Classification and Survey of Component Models

Ivica Crnković, Aneta Vulgarakis
Mälardalen research and technology centre
PO Box 883, SE-721 23, Västerås, Sweden
{ivica.crnkovic, aneta.vulgarakis}@mdh.se

Mario Žagar, Ana Petričić, Juraj Feljan, Luka Lednicki
Faculty of Electrical Engineering and Computing
University of Zagreb, Croatia
{mario.zagar, ana.petricic, juraj.feljan, luka.lednicki}@fer.hr

Josip Maras
Faculty of Electrical Engineering, Mechanical Engineering and Naval Architecture
University of Split, Croatia
josip.maras@fesb.hr

Abstract: **As component-based software engineering is growing and its usage expanding, more and more component models are developed. In this paper we present a survey of software component models in which models are described and classified respecting the classification framework for component models proposed by Crnković et. al. [1]. This framework specifies several groups of important principles and characteristics of component models: lifecycle, constructs, specification and management of extra-functional properties, and application domain. This paper gives examples three component models using the classification framework.**

1. INTRODUCTION

Today there exist many component models. Some component models target specific application domains, such as embedded systems or business domains. Other component models are domain-independent, but are based on certain technological platforms. All component models are based on some, often implicit, assumptions about the architecture of the types of systems they are targeting. For this reason different component models have similar yet different principles, often not explicitly expressed and used in the technologies. In [1], a framework that identifies characteristics and common characteristics of component models, and that enables comparison between different component models, is provided. In this paper we use this framework and give an overview of three component models, namely AUTOSAR, ProCom, a domain specific component models, and EJB a general-purpose component model.

The rest of the paper is organized as follows. Section 1 gives a short overview of the classification framework, section 2 a short introduction of three component models selected. Sections 3,4,and 5 give a more comprehensive description of each component model.

2. THE CLASSIFICATION FRAMEWORK

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

- Lifecycle. The lifecycle dimension identifies the support provided by a component model and the component forms throughout the lifecycle of components.
- Construction. The construction dimension identifies principles and mechanisms for building systems from components including (i) the component functional specification (of which the *interface* is a prominent part), (ii) the means of establishing connections between the components, i.e. *biding*, and the means of intercommunications, i.e. *interactions* between the components.
- 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.

Details of these three dimensions are in detail described in [1], and according to it the models are described below.

3. CLASSIFICATION OF SELECTED COMPONENT MODELS

This section provides a classification of three component models according to the classification framework; ProCom – a research component model for embedded systems developed at Mälardalen University; Enterpise JavaBeans – an industrial model developed by Sun Microsystems and primarily used for a client – server model of distributed computing; AUTOSAR – an industrial component model used in development of vehicular embedded systems. The classification is presented in Table 1.

Table 1: Classification of selected component models

Lifecycle				
Component models	**Modeling**	**Implementation**	**Packaging**	**Deployment**
AUTOSAR	use of virtual functional bus	C	non-formal specification of container	at compilation
EJB	N/A	Java	EJB-Jar files	at run-time
ProCom	ADL-like language, timed automata	C	file system based repository	at compilation

Constructs - interface specification					
Component models	**Interface type**	**Distinction of provides and requires**	**Distinctive features**	**Interface language**	**Interface levels**
AUTOSAR	operation-based, port-based	yes	AUTOSAR Interface	C header files	syntactic
EJB	operation-based	no	N/A	Java + annotations	syntactic
ProCom	port-based	yes	data- and trigger ports	XML based, timed automata	syntactic, behavioral

Constructs - interaction				
Component models	**Interaction styles**	**Communication type**	**Binding type**	
			Exogenous	**Hierarchical**
AUTOSAR	request-response, message passing	synchronous, asynchronous	no	delegation
EJB	request-response	synchronous, asynchronous	no	no
ProCom	pipe-and-filter, message passing	synchronous, asynchronous	yes	delegation

EFPs			
Component models	**Management of EFPs**	**Properties specification**	**Composition and analysis support**
AUTOSAR	endogenous per collaboration (A)	N/A	N/A
EJB	exogenous systemwide (D)	N/A	N/A
ProCom	endogenous systemwide (B)	timing and resources	timing and resources at design- and compile-time

Domains	
Component models	**Domain**
AUTOSAR	specialized
EJB	general-purpose
ProCom	specialized

4. PROCOM

ProCom [2] is a component model for control-intensive distributed embedded systems and is designed to cover the whole development process in the vehicular-, automation- and telecommunication domains.

Typically, complex distributed embedded systems from targeted domains have different characteristics at different levels of granularity. ProCom tackles this problem by using two layers: *ProSys* and *ProSave*.

4.1 ProSys

ProSys is a hierarchical component model which acts as an upper layer that models the system as a number of active and concurrent subsystems which communicate by asynchronous message passing.

ProSys subsystems can be: composite subsystems, subsystems realized with ProSave or wrapped legacy subsystems. Each subsystem is specified by:

- Typed input- and output ports which express what messages the subsystem receives and sends. Message ports are connected with message channels which support n-to-n communication.
- Attributes and models related to functionality, reliability, timing and resource usage.

4.2 ProSave

ProSave is the lower layer which models the internal design of a single ProSys subsystem down to primitive functional components implemented by code.

ProSave components are passive, reusable units of functionality that can either be realized by code (C functions), or by interconnected sub-components. They use pipe-and-filter communication paradigm and are typically not distinguishable as individual units in the final executing system.

The architectural style is based on a data/control flow model with a separation of data transfer and control flow, which is manifested with the existence of data- (enable data read, write) and trigger ports (control the activation of components).

Information about a component is represented using structured attributes and its functionality is made available by a set of services. Attributes define simple or complex types of component properties such as behavioral models, resource models, dependability measures and documentation. Each service consists of:

- *Input port group* – contains a trigger port for activation and a set of data ports for required data.
- *Output port groups* – contains a set of data ports and a trigger port which indicates when new data is available.

Components can be connected using:

- Simple connections that connect two ports and that can be used to transfer data or control.
- Connectors – constructs that may be used to control the data- and control-flow, for example to fork or join data or trigger connections.

Components and information about them (requirements, behavior models, resource usage) are stored in a file system based repository.

4.3 Connecting ProSave and ProSys

ProSys subsystems can be defined with a collection of interconnected ProSave components, ProSave connectors, and additional connector types such as:

- Input message port which acts as a ProSave connector with one output trigger port and one output data port. Whenever a ProSys subsystem receives a message, the message port writes message data to the output port and activates the output trigger.
- Output message port is similar to the input message port only it has one input trigger and one input data port. When a trigger is received it sends a message with the data from the data port.
- Clock is used for generating periodic triggers. It only has one output port which is periodically triggered.

5. EJB

Enterprise JavaBeans (EJB) is a component model developed by Sun Microsystems with current version 3.0 [3]. EJB has quite limited scope but despite its limitations, it has been widely used and popular in Java community. EJB is primarily used for a client – server model of distributed computing. It envisions the construction of object-oriented and distributed business applications. The model simplifies the development of middleware by providing server support for a set of services, such as transactions, security, persistence, concurrency and interoperability.

The EJB component model logically extends the JavaBeans [4] component model to support *server components*. Server components are reusable, prepackaged pieces of application functionality that are designed to run in an application server. They are similar to development components, but they are generally larger grained and more complete than development components. EJB components (*enterprise beans*) cannot be manipulated by a visual Java

IDE in the same way that JavaBeans components can. Instead, they can be assembled and customized at deployment time using tools provided by an EJB-compliant Java application server.

5.1 Constructs

EJB components

An enterprise bean is a reusable, portable J2EE component which consists of methods that encapsulate business logic, and run inside an EJB Container. EJB components are limited to Java programming language, but they may be invoked from various other languages e.g. C++, C#, Visual Basic .NET. The EJB 3.0 bean class can be a pure Java class often referred as POJO and the interface can be a simple business interface.

EJB specification introduces three kinds of components called *beans*: *Entity beans*, *Session beans* and *Message – driven beans*.

Entity beans

An entity bean is a complex business entity which represents a business object that exists in the database. Its purpose is to access to data remotely over network. Each entity bean represents an object view on one record from the database and is defined by primary key. Entity beans may be shared between multiple users that use a primary key to access a particular bean. Invocations are performed synchronously. Entity beans are state full due to permanent storage background.

Entity beans introduced in EJB 3.0 specification are represented by Java Persistence API [5] entities, and they differentiate from the concept of entity beans that existed in previous EJB specifications (EJB 1.x, EJB 2.x). The EJB 1.x and 2.x entity beans must conform to a strict component model. Each bean class must implement a home and a business interface. The EJB 1.x and 2.x container requires very detail XML configuration files to map the entity beans to tables in the relational database. All these requirements are the reason why entity beans were obviated by software developers.. Introducing of entity beans as POJOs, made EJB 3.0 much more eligible an it simplified enterprise Java development with EJB.

Session beans

Session beans perform a task for a client; optionally they may implement a web service. Contrary to entity beans, session beans are not permanent and have no primary key since they are not backed by a database or other form of permanent storage.

Session beans are not shareable, as each session bean represents a single client inside the application server. To access an application that is deployed on the server, the client invokes the session bean's methods. The session bean performs work for its client, shielding the client from complexity by executing business tasks inside the server. Invocations of session beans are synchronous.

Session beans may be statefull or stateless. Statefull bean maintains its state across different method calls through its instance variables which represent the state of a unique client-bean session. As a consequence, statefull session bean can be used by one remote client at a time. Stateless bean does not hold its state, when a client invokes the methods of a stateless bean, the bean's instance variables may contain a state specific to that client, but only for the duration of the invocation. Except during method invocation, all instances of a stateless bean are equivalent, therefore stateless beans may be used by more than one remote client at a time.

Message – driven beans

Message-driven beans act as a listener for a particular messaging type, such as the Java Message Service (JMS) API. Similar to session beans, message-driven beans do not represent any data directly, however they may access any data in an underlying database. The most visible difference between message-driven beans and session beans is that clients do not access message-driven beans through interfaces. In other words, client components do not locate message-driven beans and invoke methods directly on them. Instead, a client accesses a message-driven bean through some messaging service (for example JMS). Message-driven beans are executed when a message from a client arrives on a server, this means that their invocation in asynchronous. A single message-driven bean can process messages from multiple clients.

EJB Interfaces

An interface of an enterprise bean is specified as a set of methods and attributes, using Java programming language. Unlike session beans, message-driven and entity beans do not have interfaces that define client access because they have a different programming model.

A client can access a session bean only through the methods defined in the bean's business interface. All other aspects of the bean (method implementations and deployment settings) are hidden from the client. Session beans can expose one of two kinds of interfaces:

- remote interface: represents provisions of a bean. Provides an access point for a remote client and defines the business and life cycle methods that are specific to the bean
- local interface: defines the bean's business and life cycle methods that allow only local access (a local client must run in the same Java virtual machine (JVM) as the enterprise bean it accesses)

Each session bean has to implement at least one interface (remote or local). Although it is uncommon, it is possible for an enterprise bean to allow both remote and local access. Both kinds of bean interfaces are provided interfaces. EJB does not support required interfaces of a bean.

Message-driven beans and entity beans can also define and implement some interface, but it is not obligatory.

In addition, bean class can, but is not required to implement interfaces that it defines. However, implicitly, the interface of an enterprise bean is a set of the methods it implements and its attributes.

In order to additionally specify an enterprise bean, EJB 3.0 uses metadata annotations which are inspected by service framework. The EJB 3.0 specification itself defines a wide range of annotations that cover different attributes such as transaction or security settings, object-relational mapping and injection of environment or resource references. Metadata annotations are also used to specify the bean or interface and run time properties of enterprise beans. For example, a Session bean is marked with @Stateless or @Stateful to specify the bean type, message-driven beans are marked with @MessageDriven annotation.

As an alternative to Java annotations, there are deployment descriptors which were also used in previous versions of EJB (EJB 1.x, EJB 2.x). Deployment descriptor is an XML file which can be used to override some annotations, but also for describing application level metadata.

Composition of constructs

It is important to mention that EJB does not support connection-oriented programming, but follows traditional object-oriented composition (third party can not bind EJBs, but an EJB can specify dependencies to other components). Binding of enterprise beans is performed at runtime. In addition the composition specification of EJB components is location-transparent; the run-time location of components (placed on a local or a remote node) is specified separately from the binding information. A strength of EJB is automatic composition of component-instances with appropriate services and resources that component-instances are dependent on. This includes automatic configuration of necessary implicit middleware services based on needs specified by annotations or in the deployment-descriptor (transactions, persistence and security)

Communication between beans or between client and a bean is performed using Remote Method Invocation [6], which is a Java implementation of a Remote Procedure Call. Communication between enterprise beans is managed by JVM.

5.2 Life cycle

Packaging

EJB are packaged into an EJB JAR file, the module that stores the enterprise bean. An EJB JAR file is portable and can be used for different applications. To assemble a Java EE application, one or more modules (such as EJB JAR files) are packaged into an EAR file, the archive file that holds the application.

Deployment

EJB beans are deployed in an EJB Container which is in charge of their management at runtime (start, stop, passivation or activation) and extra-functional properties (such as security, reliability, performance). The Container can hide to application programmers some of the complexities inherent in the handling of non-functional aspects in a software system, such as distribution and fault-tolerance.

5.3 Extra-functional properties

EJB is primarily aiming at industrial use and it has been designed to support component developers at an implementation level, while lacking the sufficient support for specifying or analyzing extra-functional properties.

5.4 Benefits of Enterprise Beans

For several reasons, enterprise beans simplify the development of large, distributed applications. First, the EJB Container provides system-level services to enterprise beans so the bean developer can concentrate on solving business logic problems. The EJB container, rather than the bean developer, is responsible for system-level services such as transaction management and security authorization.

Another benefit is that enterprise beans contain the application's business logic, therefore the developer of an enterprise bean client can focus on the presentation of the client. The client developer does not have to code the routines that implement business rules or access databases. As a result, the clients are thinner, a benefit that is particularly important for clients that run on small devices.

Due to the fact that enterprise beans are portable components, the application assembler can build new applications from existing beans. These applications can run on any compliant Java EE server provided that they use the standard APIs.

6. AUTOSAR

AUTOSAR is a new standardized architecture created by a partnership of a number of automotive manufacturers and suppliers. The goal of AUTOSAR is to provide a way for managing increasing complexity of vehicular embedded systems, enable detection of errors in early design phases and improve flexibility, scalability, quality and reliability of such systems [7].

AUTOSAR defines a layered software architecture consisting of five layers. First three layers, Microcotroller Abstraction Layer, ECU Abstraction Layer and Service Layer sit on top of hardware and provide a standardized and hardware-independent interface to the AUTOSAR Runtime Environment. This Runtime Environment then supports the Application Layer, the AUTOSAR Component Model.

The main goal of AUTOSAR is to provide a standard for location independence and portability of software components for the automotive industry. Thus, the component model itself is not very advanced and does not fully reflect the capabilities of current state-of-the-art models [8].

During the development process, AUTOSAR provides some levels of system modeling by giving us the ability to interconnect components using a Virtual Functional Bus (VFB). The VFB provides an abstract level of viewing all communication mechanisms provided by AUTOSAR. In this way AUTOSAR enables early system integration that is independent of the physical allocation of components. At the time of deployment, the VFB is replaced by the AUTOSAR Runtime Environment that provides implementation for selected communication mechanisms.

During deployment of a system, AUTOSAR Software Components are compiled and linked into ECU specific executable. Although this provides a more efficient systems, ti also means losing the benefits of the component-based approach during run-time.

AUTOSAR Software Component package consist of implementation and component description. Implementation of a component can be either object code, or C source code. Component description consists of operations and data that the component provides and requires, requirements that the component has on the infrastructure, resources needed by the component and information about specific implementation of the component. Because of the hardware abstraction layer provided by AUTOSAR Runtime Environment the component's implementation is independent from the hardware infrastructure, e.g. type of microcontroller or ECU.

The AUTOSAR Software Components are defined as applications which run on the AUTOSAR infrastructure. These components are atomic, meaning that one component cannot be distributed over several AUTOSAR ECUs. An exception to this is *composition*, a logical interconnection of components packaged as a new component. The components inside the composition can be distributed over several ECUs.

A special type of AUTOSAR software components are sensor/actuator components. These components encapsulate dependencies on specific sensor or actuator hardware. They are dependent on a specific sensor or actuator, but independent of the ECU.

AUTOSAR Software Components interact with each other through their well-defined ports. Services or data that a port provides or requires are defined by AUTOSAR Interfaces (which, accordingly, a port can provide or require). AUTOSAR Interfaces are described by C header files and cover only syntactical information.

Communication between components can follow either Client-Server (Request-Response) or Sender-Receiver (message passing) pattern. In case of Client-Server communication pattern providing port (server) implements operations defined by the interface, while the port that requires the interface (client) can invoke those operations. This type of communication can be either synchronous (if the client blocks its execution until the server returns a response) or asynchronous (in case the client does not block after the operation request is initiated). The Sender-Receiver pattern allows only asynchronous transfer of data. In this pattern the providing port (sender) generates the data and requiring port (receiver) has the ability to read this data. After the sender generates the data it doesn't wait or expect any response from the receiver. Type of communication is defined by the AUTOSAR interface that a port provides or requires.

Binding of AUTOSAR components is endogenous, having no separate connector entities. The connection between ports is managed by the ports themselves.

AUTOSAR allows use of compositions for sub-system abstraction. However, they are only used to group existing software components to manage complexity when designing logical system architecture [9]. They do not add any new functionality to that already defined by the components inside the composition, and do not have any binary footprint when deployed to ECU. Surface ports of a composite exposes can by explicitly defined by delegating ports of the aggregated components.

Although AUTOSAR Software Component descriptions have the ability to specify some extra functional properties, like resource (memory, CPU-time, etc.) that a software component requires, there is a lack of the capability to express the multitude of non-functional constraints, insufficient expressiveness of the interfaces [8]. In AUTOSAR, there is also a lack of ability to analyze properties of component composition, e.g. ability to guarantee that component's properties are preserved across integration, or that requirements of global properties of composed objects are meet.

7. CONCLUSION

In this paper, we have presented a framework for the classification and comparison of component models, which identifies issues related to component-based development. The survey made on three selected component models indicates that many principles comprised in the component-based approach are not always included in every component model.

The intention of this work is to increase the understanding of component-based approach by identifying the main concerns, common characteristics and differences of component models. The proposed framework does not include all the elements of all component models, however it identifies the minimal criteria for assuming a model to be a component model and it groups the basic characteristics of the models.

REFERENCES

[1] Ivica Crnković, Séverine Sentilles, Aneta Vulgarakis, Michel Chaudron, A Classification Framework for Component Models, 2008

[2] S. Sentilles, A. Vulgarakis, T. Bureš, J. Carlson, I. Crnković, A Component Model for Control-Intensive Distributed Embedded Systems, Proceedings of the 11th International Symposium on Component Based Software Engineering (CBSE2008), Springer Berlin, 2008

[3] Sun Microsystems Inc., Enterprise Java Beans 3.0, Final Relase, 2006

[4] Sun Microsystems, JavaBeans

[5] Sun Microsystems Inc., Java Persistence API

[6] Sun Microsystemc Inc., Remote Method Invocation

[7] AUTOSAR Development Partnership, AUTOSAR - Technical Overview, 2008

[8] Heinecke H., Damm W., Josko B., Metzner A., Kopetz H., Sangiovanni-Vincentelli A., Di Natale M., Software Components for Reliable Automotive Systems, 2008

[9] AUTOSAR Development Partnership, AUTOSAR - Software Component Template v3.0.1, 2008