

Fig. 1. A component-based system.

The remainder of this paper is organized as follows: Section 2 gives definitions and explanations of basic concepts used in CBSE. Section 3 defines, explains, and motivates the different dimensions of the classification framework. Section 4 discusses the criteria for inclusion of different component models into our component models survey and the classification framework. The comparison framework and observations from the comparison are presented in Section 5 and further analyzed in Section 6. Related work is covered in Section 7, and Section 8 concludes the paper. A brief overview of the selected component models on which the classification framework has been mapped is given in the Appendix.

2 Main Concepts of Component Models

The classification of component models requires an understanding of the main concepts and unique terminology used in CBSE. Therefore, we define the concepts related to the notion of component models.[2] The terminology defined in this section includes *component model*, *component-based system*, *component*, and *binding*. A component itself is defined relatively to a specific component model [6].

Definition. *A component model defines standards for 1) properties that individual components must satisfy and 2) methods for composing components.*

In this definition, the term "component properties" is meant to include functional and extra-functional specifications of individual components. The term "composing components" is meant to include mechanisms for component interaction. To explain these terms further, we start from an architectural specification of a component-based system.

A component-based system identifies 1) components, 2) an underlying platform, and 3) the binding mechanisms, as shown in Fig. 1, and presented formally as

$$\mathbb{CBS} = <\mathbb{P}, \mathbb{C}, \mathbb{B}>,$$

2. The focus of this paper is on component models. Therefore, the terms from the general domain of CBSE that are not specifically related to component models will not be discussed in this paper.

where

- $\mathbb{CBS} =$ component-based system;
- $\mathbb{P} =$ system platform;
- $\mathbb{C} =$ a set of components C_i;
- $\mathbb{B} =$ set of bindings B_i.

A component is executable.[3] In contrast to arbitrary executable code, a component is formed to interact with other components according to predefined rules. In other words, a component is a software module that includes both execution code and machine-readable metadata (typically including the interface signature) which explicitly describes the services that the software provides and the services that it requires from other components and its execution environment. The metadata support the component framework in composing a component with other components and in deploying it into an execution environment. In addition, the metadata can include information about extra-functional properties (EFPs) of components.

More formally, we specify a component C by a set of 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" [10]. There is no unique taxonomy of properties, and there exist different property classifications. One commonly used classification is to distinguish functional from extra-functional properties (also designated as nonfunctional, or Quality of Services, or "ilities"). While functional properties describe functions or services a component provides or requires, extra-functional properties describe its nonfunctional characteristics. Typical examples of extra-functional properties are quality attributes such as reliability and response time. A component C can expose its functional properties by the means of an interface I. Hence, we can characterize a component C by its functional interface I and by a set of extra-functional properties P:

$$C = \langle I, P \rangle,\ with\ I = \{i_1, i_2, ..i_n\};\quad P = \{p_1, p_2, ..p_k\}.$$

I defines a set of functional properties (services) i_k that a component provides or requires.

P defines a set of extra-functional properties p_i of the component.

If a component $C = \langle I, P \rangle$ complies with a component model CM, then this implies that its interface and its properties must comply with the rules of the component model. This is formally denoted as follows:

$$C \models CM \Rightarrow I, P \models CM.$$

Bindings define connections between interfaces. We distinguish bindings between 1) the components and the platform (which enable component integration into a system) from 2) bindings between components (which enable component interaction). In the first case, we talk about *component deployment* (denoted as ① in Fig. 1) and in the second about *component binding* (denoted as ②).

The components C_1 and C_2 bounded by their interfaces I_1 and I_2 construct an *assembly* $A = \{C_1, C_2\}$. If a component model includes assembly as an architectural element, then the assembly is specified by its interface I_A:

$$A = \{C_1, C_2\}, A = \langle I_A \rangle | I_A = \langle I_1 \oplus I_2 \rangle.$$

Note that an assembly is not necessarily a component itself; it is not necessary that it conform to the component model. If an assembly $C = \{C_1, C_2\}$ conforms to the component model, i.e.,

$$C = \langle I, P \rangle; I = \langle I_1 \oplus I_2 \rangle, C \models CM,$$

the assembly is a component, also called a composite component.

A composite component also exhibits a set of extra-functional properties. In the above example, the composite component is specified by $C = \langle I, P \rangle$, but we did not define P as a composition of component properties P_1 and P_2. We can state a question: Can P be defined as a composition of P_1 and P_2? As we will see later, the extra-functional properties of a composite component are, in most cases, not only the result of component property composition, but also of the external environment (e.g., underlying platform and other components). Formally, we express this as

$$C = \langle C_1 \oplus C_2 \ \rangle \ \Rightarrow I = \langle I_1 \oplus I_2 \rangle \ \wedge \ P_{ex} \vdash P = \langle P_1 \oplus P_2 \rangle,$$

where P_{ex} denotes a specification of the external (system) context that has an impact on the composition of component extra-functional properties. A more detailed discussion about binding and composition is presented in Section 3.2.2.

3 The Classification Framework

The rules a component model defines for the design and composition of components cover different principles and hide many complex implementation mechanisms. Furthermore, different component models cover different phases in the component lifecycle; while some support only the modeling and design stage, others mainly support the implementation and runtime stages. For this reason, we cannot simply list all possible component models characteristics, but we group the characteristics according to their similar concerns and aspects.

Starting from these premises, we divide the basic characteristics and principles of component models into the following three dimensions:

D1. **Lifecycle.** The lifecycle dimension identifies the support provided by a component model and the component forms throughout the lifecycle of components. CBSE is characterized by a separation of the development processes of individual components from the development process of the overall system. A component lifecycle covers stages from the component specification until its integration into the systems and possibly its execution and replacement.

D2. **Construction.** The construction dimension identifies principles and mechanisms for building systems from components including 1) the component functional specification (of which the *interface* is a prominent part), 2) the means of establishing connections between the components, i.e., *binding*,

3. Note that executable property does not necessarily mean binary code. For example, the execution can be achieved through an interpreter or by a virtual machine, or even through compilation before the execution.

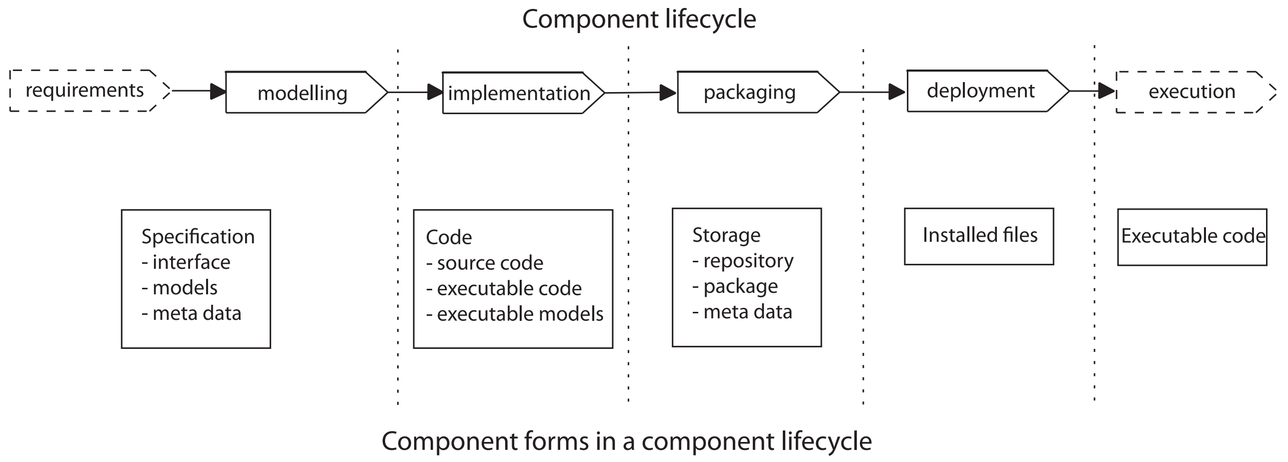

Fig. 2. Component lifecycle and component forms.

and the means of intercommunications, i.e., *interactions* between the components.

D3. **Extra-functional properties.** The extra-functional properties dimension identifies the facilities a component model offers for the specifications, management, and composition of extra-functional properties.

Below, we discuss these dimensions and introduce their features, i.e., the characteristics of component models.

3.1 Lifecycle

An important characteristic of CBSE is the separation of the development process of the overall system from the development processes of individual components [7]. These processes can be completely independent as, for example, in the development of Commercial Off-The-Shelf (COTS) components and COTS-based systems, up to the point where a component is integrated into a system.

The development of an individual component follows the following stages (see Fig. 2): requirements, design, implementation, deployment, and execution. During its lifecycle, a component has different forms [16]: Initially, a component is represented by a set of requirements, yet, during design, the same component is represented by a *set of models*. Subsequently, the same component is represented by means of *source code*, complemented by *metadata*. After deployment, the component is integrated in an execution environment. And at runtime, the same component[4] is now represented by *object code* of the target platform. Optionally, at intermediate stages, a component may be packaged and represented by means of a set of files in a directory or zip file. Fig. 2 shows these successive stages of a component's lifecycle. The lower half of the figure lists the ways in which components may be represented in that particular stage of the lifecycle. In the figure, the requirements and execution stages are depicted with dashed lines to indicate that in these stages, components do not necessarily exist as independent units.

Most component models provide support for several stages of the component's lifecycle. Support in the design stage may consist of a dedicated design notation or predefined approach for modeling different aspects of components. For example, the Koala component model [17] has an explicit design notation which includes representations for, among others, components, interfaces, and bindings. Other component models dictate the use of state machines for modeling the behavior of components. In the implementation stage, a component model typically defines which construction elements should be used for encoding a component in a programming language. Implementation-level rules typically include conventions for the naming and structuring of interfaces. The component models that cover several stages often provide a support for transformation between the different component forms; typical examples are transformations from models to code, such as interface specifications to stubs in programming languages. In some cases, the transformation rules can be quite complex, as, for example, in the domain of real-time systems in which the design units, the components, are transformed into executable units, the real-time tasks.

3.1.1 Component Lifecycle Stages

We identify the following stages of the component lifecycle:

L1. *Modeling stage.* Component models provide support for the modeling and the design of components and component-based systems. Models are used either for the architectural description of the systems, the components and the interaction between them (for example, using a standard or a dedicated ADL) or for the modeling and verification of particular system and component properties (using different modeling techniques such as statecharts or different variants of finite automata). For example, the Palladio [18] and KOmponenten-BasieRte Anwendungsentwicklung (KobrA) [19] component models use UML profiles with new or modified UML architectural elements and annotations, while ProCom [14] and Pin [20] have their own modeling languages.

L2. *Implementation stage.* Component models provide support for the production of code. Support for the implementation stage may stop with the provision of the source code or may continue up to the generation

4. Actually, an *instance* of this component.

of a binary (executable) code.[5] Most of the component models use standard programming languages. Some component models assume the use of a particular language for the implementation. In such cases, the component model may require that (elements of the) language be used according to some specific rules. For example, the Entreprise JavaBeans (EJB) component model [21] uses Java, with some extensions and additional requirements. Other component models explicitly aim to be language independent for the implementation. Such component models may have translators from their modeling and specification languages to a particular, or sometimes multiple, programming language(s) as for the CORBA Component Model (CCM) [22].

L3. *Packaging stage.* Because of the separation of the development processes in the component-based lifecycle, there is a need for the storage and packaging of components, either in a repository or for distribution. A component package is a set of metadata and code (source or executable). The metadata contain information about the contents of the files in the package. Accordingly, the result of this stage can be a file, an archive, or a repository in which the packaged components reside prior to their use. For example, in Koala [17], components are packed into a file system-based repository, with one folder per component. The folder includes a number of files: a Component Description Language (CDL) file and a set of C and header files, test file, and different documents. Another example of packaging is used in the EJB [21] component model. There, packaging is done through JAR archives, called EJB-JAR. Each archive contains an XML deployment descriptor, a component description, a component implementation, and interfaces.

L4. *Deployment stage.* At a certain point in time, a component is integrated into an executable system or some target environment and becomes ready for execution. This may happen at different stages in the system's lifecycle. In general, a component can be deployed at:

a. *Compilation time.* Components are integrated before the system starts executing. Compilation (and linking) achieves integration of components through the resolution of references to interface names. Binding at compilation time is typical for embedded systems in which the components and the execution platform are compiled and linked together into an executable image. This happens, for instance, in the Koala component model.

b. *Runtime.* Components may be added or replaced in a system which is executing. Runtime deployment may be realized by using a registry (COM [23]), or by containers which handle installation and communication of the component using information of the deployment descriptor packed with the component implementation (CCM [22], EJB [21]).

5. Considering the component model definition and Szyperski's definition of a component, it can appear strange that component models do not address the implementation stage. However, the component models specify characteristics of components that are executable units, although not necessarily the implementation rules themselves.

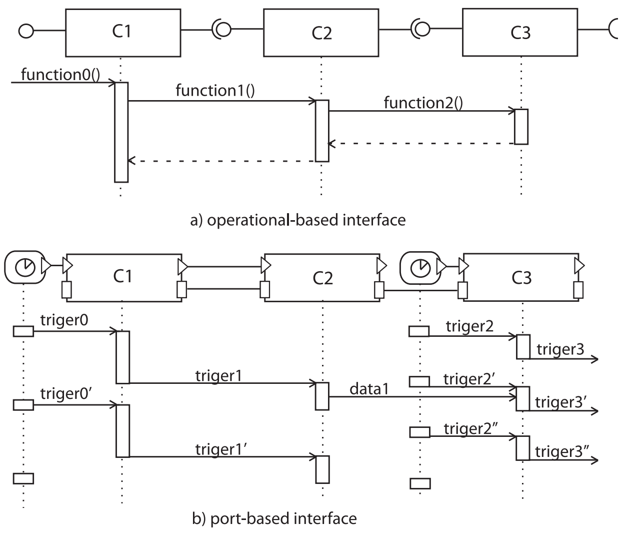

Fig. 3. Operation-based and port-based interfaces.

3.2 Construction

As defined in the *Oxford Advanced Learners Dictionary* [24], *construction* means "the process or method of building." The construction dimension of our classification includes three parts: 1) connection points, i.e., *interfaces*, 2) mechanisms for establishing connections, i.e., *binding* mechanisms, and 3) communication itself, i.e., *interaction*. The next section discusses each of these aspects in more detail, and provides a list of elements that characterize this dimension.

3.2.1 Interface

A component interface defines a set of actions which is understood by both the provider (the component) and user of that interface (other components, or other software). The actions of an interface can be characterized by a name and a list of parameters that are input to or output from the action. A very common way of specifying an interface is by means of a set of operations (functions) with parameters, as, for example, used in Java Beans (JB) [25] and Open Services Gateway Initiative (OSGi) [26]. However, there exist other types of interfaces; so-called "port-based,"[6] where ports are entries for receiving/sending different data types and events, as, for instance, implemented in IEC61131 [28] and SaveCCM [13]. Fig. 3 illustrates the "operation-based" and "port-based" interfaces and interaction styles. In the first case, a component invokes an operation from another component (which may return a result), while in the second case, a component pushes data to another component and possibly starts the execution of this other component by sending a trigger. Alternatively, triggers can be sent by a clock invoking the periodical execution of the component.

Most component models distinguish the actions that components provide to their environment, called *provided*

6. Note that the "port-based" concept is different from the concept in UML 2.1 [27] in which a port is defined as a set of interface specifications.

interface, from the actions they require from this environment, called *required interface*. This is an important feature that makes explicit the dependencies of a component. This in turn facilitates independent development and deployment of components.

An interface is not a constituent part of a component, but can exist independently of components as a standard for representing some piece of functionality in a system. The independent existence of interfaces makes it possible to specify interfaces independently of their implementation.

In different stages of development, an interface may be defined through different languages. In the modeling stage, component models may either provide their own languages (often similar to some ADL) or use UML (possibly with some extensions or profiles) for defining interfaces. In the implementation stage, there are two common ways of defining interfaces.

One way is to describe interfaces by means of an Interface Definition Language (IDL) that is independent from a particular programming language. Through mappings between specific programming languages and the IDL, interoperability between multiple programming languages is achieved: Components implemented in different programming languages can be combined into one system. IDLs focus only on syntactic interoperability, but they (implicitly, and sometimes unintentionally) also determine the styles of interaction through which components can communicate. The syntactic interoperability achieved by IDLs yields the benefit of using different programming languages for the component implementations.

Another way of specifying an interface is to directly use a programming language, as, for example, using an object-oriented language. Typically, in object-oriented programming languages, a component is expressed as a class in which the interface is defined as a set of methods and attributes, possibly with some extensions or syntactic convention to distinguish component architectural elements (for instance, required and provided interface). In other languages, the structured (stereotyped) use of header files or abstract classes serves as a means of defining interfaces.

Driven by the requirements of independent deployment and dynamic reconfiguration, some component models define a standard for the binary representation of interfaces. This binary representation is used at the deployment stage and during runtime. Microsoft Component Object Model (MS COM) is an example of a component model that has such a binary standard for interfaces.

To make it possible to perform advanced checks on the compatibility between interfaces, the notion of contract has been adjoined to interfaces. According to [9], contracts can be classified hierarchically in four levels which, if taken together, may form a global contract. In our classification, we adopt the first three levels since the last level is concerned with extra-functional properties which are covered in more detail in Section 4:

- *Syntactic level*: Describes the syntactic aspect, also called the signature of an interface. This level ensures that the interacting components refer to the same data types. This is the most common and most easy agreement to certify as it relies mainly on a (either static or dynamic) type-checking technique.
- *Functional Semantic level*: Reinforces the previous level of contracts in certifying that the values of the parameters are within the proper range. This can be asserted using preconditions, postconditions, and invariants.
- *Behavior level*: Expresses either constraints on the temporal ordering of interactions between components or constraints on the component's internal behavior (e.g., allowed internal states) in response to interactions. Behavior contracts are typically expressed by statecharts or different variants of finite state machines.

We conclude our discussion on aspects of interfaces by pointing out that several component models have distinctive features related to evolvability and variability. For instance, for evolvability (e.g., to support creating new functionality but maintaining backward compatibility), a component may offer multiple interfaces for the same functionality. This makes it possible to embody several versions or variants of functions in the component.

3.2.2 Binding Mechanisms

Binding is the process that establishes connections between components (through use of their interfaces and interaction channels). In CBSE, binding is also often called *component composition* by reference to the composition of the functionality of the components. Similarly, by association with wires in electrical engineering, binding is also referred to as *wiring* in the literature, e.g., [3] and [8].

An important question coming from the possibilities offered by binding mechanisms relates to the composability of components [10]: "Can an assembly, i.e., a set of components mutually connected, be treated as a component itself?" That is, does an assembly composed of a set of components fully comply with the rules imposed by the component model, both in terms of functional and extra-functional properties? The answer is not simple. To discuss component composition, we must first distinguish different types of binding: *horizontal binding* and *vertical binding*, as defined below.

Let us assume that the following components, $C_i = \langle I_i, P_i \rangle$ and $C_j = \langle I_j, P_j \rangle$, satisfy the rules imposed by a component model CM, i.e.,

$$C_i, C_j \models CM \Rightarrow I_i, I_j, P_i, P_j \models CM.$$

If we compose C_i and C_j together through a *horizontal binding* meaning that their respective interfaces are connected together (i.e., $< I_i \oplus I_j >$), then the assembly A resulting from this composition is merely a set of components cooperating together to realize a functionality, i.e., $A = \{C_i, C_j\}$. Here, A does not necessarily comply with the component model CM. In spite of this, this type of binding is often improperly referred to as horizontal composition. At the modeling stage, horizontal binding is often realized by connecting a provided interface of a component with a required interface of another component. At the implementation stage, this horizontal binding is typically realized through glue code or wrappers.

On the other hand, if we identify the assembly A as a component with an interface I_A which is a composition of interfaces of the involved components, i.e., if we have

$$A = \{C_i, C_j\}; A = \langle I_A \rangle \Rightarrow I_A = \langle I_i \oplus I_j \rangle \quad \text{where } I_A \models CM,$$

then A results from a *vertical binding* and has an interface I_A that satisfies the rules of the component model CM. At the modeling stage, vertical binding is often attained through connecting two interfaces of the same kind: a provided interface of the assembly (respectively, required interface) to a provided interface of an inner component (respectively, required interface). This type of connection is called *delegation*. Whereas when all of the interfaces of the inner components are made available to the outside environment through the interfaces of the assembly, we speak of *aggregation*.

If the assembly A satisfies the component model's rules with respect to both its interface I_A and its properties P_A, i.e.,

$$A = \langle I_A, P_A \rangle \Rightarrow A = \langle I_i \oplus I_j, P_{ex} \vdash P_i \oplus P_j \rangle \quad \text{where } I_A, P_A \models CM,$$

then the component model supports *vertical composition*. This is a very powerful property, but unfortunately, very difficult to achieve in practice. Nevertheless, many component models support *partial vertical composition*, in which functional interfaces can be composed recursively.

In SaveCCM [13], vertical binding is supported and the component model defines an assembly as a set of components which exports by delegation a set of selected ports, the interface elements. If the assembly also preserves the "read-execute-write" semantics defined by SaveCCM for components, then in that particular case, the assembly is a component because it complies with the definition of a SaveCCM component.

Binding does not necessarily correspond only to a one-to-one direct connection between two components; some component models also support indirect connections through the utilization of connectors. When introduced as first class citizens of a component model, connectors act as mediators between components and enable 1) making the interaction between components explicit and 2) the addition (and removal) of advanced mediation mechanisms that is transparent to components. In several component models, connectors are implemented as special types of components (e.g., adaptors, brokers, or proxies). Implementing connectors in terms of implementation-level components opens up the possibility of building more complex interactions patterns in comparison to using basic connectors.

The use of connectors corresponds to the concept of *exogenous composition* because the (logic for handling the) interaction between components is handled outside of the components themselves. In contrast to exogenous composition, *endogenous composition* refers to a binding without any intermediary connector. In this case, the handling of binding and interaction protocols is part of the components themselves.

At the modeling and implementation stages, binding is done by a system developer who explicitly states which components are assembled together by connecting the interfaces of the involved components. This is one of the forms of *third-party binding* in which the establishment of the binding is initiated by an entity outside the components involved in the binding. On the other hand, in a *first-party binding*, a component itself decides which other component it is to be bound to. Most of the component models enable the third-party binding. Typical solutions for first-party binding use an introspection (or reflection) interface, which enables the discovery of the interfaces of the components to connect to, and a registry, which can look up the identity of the components that support a specific functionality (or interface).

When the binding occurs at deployment stage, a docking interface is commonly used. 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 runtime infrastructure. In many component models (e.g., CCM, EJB), the binding specification is location transparent: the runtime location of components (placed either on a local or a remote node) is specified separately from the binding information.

3.2.3 Interactions

Component models use one or more architectural styles following specific *interaction styles* to define the patterns of interactions between components, i.e., how components communicate with each other. For instance, the client-server architectural style, widely used for distributed computing, uses a *request-response* interaction model. This means that for any interaction between two components, one component sends a request to a specific other component, which then returns a reply. Hence, traffic across the binding is bidirectional.

Two variants of request response are distinguished. In *asynchronous* request response, the client initiates the communication and continues its activity until, at some point, it receives the results of its request from the server component. The interaction can also be *synchronous*, which means that the client waits until its request has been processed.

Another typical interaction style is *pipe & filter*, which is mostly used for the streaming of events. This style uses unidirectional communication between components. In this style, components are filters that process the data, and the bindings are the pipes that transfer the data to the next filters. A characteristic of this style is that it allows the separate control of the data flow and control flow between components. The control flow is activated by a triggering interaction model which enables the activation of a particular component in response to a particular signal, such as an event, a clock tick, or a stimulus from another component, as illustrated in Fig. 3b. This interaction model includes event-triggering, or event-driven, and time-triggering. The pipe & filter architectural style is widely used in embedded and real-time systems because control theory can be easily mapped to this interaction model. Some component models such as Rubus [29] decouple the specification of data flow from control flow.

There are other interaction styles utilized in component models, and some prominent examples are broadcast, blackboard, and publish subscribe. In most cases, component models provide a single basic interaction style. Support for this style is often hardwired in the execution platform. However, some component models, such as Fractal [30], Pin [20], and Behavior, Interaction, Priority (BIP) [31], allow the construction of different interaction styles.

An interaction style determines which types of dependencies must or may exist between components. As a result, the architectural styles supported by a component model have a large impact on the flexibility during both the development and the execution of components. In general, a style which induces more or stronger dependencies will need more complex protocols for binding and, hence, for the replacement of components.

Components may differ with respect to the way their internal activity and interactions are initiated. *Passive components* are activated only by external events (for example, being called by another component), whereas *active components* manage their activation themselves, and can be executed in a separate thread. Some component models provide support only for passive components (e.g., AUTOSAR, SaveCCM), while others have developed different ways for component startup and execution (e.g., CCM, MS COM). Often, the mechanisms for the activation of components are governed by the underlying middleware [32] or operating system or are taken from the supporting implementation language.

3.2.4 Construction Classification

In accordance with the observations and reasoning from above, we identify the following classification characteristics for interfaces and connections in the construction dimension.

C1. *Interface specification*, in which different characteristics allowing the specification of interfaces are identified:

  a. The distinction of interface type: operation-based (e.g., methods invocations) and port-based interface (e.g., data passing).
  b. The distinction between the provides part and the requires part of an interface.
  c. The existence of some distinctive features.
  d. The language used to specify the interface.
  e. Interface levels which describe the levels of contractualization of the interfaces, namely, syntactic, functional semantic, and/or behavior level.

C2. *Binding*, which describes the characteristics of the patterns and mechanisms used for binding components. It consists of two subtypes:

  a. The exogenous subcategory describes whether the component model includes connectors as architectural elements or not.
  b. The hierarchical subcategory expresses the possibility of having a hierarchical composition of components (horizontal composition is an intrinsic part of all component models, thus it is implicitly assumed to be supported).

C3. *Interactions*, which comprise the following characteristics:

  a. Interaction style, which describes the main underlying architectural style used.
  b. Communication type, which details if the communication used is synchronous and/or asynchronous.

3.3 Extra-Functional Properties

Components and component-based systems are carriers of a number of extra-functional properties. The most basic support that a component model can provide for EFPs is to facilitate specifying such extra-functional properties. For example, in Robocop [15], components may specify the maximum execution time per method of an interface. A specification of such properties makes it possible to check, at the component's deployment, whether a component breaks the system integrity or requires more resources than the system can ensure.

Another type of support that a component model can provide is related to the management of particular EFPs. For example, CCM [22] explicitly provides redundancy mechanisms for managing reliability.

Yet another type of support provided by component models is related to property compositions; it enables the prediction of systems properties derived from the properties of the integrated components and the underlying component framework.

In this section, we discuss the EFP specification, management mechanisms, and EFP composition issues, and then we identify the elements in the classification framework that make it possible to distinguish different component models.

3.3.1 Specification of Extra-Functional Properties

Component models rarely address the specification of EFPs (which by definition belongs to metadata). In many cases, EFPs are specified implicitly, not as a part of a component model, but as a part of the component technology. A basic form of EFP specification is the one proposed by Shaw [33], where an EFP is specified as a triple <*Attribute*, *Value*, *Credibility*> where *Attribute* describes the property itself, *Value* the corresponding data, and *Credibility* specifies the confidence in the value. The attribute *Value* is often a simple data type, but some component models provide a more complex value type (such as a reliability distribution). The Pin component model has an associated "Predictability-Enabled Component Technology (PECT)" [20], [34], which enables the specification and handling of the extra-functional properties through "analytical interfaces." Pin requires that a reasoning framework is specified which defines how to analyze a particular type of property. In Robocop [15], a resource model describes the resource consumption of components in terms of mathematical cost functions, and a behavioral model specifies the sequence in which their operations must be invoked. Based on this information, associated analysis techniques can then analyze the total resource usage and response times. Palladio [18] uses annotations and contracts to specify extra-functional properties that are needed for predicting performance properties of the system under design. Most of the component models define EFPs as attributes of components or, more rarely, as attributes of assemblies or of systems. In ProCOM [35], EFPs can be specified as multiple values including context dependencies and the possibility

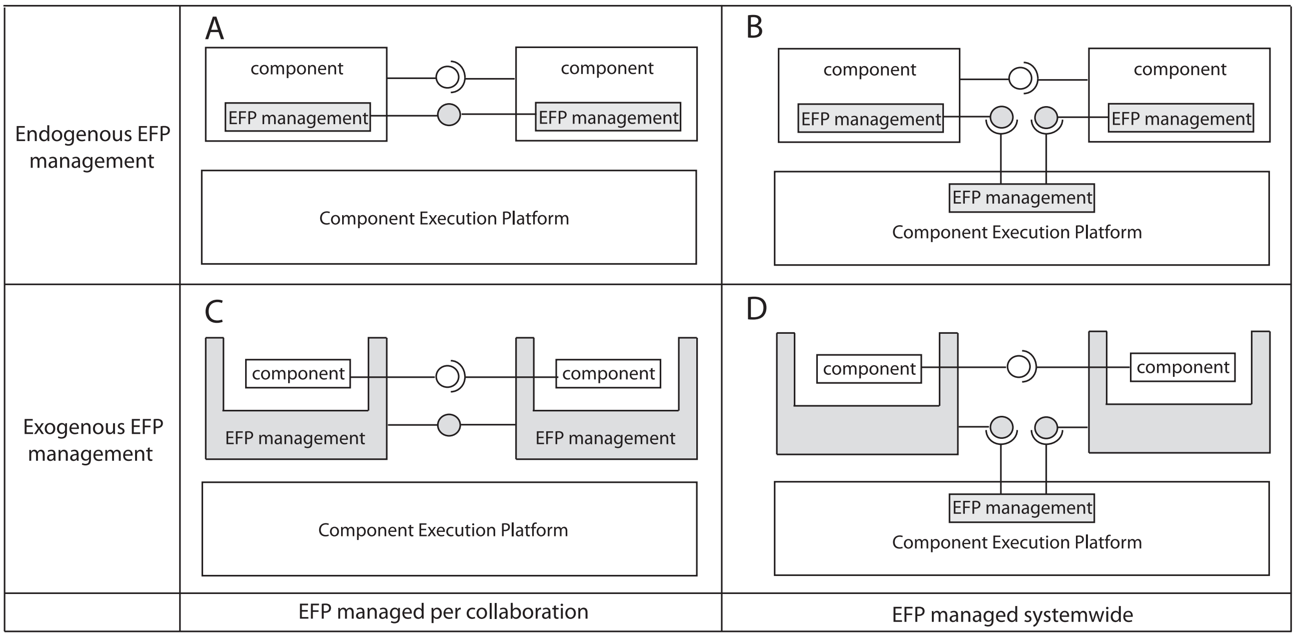
Endogenous EFP management
Exogenous EFP management
A
B
C
D
component
EFP management
Component Execution Platform
EFP managed per collaboration
EFP managed systemwide

Fig. 4. Management of extra-functional properties.

to attach attributes to various component model elements. Here, EFPs are specified as *Attributes* in the following form:

$$Attribute = \langle TypeID,\ Value^{+}\rangle$$
$$Value = \langle Data,\ Metadata,\ ValidityCondition^{*}\rangle,$$

where

- *TypeID* defines the extra-functional property;
- *Data* contain the concrete value for the property;
- *Metadata* provide complementary information on data that allows to distinguish the values; and
- *ValidityConditions* describe the conditions which need to be satisfied to keep the value valid upon reuse.

3.3.2 Management of Extra-Functional Properties

Component models provide different types of support for managing EFP. This management is related to runtime EFPs and realized in combination of components and underlying component execution platform that can often be integrated as a part of a middleware. Different mechanisms for management of EFPs (as well as for component deployments and communication mechanisms) can be found in [32]. We distinguish four types of support (see Fig. 4):

1. *Exogenous Management*. The EFP management is provided outside the components.
2. *Endogenous Management*. The EFP management is implemented in the components, i.e., the component developers are responsible to implement it.
3. *Management per Collaboration*. The EFP management is realized in direct interactions between components.
4. *System Wide Management*. The EFP management is provided by the component framework, or underlying middleware.

By a combination of these types, we get four possible types of the EFP support:

- *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 toward a specific system, and also can cater to 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. A risk of using this approach is a hetereogeneity of policies for handling a single EFP in a system. As a result, managing and predicting emerging properties at the system level can be very difficult.
- *Approach B* (*endogenous system wide*). 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 applied to EFP management.
- *Approach C* (*exogenous per collaboration*). In this approach, components are designed such that they address only functional aspects and are oblivious to EFP. Consequently, in the execution environment, these components are surrounded by a container. This container contains the knowledge on how to manage EFPs. In this approach, containers are connected to other containers. Connected containers then manage the EFPs for the components that they encapsulate.

  The container approach is a way of realizing the 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 EFPs. Because these components do not address EFPs, they are simpler to implement. A disadvantage of the container approaches might be a degradation of the system performance.

- *Approach D* (*exogenous system wide*). This approach is similar to approach C, except that the system can coordinate the management of an EFP from a global system-wide perspective (e.g., global load balancing). Consequently, a more complex support needs to be built into the component execution platform.

3.3.3 Composition of Extra-Functional Properties

The most difficult challenge in CBSE is related to composing EFPs. Compositions of EFPs are based on different composition theories, and, in addition, they are often not only the result of compositions of component properties, but also depend on other elements of a particular system architecture or even its environment. For example, determining the composition of component performance may depend on the scheduling policies and the system architecture. According to [10], EFPs can be classified in categories depending on the composition domains (i.e., type of parameters that determine the composition). The following categories are proposed:

- *Directly composable properties*. A property p_k of an assembly $A = \langle C_1 \oplus C_2 \rangle$ is a function of, and only of, the same property of the components involved:

$$p_k(A) = f(p_k(C_1), p_k(C_2)).$$

An example of such property is static memory consumption. In the simplest case, the system static memory is the sum of component static memories plus a constant.

- *Architecture-related properties*. A property p_k of an assembly $A = \langle C_1 \oplus C_2 \rangle$ is a function of the same property of the components and of the software architecture SA:

$$p_k(A) = f(SA, p_k(C_1), p_k(C_2)).$$

An example of such a property is performance: increasing the amount of parallel processing impacts the performance of the system without changing the properties of individual components (for details, see [10]).

- *Emerging properties*. A property p_k of an assembly $A = \langle C_1 \oplus C_2 \rangle$ depends on several different properties p_i, p_j of the components and of the software architecture:

$$p_k(A) = f\big(SA, p_i(C_1), p_i(C_2), p_j(C_1), p_j(C_2) \dots \big).$$

An example of an emerging property is the response time of an assembly which depends on the execution time and resource consumption of the involved components.

- *Usage-depended properties*. A property of an assembly is determined by its usage profile U:

$$p_k(A, U) = f\big(SA, \dots p_i\big(C_j, U_j\big) \dots\big).$$

Reliability is an example of such a property type. The reliability of a same system can be different for the different usage profiles of that system.

- *System environment context properties*. A property of a system S is determined by other properties and by the state of the system context X defined by external parameters outside the system:

$$p_k(S, U, X) = f\big(SA, X, \dots p_i\big(C_j, U_j\big) \dots\big).$$

Examples of this type are security and safety. These properties also depend on external conditions (such as different measures and procedures).

- *Noncomposable properties*: Properties that are not composable. Examples of such properties are maintainability, robustness, portability, etc.

This classification indicates the limitations of the compositions of EFPs. In general, determining the compositions of component properties becomes feasible only when restrictions are imposed on the design of individual components. In practice, such restrictions are imposed by the rules/constraints 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). Other properties are related to usage profile, and if we cannot predict/specify the usage profile, we cannot predict the system properties.

3.3.4 Extra-Functional Properties Classification

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

E.1. *Management of EFPs*. Which type of management (if any) is provided by the component model?

E.2. *EFP specification*. Does the component model contain means for the specification of specific EFPs? If yes, which properties and in which form?

E.3. *Composability of EFPs*. Does the component model provide means, methods, and/or techniques for the composition of certain extra-functional properties and/or what type of composition?

3.4 The Classification Overview

Fig. 5 summarizes the classification framework in a graph form. The numbered items that describe the classification elements of the three dimensions are listed in the figure.

4 Survey of Component Models

Using the classification framework, we can analyze component models developed in different research groups or in industry. In our classification of component models, the first question is whether a particular approach (model, technology, or method) is a component model or not. This appeared to be a difficult task due to the diversity of component models. Similarly to biology, where viruses straddle the border between life and nonlife, there is a wide range of models, from those having many elements of component models, yet not being considered component models, via those that lack many elements, but still are designated as component models, to those that are widely accepted as component

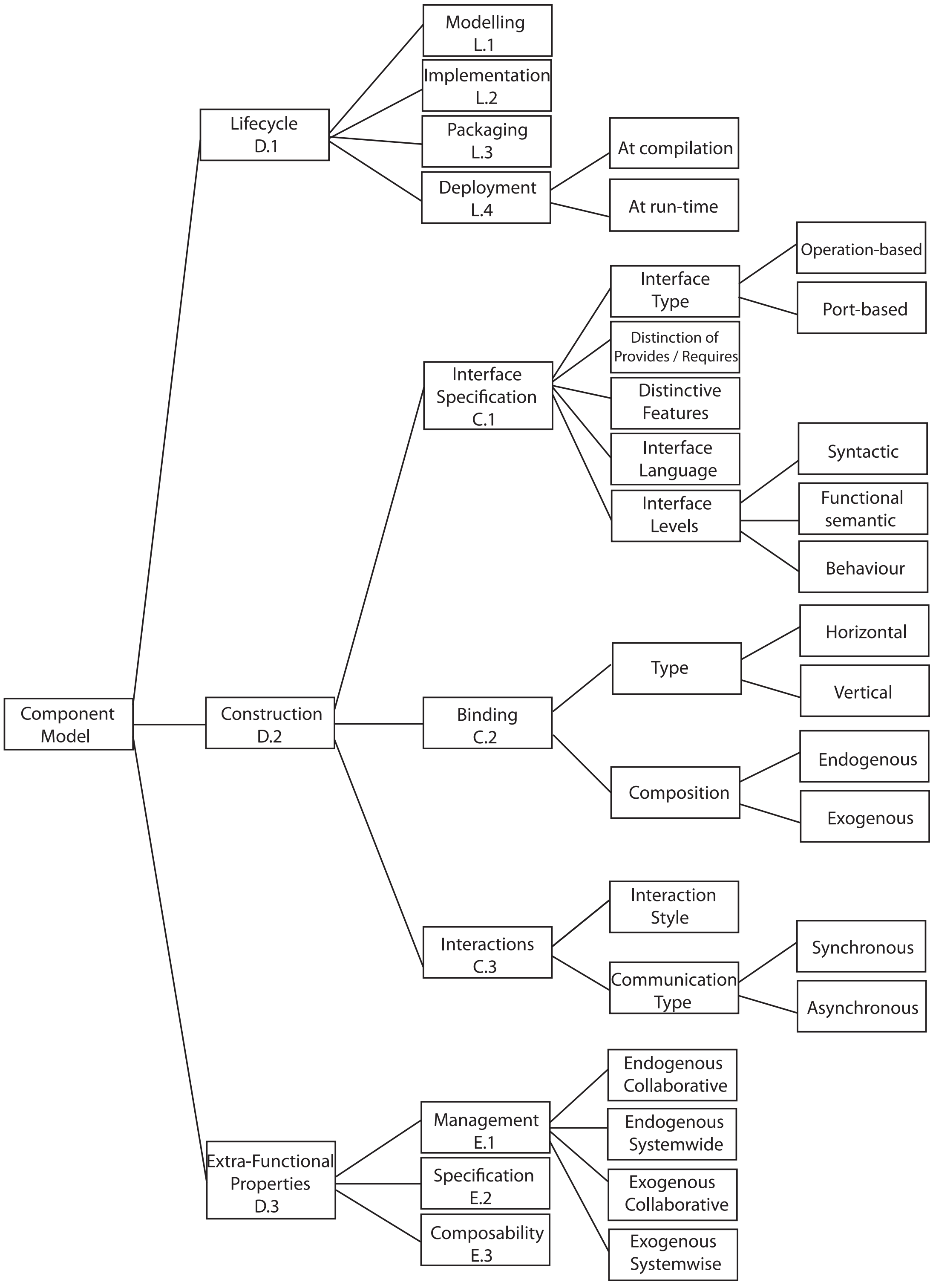

Fig. 5. The hierarchical structure of the classification framework.

models. Therefore, we identify the minimum criteria required to classify an approach as a component model.

The minimum criteria correspond to the definition of component models given in the introduction and in Section 2:

1. A component model includes a component definition.
2. A component model provides rules for component interoperability.
3. Component functional properties are unambiguously specified by component interface.
4. A component interface is used in the interoperability mechanisms.
5. A component is an executable piece of software and the component model either directly specifies its form or unambiguously relates to it via interface and interoperability specification.

Note that the items from the "lifecycle" and "construction" dimensions from the classification framework belong in the minimum criteria, while EFPs are not included in the minimum, and many component models do not provide that support.

There is a wide range of approaches that comply with some of the elements in the minimum criteria. For example, many modeling languages have "components" and even (semi)formally specify components and component compositions. For instance, in ADLs, the basic elements are components [5]. UML 2.0 provides a metamodel for components, interfaces, and ports. Still, we have deliberately chosen not to select them as component models, in contrast to other classifications such as [12]. One reason is that their purpose is not component-based development, but rather the specification of system architectures, and they do not provide any support for components as executable units. Certain languages derived from UML, such as xUML [36], in which the component specification is translated into an executable entity, are even stronger candidates for consideration as component models. However, xUML and similar languages do not operate with components as first class entities (for example, components are not treated as separate development or executable entities), but again the components are treated only as architectural elements.

On the other side of the lifecycle line are services. It can be argued that services are special types of components. Services are focused on runtime retrieval and runtime deployment. Similarly to components, services are specified by an interface and provide support for construction [37]. Still, we have not included services in the classification for similar reasons as those that applied to ADLs—they are not defined as executable units. In analogy to ADLs, services are not component models but rather use component models.

4.1 Component Models

In our classification framework, we have selected 24 component models that we encountered in the research literature and in practice, namely,

- AUTomotive Open System ARchitecture [38],
- Behavior, Interaction, Priority [31],
- BlueArX [39],
- CORBA Component Model [22],
- COMponent-based design of software for Distributed Embedded Systems, version II (COMDES II) [40],
- CompoNETS [41],
- Entreprise JavaBeans [21],
- Fractal [30],
- Koala [17],
- KOmponentenBasieRte Anwendungsentwicklung [19],
- IEC 61131 [28],
- IEC 61499 [42],
- Java Beans [25],
- Microsoft Component Object Model [23],
- OpenCOM [43],
- Open Services Gateway Initiative [26],
- Palladio Component Model [18],
- PErvasive COmponent Systems (Pecos) [44],
- Pin [20],
- PROGRESS Component Model [14],
- Robust Open Component Based Software Architecture for Configurable Devices Project [15], [45],
- Rubus [29],
- SAVE Components Component Model [13], and
- Software Appliances (SOFA) [46].

While some of these component models are in widespread industrial use, others are used as demonstrators or vehicles for illustrating research ideas.

The classification framework does not show the success of particular component models, or any business model, but is based only on their technical characteristics. The component models that we have included in the list are briefly characterized in the Appendix. A more detailed description of each component model with the characteristics defined in the classification framework can be found in the technical report [47].

For some of the component models that we found, our selection criteria were satisfied; however, because of the scarcity of available documentation about some component models, it was impossible to get the necessary 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.

5 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), construction (Tables 2 and 3), and extra-functional properties (Table 4). Following each table, a short discussion summing up our observations is presented.

5.1 Lifecycle Classification

Table 1 shows the lifecycle dimension, indicating the characteristics of the selected component models in different lifecycle stages (modeling, implementation, packaging, and deployment). From this table, we can observe that the most common focus of component models is on the implementation stage. Some component models even exclusively support the implementation stage. Additionally, some component models support the runtime stage by providing a runtime platform that facilitates runtime reconfiguration or a management of extra-functional system properties.

The modeling stage is characterized by extensive use of domain-specific modeling languages, whereas standard modeling languages, such as UML or ADLs, are less common. We can also note that 32 percent of the component models gathered in the framework do not provide any support for the modeling of components or component-based applications, but cover only the implementation part (specification and deployment). All of these component models that omit the modeling stage are from the state of the practice, and many of them widely used. One can ask why component models in practice seldom cover component and system modeling. The reason for this can be found in the common state of the practice. In many industrial projects, designs are expressed in a nonformal way, mainly for documentation purpose only, or in a semiformal way, possibly using UML. In both cases, neither the precise definitions of components nor their

TABLE 1
Classification for the Lifecycle Dimension

Component Models	Modelling	Implementation	Packaging	Deployment
AUTOSAR	N/A	C	Non-formal specification of container	At compilation
BIP	A 3-layered representation: behaviour, interaction, and priority	BIP Language	N/A	At compilation
BlueArX	ASCET-MD models	C	Packages	At compilation
CCM	N/A	Language independent	JARs, DLLs	At run-time
COMDES II	ADL-like language	C	N/A	At compilation
CompoNETS	Petri Nets	Language independent	JARs, DLLs	At run-time
EJB	N/A	Java	JARs	At run-time
Fractal	ADL-like language (Fractal ADL, Fractal IDL), Annotations (Fractlet)	Java (Julia, Aokell) C/C++ (Think) .Net lang. (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	JARs	At compilation
MS COM	N/A	OO languages	DLLs	At compilation and at run-time
OpenCOM	N/A	OO languages	DLLs	At run-time
OSGi	N/A	Java	JARs	At compilation and at run-time
Palladio	UML profile	Java	N/A	At run-time
PECOS	ADL-like language (CoCo)	OO languages	JARs, DLLs	At compilation
Pin	ADL-like language (CCL)	C	DLLs	At compilation
ProCom	ADL-like language REMES	C	File system based repository	At compilation
ROBOCOP	ADL-like language Resource management model	C and C++	ZIP file	At compilation and at run-time
RUBUS	Rubus Design Language	C	File system based repository	At compilation
SaveCCM	ADL-like (SaveComp) Timed automata	C, Java	File system based repository	At compilation
SOFA 2.0	Meta-model based specification language	Java	File system based repository	At run-time

interactions are assumed to be of high priority, and no high needs for modeling components and component-based systems are expressed. This is also an indicator of the differences between state of the art and state of the practice: Many solutions from the state of the art that include the modeling have still not been realized or scaled up in practice.

Further, we can observe from Table 1 that with regard to implementation, component models can be divided into four groups:

1. Language independent (18 percent).
2. OO language-based (36 percent), with a clear dominance of Java.
3. C language (36 percent).
4. Domain-specific language-based (10 percent), either compiled to C or directly interpreted.

The dominance of OO languages is not surprising since technologies based on the OO paradigm are dominant today, and because many principles from OO are directly used or further developed in CBSE. The "C language" component models are prevalent for domain-specific component models that target the development of embedded and real-time systems more. The C language provides more and easier access to details of operating system and underlying hardware platforms facilitating optimizations. Domain-specific programming languages are tightly related to the modeling of component-based systems and components, and are obviously used for a more efficient design and implementation.

Packaging and component repositories are not the main focus of component models. In most cases, certain standard archives are used (such as DLL or JAR packages), also as deployment units. The lack of repositories indicates a low focus on reuse, in particular of COTS components.

Deployment at compile time and runtime almost occurs to an equal extent among the component models being studied. Deployment at compile time limits the flexibility at runtime, but, on the other hand, enables easier predictability, richer

TABLE 2
Classification for the Construction Dimension—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	Operation-based Port-based	No	Complete interface Incomplete interface	BIP Language	Syntactic Semantic Behaviour
BlueArX	Port-based	Yes	Configuration interface Analytic interface	XML adhering to the MSRSW DTD	Syntactic
CCM	Operation-based Port-based	Yes	Facet and receptacle Event sink and event source	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	Facet and receptacle Event sink and event source	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, 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	Data Event	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 interface	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 port Message port	XML based, REMES	Syntactic Behaviour
Robocop	Port-based	Yes	Ability to extend and annotate interface	Robocop IDL (RIDL), Protocol specification	Syntactic Behaviour
RUBUS	Port-based	Yes	Data and trigger port	C header files	Syntactic
SaveCCM	Port-based	Yes	Data, trigger, and data-trigger port	SaveComp (XMLbased) Timed Automata	Syntactic Behaviour
Sofa 2.0	Operation-based	Yes	Utility interface Possibility to annotate interface and control evolution	Java SPC algebra	Syntactic Behaviour

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 (see Table 5).

5.2 Construction Classification

Table 2 presents the interface characteristics of the selected component models, and Table 3 the binding and interaction specifications. Table 2 shows that most of the interfaces are of the operation-based type, which means that the component models use methods and parameters for defining interface signatures. Still, many component models use ports as the interface elements to exchange data. In port-based interfaces, input and output interfaces consist of ports that receive and send data, respectively (often designated as sink and source), hence corresponding to the concepts of provided and required interface. Such component models are typically used in embedded systems and have their basis in hardware components. Several of the component models examined do not distinguish required from provided interfaces, but their interface is referred only to the "provided" interface, which is

TABLE 3
Classification for the Construction Dimension—Binding and Interactions

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

similar to what exists in the object-oriented approach. These component models are essentially used in practice, and were developed earlier, and are even on the way to becoming obsolete (like MS COM, for example). They illustrate the evolution of CBSE.

Because interfaces are a mandatory 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 represented by a particular form of finite state machines (statecharts or timed automata). Here, we distinguish behavior specification of components (used for the modeling and predictability of the behavior of the system) from specifications used for synchronization (for the communication between the components). In a few cases, component models allow behavior specification with resource consumption to be combined, or some other attribute specifications, which makes it possible to model resource usage or performance or some other properties. Examples of such component models are Palladio, SaveCCM, ProCom, and Pin.

Only a few component models offer support for defining the functional semantic level of interfaces. If there is support, then this is mostly addressed through the use of pre and postconditions.

Table 3 (binding and interactions) shows that binding mechanisms in component models are, in most of the cases, of the endogenous type—i.e., connectors are not defined as particular architectural elements. However, many component models use components as connectors or the connectors are automatically generated in the integration/deployment stage and are not being used as entities for modeling.

From Table 3, we can observe that many component models do not support vertical binding. Vertical binding is implemented either through delegated interfaces (i.e., selected interfaces from subcomponents build up the interface of the composite components) or as aggregation, in which the composite component includes all of the interfaces of the aggregated components. Very few component models provide means of hierarchical composition, and if so, then it is only with regard to a few particular EFPs (for example, BIP and SaveCCM for timing properties).

From the information in Table 3, one can conclude that the dominating interaction styles in component models are "request response" (typically used in client/server architectures) and "pipe & filter." Some component models even have additional interaction styles such as event-driven, broadcast, or rendezvous. The choice of the interface style is strongly correlated with the interface type (operation versus port-based) provided by the component model.

TABLE 4
Classification for the Extra-Functional Properties Dimension

Component Models	Management of EFP	EFP specification	Composability of EFP
AUTOSAR	Endogenous per collaboration (A)	N/A	N/A
BIP	Endogenous system wide (B)	Timing properties	Behaviour compositions
BlueArX	Endogenous system wide (B)	Resource usage and timing properties	Reasoning frameworks
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	Exogenous system wide (D)	N/A	N/A
Fractal	Exogenous per collaboration (C)	Ability to add property (by adding property controller)	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	Performance properties
PECOS	Endogenous system wide (B)	Generic specification of properties including timing properties	N/A
Pin	Exogenous system wide (D)	Timing properties (by adding analytic interface)	Different EFP composition theories (ex: latency)
ProCom	Endogenous system wide (B)	Generic specification of properties including timing and resources properties	Timing and resources properties 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 properties	Timing properties at design time
SaveCCM	Endogenous system wide (B)	Generic specification of properties including timing properties	Timing properties at design time
SOFA 2.0	Endogenous system wide (B)	Behavioural (protocols)	Composition at design

The dominant communication type in component models is synchronous. Component models that provide support for asynchronous communication also support synchronous communication. This indicates that component models are not concerned with architecture (architectural design), but rather with targeting detailed design.

5.3 Extra-Functional Properties Classification

Table 4 summarizes the characteristics of the selected component models with respect to EFPs. We observe that many component models provide certain support for the management of EFPs, either system-wide or per container (characteristic examples are redundancy, or authentication support). In several cases, a particular EFP support is implemented as an extension to a standard technology (for example, COM+ used in MS COM and .NET technologies). However, a smaller number of component models have formalisms for EFP specifications. A significantly smaller number of component models provide means for the composition of EFPs. This is particularly true for commercial component models. Clearly, the composition of EFPs still belongs to the research challenges. A majority of EFPs that are managed by component models belong to resource usage and timing properties.

TABLE 5
General-Purpose and Domain-Specific Component Models

Domain	AUTOSAR	BIP	BlueArX	CCM	COMDES II	CompoNETS	EJB 3.0	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	

6 Component Models and Domains

The characteristics listed in the classification framework show some patterns: Similar solutions belong to component models from similar application domains, as, for instance, embedded systems or information systems. That is to say that the requirements from the application domain penetrate into the component model. Such component models are, as a consequence, specialized and not so usable in domains that are subject to different requirements.

The other type of component models that have similar solution patterns are general-purpose component models. They provide basic mechanisms for the specification and composition of components, but do not assume any specific architecture beyond general assumptions (like interaction style, support for distributed systems, compilation, or runtime deployment). A general solution that enables component models to be both generally applicable and to cater for specific domains is the use of optional frameworks.

According to this, we distinguish the component models as

- general-purpose component models;
- specialized component models.

Table 5 lists the selected component models according to their dominant use in particular domains.

We see that the distribution between general-purpose component models and specialized component models is equal. It is likely that there are more specialized, proprietary component models that are not published. 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 a general-purpose component model in different domains. Conversely, general-purpose component models have been adapted for particular domains by the addition of new features or by applying some restriction to certain functions.

Specialized component models from our selection belong to two domains: 1) embedded systems and 2) distributed information systems. Component models from the embedded systems domain have some common characteristics: The "pipe & filter" interaction style is used, components are usually deployable at compilation time, are resource aware, and often there is support for the management of timing properties. These component models are significantly different from general-purpose component models. The component models from the information systems domains are more similar to general-purpose component models. Typically, they have similar characteristics as general-purpose component models, such as the use of "request-response" interaction style, support for runtime deployment, expandable interface, and implementation in object-oriented languages. Component models that target information systems differ 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. In some cases, an extension of a component model is used for its specialization (for example, COM+ is an addition to COM used for distributed component-based systems).

Some general-purpose component models have a special feature; they have mechanisms for generating new component models. They provide a set of common principles and mechanisms to add new features, or change the existing ones (for example, different implementation mechanisms for bindings or interactions). An example of these "generative" component models is Fractal. Fractal supports several variants of particular component model elements—for example, different type of binding and interaction, and the use of different programming languages (Fractal has Java-based and C-based implementations). Another example of such a component model is Robocop. It provides a mechanism for adding different elements of the model (such as modeling languages, implementations, metadata in a form of documentation, and management for EFPs). A particular instance of a Robocop is a component model that includes selected elements.

From the characteristics defined in the tables, we can observe that although there are many component models, they show similar patterns within the same or related domains. We can conclude that this gives us a good basis to converge different component models into a smaller number of component models dedicated to domain-specific requirements.

7 Related Work

Over the last decade, several attempts have been made to identify key features of aspects of component software approaches: classification studies of components and interfaces [48], [49], interfaces, extra-functional properties [10], ADLs [5], component models [12], and characteristics of component models for particular business domains [11], among others.

The work presented in [48] and [49] does not consider any component model but rather focuses on practical issues of component utilization and reutilization. In [48], 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 concentrates on the interaction between components and the executing platform. Similarly, in [49], a model for characterizing components is proposed which reuses the classification model of interfaces from Yacoub et al. [48], where a component is 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 features that relates to the use of a component in a team and over time. These features can influence the selection of a component such as its age, its provenience, 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 nonfunctional 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, behavioral aspects, and accessibility to source code).

A classification that is similar in spirit to our work is proposed in [50]. The classification framework presented in [50] attempts to determine the core features of a software component. This classification is different from ours: It includes the 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 nonorthogonality of some of the characterized items.

In [5], where 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 [11], 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 50 elements among 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 [8], 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 evaluated, but rather these characteristics are described for each component model.

In [12], 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. Furthermore, the different types of bindings (compositions) of some of the component models are discussed in more detail. This taxonomy does not consider EFPs.

8 Conclusion

In this survey, we have presented a framework for the classification and comparison of component models. This survey indicates that many principles comprised in the component-based approach are not always included in every component model. Hence, there is no complete set of principles that applies to all component models. Many of the principles used in component models are taken and further developed from other approaches (OO development, ADLs), which provided diverse solutions for similar approaches.

The intention of this work is to increase understanding of the component-based approach by identifying the main concerns, common characteristics, and differences of component models. The proposed framework does not include all of the elements of all component models since many of them have unique solutions—some related to models, some related to particular technology solutions. However, the framework identifies the minimal criteria for considering a model to be a component model, and it groups the basic characteristics of component models and enables a more systematic approach in their analysis and comparison.

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" is the predominate style. We can also observe that the support for composition of extra-functional properties is rather scarce. There are several reasons for that: in practice, explicit modeling and reasoning about of EFPs is still not widespread. Furthemore, there are many different EFPs and many of them are not composable, or not directly composable, but depend of external factors such as underlying platform, usage scenario, or a content in which the system is running. On the other hand, support for managing EFPs is a common feature of component models.

Based on experiences coming from other technologies, we could expect a convergence of the main characteristics of component models, i.e., they become more standardized, using more commonly accepted concepts and terminology, even if the number of different component models will not necessarily decrease. The aim of this work is to provide a help in this convergence process.

Appendix

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.

AUTomotive Open System ARchitecture [38], the new standard in the 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 AUTSOAR is the architecture not the component model itself.

Behavior, Interaction, Priority [31] framework developed at Verimag is used for modeling heterogeneous real-time components. This heterogeneity is considered for components having different synchronization mechanisms (broadcast/rendezvous), timed components, or nontimed components. BIP focuses on component behavior through a model with a three-layer structure of the components (Behavior, 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-tool set [51] and the PROMETHEUS tools [52] are also provided.

BlueArX [39] 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 runtime or memory resources on the target hardware. A BlueArX component consists of specification, documentation, and implementation (as object or C source code). Modeling is usually done using ASCET-MD[7] models. Implementation is done in C. Components are delivered as so-called packages, and are both exchanged between Bosch teams and shipped to customers in this format. BlueArX interfaces are specified using Manufacturer Supplier Relationship Software (MSRSW), 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, such as timing, memory usage, or generic specification of other properties.

COMponent-based design of software for Distributed Embedded Systems, version II [40], developed at the 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 latter two describe the sequential behavior (control flow). All nonfunctional information such as physicality, real time, and concurrency is specified with respect to actors.

CompoNETS [41], developed at Université Toulouse 1, is based on CCM where, additionally, the internal behavior of a software component and intercomponent 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.

7. A software modeling tool developed by the ETAS group http://www.etas.com.

CORBA Component Model [22] 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. CCM provides a Component Implementation Framework (CIF) which relies on Component Implementation Definition Language (CIDL) and describes how functional and nonfunctional parts 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.

Enterprise JavaBeans [21], 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, and interoperability. EJB offers 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, and 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 the 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 [17] 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 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 its implementation language and uses source code components with simple interaction model. Koala pays special attention to resource usage such as static memory consumption.

KOmponentenBasieRte Anwendungsentwicklung [19] 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, and .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 [28] 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 [42], 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.

Java Beans [25], developed by Sun Microsystems, is based on the Java programming language. In the Java Beans 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: 1) properties (similar to the attributes of a class), 2) methods, and 3) events, which are an alternative to method invocation for sending data. Java Beans was primarily designed for the construction of graphical user interface. The model defines three types of interaction points, referred to as ports: 1) methods, as in Java, 2) properties, used to parameterize the component at composition time, and 3) event sources and event sinks (called listeners) for event-based communication.

Microsoft Component Object Model [23] 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 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 [43] 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.

Open Services Gateway Initiative [26] is a consortium of numerous industrial partners working together to define a service-oriented framework with an open specification 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 also offers, 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 runtime. In relying on Java, OSGi is platform independent. There exist several additions of OSGi that provide additional characteristics.

Palladio Component Model [18], developed at the University of Oldenburg and the 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 systems 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.

PErvasive COmponent Systems [44] is a joint 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 the design phase, composite components are made by linking their ports with connectors. Pecos targets C++ or Java as the implementation language, so the runtime 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 behavior of embedded systems.

Pin [20] component model developed at the Carnegie Mellon Software Engineering Institute (SEI) is used as a basis in prediction-enabled component technologies. 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 for Pin Component model: for predicting average latency in assemblies with periodic tasks, for predicting average latency in stochastic tasks managed by a sporadic server, and 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.

PROGRESS Component Model [14] 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 make 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). In ProCom, the functional and extra-functional behavior (such as timing and resource consumption) of components may be described in a dense time state-based hierarchical modeling language called REsource Model for Embedded Systems (REMES) [53]. Further, ProCom considers deployment as a specific activity which includes component allocations, transformation of components to the entities complied with the execution model, and synthesis, i.e., creation of a glue code.

Robust Open Component Based Software Architecture for Configurable Devices Project [15] 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 of 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 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 ISO standard ISO/IEC 23004-1:2007 Information technology—Multimedia Middleware.

Rubus [29] 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 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.

SAVE Components Component Model [13], 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 analyzability of the dependability and of the real-time properties. The model takes into consideration the resource usage, and provides a lightweight runtime framework. For component and system specification, SaveCCM uses "SaveCCM language" which is based on a textual XML syntax and on a subset of UML 2.0 component diagrams.

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 as a gray-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. 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.

ACKNOWLEDGMENTS

This work was partially supported by the Swedish Foundation for Strategic Research via the strategic research centre PROGRESS and Q-ImPrESS research project (FP7-215013) by the European Union under the Information and Communication Technologies priority of the Seventh Research Framework Programme. The authors would like to thank a number of researchers and industrial experts, in particular, Kurt Wallnau, (SEI/Carnegie Mellon University), 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), and Christian Zeidler (ABB Research) for their valuable comments, and the anonymous reviewers who have read previous versions of this paper and provided constructive suggestions.

References

[1] B. Meyer, "What to Compose," *Software Development Magazine,* Beyond Objects column, http://www.ddj.com/architect/184414588, Mar. 2000.

[2] C. Szyperski, *Component Software: Beyond Object-Oriented Programming.* Addison-Wesley Professional, Dec. 1997.

[3] G.T. Heineman and W.T. Councill, *Component-Based Software Engineering: Putting the Pieces Together.* Addison-Wesley Longman Publishing Co., 2001.

[4] N. Medvidovic, E.M. Dashofy, and R.N. Taylor, "Moving Architectural Description from under the Technology Lamppost," *Information and Software Technology,* vol. 49, no. 1, pp. 12-31, 2007.

[5] N. Medvidovic and R.N. Taylor, "A Classification and Comparison Framework for Software Architecture Description Languages," *IEEE Trans. Software Eng.,* vol. 26, no. 1, pp. 70-93, Jan. 2000.

[6] M.R.V. Chaudron and I. Crnkovic, *Software Engineering: Principles and Practice,* third ed., ch. 18. Wiley, 2008.

[7] I. Crnkovic, M.R.V. Chaudron, and S. Larsson, "Component-Based Development Process and Component Lifecycle," *J. Computing and Information Technology,* vol. 13, no. 4, pp. 321-327, Nov. 2005.

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

[9] A. Beugnard, J.-M. Jézéquel, N. Plouzeau, and D. Watkins, "Making Components Contract Aware," *Computer,* vol. 32, no. 7, pp. 38-45, 1999.

[10] I. Crnkovic, M. Larsson, and O. Preiss, "Concerning Predictability in Dependable Component-Based Systems: Classification of Quality Attributes," *Architecting Dependable Systems III,* pp. 257-278, 2005.

[11] G. Kotonya, I. Sommerville, and S. Hall, "Towards a Classification Model for Component-Based Software Engineering Research," *Proc. 29th EUROMICRO Conf.,* pp. 43-52, 2003.

[12] K.-K. Lau and Z. Wang, "Software Component Models," *IEEE Trans. Software Eng.,* vol. 33, no. 10, pp. 709-724, Oct. 2007.

[13] M. Åkerholm, J. Carlson, J. Fredriksson, H. Hansson, J. Håkansson, A. Möller, P. Pettersson, and M. Tivoli, "The SAVE Approach to Component-Based Development of Vehicular Systems," *J. Systems and Software,* vol. 80, no. 5, pp. 655-667, May 2007.

[14] S. Sentilles, A. Vulgarakis, T. Bures, J. Carlson, and I. Crnkovic, "A Component Model for Control-Intensive Distributed Embedded Systems," *Proc. 11th Int'l Symp. Component Based Software Eng.,* pp. 310-317, Oct. 2008.

[15] H. Maaskant, *A Robust Component Model for Consumer Electronic Products,* vol. 3, pp. 167-192. Springer, 2005.

[16] J. Cheesman and J. Daniels, *UML Components: A Simple Process for Specifying Component-Based Software.* Addison-Wesley Longman Publishing Co., Inc., 2000.

[17] R. van Ommering, F. van der Linden, J. Kramer, and J. Magee, "The Koala Component Model for Consumer Electronics Software," *Computer,* vol. 33, no. 3, pp. 78-85, 2000.

[18] S. Becker, H. Koziolek, and R. Reussner, "Model-Based Performance Prediction with the Palladio Component Model," *Proc. Sixth Int'l Workshop Software and Performance,* pp. 54-65, 2007.

[19] 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 Longman Publishing Co., Inc., 2002.

[20] S. Hissam, J. Ivers, D. Plakosh, and K.C. Wallnau, "Pin Component Technology (V1.0) and Its C Interface," Technical Note: CMU/SEI-2005-TN-001. www.sei.cmu.edu/pub/documents/05.reports/pdf/05tn001.pdf, Apr. 2005.

[21] E.E. Group, "JSR 220: Enterprise JavaBeansTM,Version 3.0 EJB Core Contracts and Requirements Version 3.0, Final Release," May 2006.

[22] "OMG CORBA Component Model v4.0," http://www.omg.org/spec/CCM/4.0/, 2011.

[23] D. Box, *Essential COM. Object Technology Series.* Addison-Wesley, 1997.

[24] *Oxford Advanced Learners Dictionary,* http://www.oxfordadvancedlearnersdictionary.com/, 2011.

[25] Sun Microsystems, "Javabeans Specification," java.sun.com/javase/technologies/desktop/javabeans/docs/spec.html, 1997.

[26] OSGi Alliance, "OSGi Service Platform Core Specification, V4.1," 2007.

[27] T.O.M. Group, "UML Superstructure Specification v2.1," http://www.omg.org/spec/UML/2.1.2/, Apr. 2009.

[28] IEC, "Application and Implementation of IEC 61131-3," IEC, 1995.

[29] K. Hanninen, J. Maki-Turja, M. Nolin, M. Lindberg, J. Lundback, and K. Lundback, "The Rubus Component Model for Resource Constrained Real-Time Systems," *Proc. Int'l Symp. Industrial Embedded Systems,* pp. 177-183, June 2008.

[30] E. Bruneton, T. Coupaye, and J. Stefani, "The Fractal Component Model Specification," The ObjectWeb Consortium, technical report, http://fractal.objectweb.org/specification/index. html, Feb. 2004.

[31] A. Basu, M. Bozga, and J. Sifakis, "Modeling Heterogeneous Real-Time Components in Bip," *Proc. Fourth IEEE Int'l Conf. Software Eng. and Formal Methods,* pp. 3-12, 2006.

[32] W. Emmerich, M. Aoyama, and J. Sventek, "The Impact of Research on the Development of Middleware Technology," *ACM Trans. Software Eng. and Methodology,* vol. 17, no. 4, pp. 1-48, 2008.

[33] M. Shaw, "Truth vs Knowledge: The Difference between What a Component Does and What We Know It Does," *Proc. Int'l Workshop on Software Specification and Design,* pp. 181-185, 1996.

[34] "PECT Homepage," www.sei.cmu.edu/pacc/pect_init.html, accessed July 2008.

[35] S. Sentilles, P. Stepan, J. Carlson, and I. Crnkovic, "Integration of Extra-Functional Properties in Component Models," *Proc. 12th Int'l Symp. Component Based Software Eng.,* June 2009.

[36] S.J. Mellor and M. Balcer, *Executable UML: A Foundation for Model-Driven Architectures.* Addison-Wesley Longman Publishing Co., Inc., 2002.

[37] H. Breivold and M. Larsson, "Component-Based and Service-Oriented Software Engineering: Key Concepts and Principles," *Proc. EUROMICRO Conf. Software Eng. and Advanced Applications,* pp. 13-20, Aug. 2007.

[38] AUTOSAR Development Partnership, "AUTOSAR—Technical Overview V2.0.1," www.autosar.org, June 2006.

[39] J.E. Kim, O. Rogalla, S. Kramer, and A. Haman, "Extracting, Specifying and Predicting Software System Properties in Component Based Real-Time Embedded Software Development," *Proc. 31st Int'l Conf. Software Eng.,* 2009.

[40] X. Ke, K. Sierszecki, and C. Angelov, "COMDES-II: A Component-Based Framework for Generative Development of Distributed Real-Time Control Systems," *Proc. 13th IEEE Int'l Conf. Embedded and Real-Time Computing Systems and Applications,* pp. 199-208, 2007.

[41] R. Bastide and E. Barboni, "Component-Based Behavioural Modelling with High-Level Petri Nets," *Proc. Third Workshop Modelling of Objects, Components and Agents,* pp. 37-46, Oct. 2004.

[42] IEC, "IEC 61499 Function Blocks for Embedded and Distributed Control Systems Design," IEC, 2005.

[43] M. Clarke, G. Blair, G. Coulson, and N. Parlavantzas, "An Efficient Component Model for the Construction of Adaptive Middleware," *Proc. IFIP/ACM Int'l Conf. Distributed Systems Platforms,* 2001.

[44] M. Winter, C. Zeidler, and C. Stich, "The PECOS Software Process," *Proc. Workshop Components-Based Software Development Processes,* 2002.

[45] J. Muskens, M.R.V. Chaudron, and J.J. Lukkien, "A Component Framework for Consumer Electronics Middleware," *Component-Based Software Development for Embedded Systems,* pp. 164-184, Springer, 2005.

[46] T. Bureš, P. Hnetynka, and F. Plášil, "SOFA 2.0: Balancing Advanced Features in a Hierarchical Component Model," *Proc. Int'l Conf. Software Eng. Research, Management and Applications,* pp. 40-48, 2006.

[47] J. Feljan, L. Lednicki, J. Maras, A. Petricic, and I. Crnkovic, "Classification and Survey of Component Models," Technical Report MRTC report ISSN 1404-3041 ISRN MDH-MRTC-242/2009-1-SE, http://www.mrtc.mdh.se/index.php?choice=publications&id=2099, Dec. 2009.

[48] S. Yacoub, H. Ammar, and A. Mili, "A Model for Classifying Component Interfaces," *Proc. Second Int'l Workshop on Component-Based Software Eng. in Conjunction with the 21st Int'l Conf. Software Eng.*, pp. 17-18, 1999.

[49] S. Yacoub, H. Ammar, and A. Mili, "Characterizing a Software Component," *Proc. Second Workshop on Component-Based Software Eng. in Conjunction with Int'l Conf. Software Eng.*, 1999.

[50] K.J. Fellner and K. Turowski, "Classification Framework for Business Components," *Proc. 33rd Hawaii Int'l Conf. System Sciences*, vol. 8, p. 8047, 2000.

[51] M. Bozga, S. Graf, I. Ober, I. Ober, and J. Sifakis, "The IF Toolset," *Formal Methods for the Design of Real-Time Systems, Int'l School on Formal Methods for the Design of Computer, Comm., and Software Systems, SFM-RT 2004*, pp. 237-267, 2004.

[52] G. Gössler, "Prometheus—A Compositional Modeling Tool for Real-Time Systems." *Proc. Workshop Real-Time System Tools*, 2001.

[53] C. Seceleanu, A. Vulgarakis, and P. Pettersson, "REMES: A Resource Model for Embedded Systems," *Proc. 14th IEEE Int'l Conf. Eng. Complex Computer Systems*, June 2009.

Ivica Crnković received the MSc degrees (1981) in computer science and theoretical physics (1984), and the PhD degree (1991) in computer science, all from the University of Zagreb, Croatia. After 15 years of work in industry, he moved to academia in 1999. He is a professor of software engineering and the chair of the Software Engineering Division at Mälardalen University, Sweden, and a professor on the Faculty of Electrical Engineering, University of Osijek, Croatia. He is the coauthor of two books and more than 100 refereed publications on software engineering topics. He was a general chair of ESEC/FSE '07 conference, CBSE '06, Euromicro SEAA '05, and he organized several other conferences and workshops in the area of software engineering. Currently, he is the general chair of Comparch '11, a federated conference that includes CBSE '11, QoSA '11, and ISARCS '11. During 2009-2011, he chaired the Comparch steering committee, and he is the cochair of the Euromicro SEAA technical committee. His research interests include component-based software engineering, software architecture, software configuration management, software development environments and tools, and software engineering in general. He is a member of the IEEE.

Séverine Sentilles received the Licentiate degree (2009) in computer science from Mälardalen University, Sweden, and the MSc (2006) degree in computer science from the Université de Pau et des pays de l'Adour in France. She is currently working toward the PhD degree at Mälardalen University. Her main research interests are centered on component-based software engineering, model-driven engineering, and software development environments and tools. She is a student member of the IEEE.

Aneta Vulgarakis received the Licentiate degree (2009) in computer science from Mälardalen University, Sweden, and the MSc degree (2006) from the Faculty of Electrical Engineering, Macedonia with professional specialization in computer science, information technology, and automation. She is currently working toward the PhD degree at Mälardalen University. Her main research interests are component-based software engineering, formal models, and verification techniques for constructing correct and predictable real-time embedded systems. She is a student member of the IEEE.

Michel R.V. Chaudron received the MSc (1992) and PhD (1998) degrees in computer science from Leiden University, The Netherlands. He spent a couple of years working in the IT industry, after which he worked at TU Eindhoven. Currently, he is an associate professor at the Leiden Institute of Advanced Computer Science, where he heads the ICT in Business MSc 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 participant in conferences in these areas.