ABSTRACT

In today’s software engineering practices, building applications from components is the ongoing trend. The style and architecture of the applications being developed has significantly changed. A set of infrastructure standards and supporting technologies has emerged during the last few years. Now a days component – based development approach is in use to understand the development of complex systems. In component – based systems, it is easier to replace parts of system with commercially available components. Component models give significant support to the user to concentrate on the requirements of the software without developing the software from scratch. The objective of this paper is to describe different component technologies currently available in the software industry and to propose the trends of the future. This paper does not intend to provide in-depth coverage of these technologies but an effort has been made to identify the component model underlying each component technology.

1. INTRODUCTION

Component - Based Software Engineering (CBSE) is one recent trend in the domain of Software Engineering (SE). One major reason why this paradigm has emerged is the need to build software by assembling reusable units, or components, as opposed to building whole application from scratch. The style and architecture of the applications being developed has significantly changed over the years. Software systems are also becoming increasingly complex and are providing ever increasing functionality. A set of infrastructure standards and supporting technologies has emerged during the last few years. In order to produce such systems cost-effectively, suppliers often use component-based technologies instead of developing all the parts of the system from scratch. The motivation behind the use of components was initially to reduce the cost of development, but it later became more important to reduce the time to market. By using components it is becoming possible to produce more functionality with the same investment of time and money.

In component - based systems, it is difficult to manage components during the life-time of a system. A system of components is usually configured once only during the build-time when known and tested versions of components are used. A software component contains some code that can be executed on certain platforms and an interface that provides the only access to the component. The code represents the operations that the component will perform when invoked and the interface tells the component user everything he needs to know in order to deploy the component. Components can be deployed in many different contexts. Components are actually like black boxes which enable the user to reuse or use them without needing to know the details of the inner structure. In other words interface of a component should provide all of the information needed by its users.

The specification of a component is defined as a specification of its interface. This must consist of a precise definition of the component’s operations and its dependencies. Actually operations and context dependencies will contain the parameters of the component. The specification of a component is useful to both component users and component developers. For users, it provide a definition of its interface like its operations and context dependencies, because, it is the only interface that is visible to users so it should be precise and complete. For developers, the specification of a component provides an abstract definition of its internal structure [1].

2. COMPONENT SPECIFICATION TECHNIQUES

In the field of software development, specifications of components used are known as syntactic specifications. This form of specification includes the specifications used with technologies such as COM, CORBA and JavaBeans.
The first two of these use different forms of Interface Definition Language (IDL), whereas the third uses the Java programming language to specify the component interfaces [1].

A component provides the implementation of a set of operations or types and an interface provides the named set of operations. Each operation has zero or more input and output parameters and a syntactic specification associates a type with each of these. The interfaces provided and required by a component are often known as the incoming and outgoing interfaces of the component respectively.

The figure 1.1 is intended to be a generic representation of the relationships between components, interfaces and operations. A component implements a set of classes, each of which implements a set of interface. Thus a component implements a set of interfaces. In traditional object-oriented specification techniques, a component is itself a class that has exactly one interface.

The primary uses of specifications are type checking of client code and as a base for interoperability between independently developed components and applications. From a syntactic viewpoint, a component can safely be replaced if the new component implements at least the same interface as the older components, or, in traditional object-oriented terminology, if the interface of the new component is a subtype of the interface of the old component. For safe substitution semantic of operations can also be changed.

**Figure 1.1  Relationships between Components and Interfaces**

The semantic information about a component’s operations is necessary to use the component effectively. This information contains the combinations of parameter values an operation accepts, an operation’s possible error codes, and constraints on the order in which operations are invoked. The several techniques for designing component based systems that included semantic specifications are Catalysis [3], Object Constraint Language (OCL) [4] and UML. In the semantic specification, a component implements a set of interfaces, each of which consists of a set of operations. Also a set of preconditions and post conditions is associated with each operation. Preconditions are the assertions that the component assumes to be fulfilled before an operation is invoked and Post conditions are the assertions that the component guarantees will hold just after an operation has been invoked, provided the operation’s preconditions were true when it was invoked. An operation’s precondition and post condition often depends upon the state maintained by the component. An invariant is a predicate over the interface’s state model that will always hold. The component specification must include a set of inter-interface conditions, which are predicates over the state models of the component’s entire interface. Figure 1.2 presents a model of a state of a component with each interface, to allow the semantics of an interface’s operations to be specified.

The specification of extra functional properties of software components includes structural properties like governing how a component can be composed with other components, extra functional properties such as performance, capacity and environment assumptions specifying relationships among similar or related components.
It is not realistic to expect specifications to be complete with respect to all such properties because software components are not delivered with complete and sufficient specifications because developers are likely to discover new kinds of dependencies as they attempt to use independently developed components together, specifications should be extensible.

The concept of credentials is proposed to satisfy the above outlined requirements. A credential is defined as a combination of Attribute, Value and Credibility, where Attribute is a description of a property of a component, Value is a measure of that property and credibility is a description of how the measure has been obtained. Figure 1.3 specifies the extra functional properties of software components.

The specification of extra functional properties of software components is still an open area of research. Therefore it is very difficult to say that the model in figure 1.3 is a generic model.

3. COMPONENT MODELS

A component model could be described as 'a specification for how a component must be constructed in order to make it work in a correct manner'. There are lots of component models that have been introduced the last few years. Most component developers are focused on three standards: CORBA [5], COM [6], and JavaBeans [7]. Currently
available component models also include COM+ [6], .NET [6], and the Open Service Gateway Initiative (OSGI) [8]. There are a lot of technical details that concern these component models but we are just giving a brief understanding of these component models [9].

3.1. COMMON OBJECT REQUEST BROKER ARCHITECTURE (CORBA)

The Object Management Group (OMG) is the founder of the Common Object Request Broker Architecture (CORBA) [5] with a view to establish industry guidelines and object management specifications that provide a common framework for distributed object and component-based application development.

CORBA was designed by the OMG in order to make it possible to communicate between multiple languages on multiple platforms. CORBA is the basic architecture for communication between heterogeneous objects on heterogeneous platforms [10]. According to Brown [11] ‘There are three major aspects of CORBA:

- The OMG’s Interface Definition Language (IDL), which describes how business functionality is packaged for external access through interfaces.
- The CORBA component model describing how components can make requests of each other’s services.
- The Internet Interoperability Protocol (IIOP), which allows different CORBA implementations to interoperate.

In CORBA, every object instance has its own unique object reference, an identifying electronic token. Clients use the object references to direct their invocations, identifying to the ORB the exact instance they want to invoke. The client acts as if it's invoking an operation on the object instance, but it's actually invoking on the IDL stub which acts as a proxy. Passing through the stub on the client side, the invocation continues through the ORB (Object Request Broker), and the skeleton on the implementation side, to get to the object where it is executed [5].

3.2. COMPONENT OBJECT MODEL (COM)

Microsoft is the founder of Common Object Model (COM) (1995). It specifies how to build components that can be dynamically interchanged. ‘COM provides the standard that components and clients follow to ensure that they can operate together’ [6] [12]. Microsoft has extended COM to support Windows NT cross platform communication. As a result of that the Distributed Component Object Model (DCOM) has been presented.

Microsoft has defined its strategy of component-based applications to consist of two parts. The first is its packaging and distribution services, DCOM, providing inter-component communication. The packaging and distribution services consist, according to Brown [11], of three aspects.

- Microsoft Interface Language (MIDL), which describes how business functionality is packaged for external access through interfaces.
- Requests, the COM component model describing how components can make requests of each other’s services.
- Transparency, the DCOM additions to COM providing support for location transparency of component access across a network.

Microsoft provides additional component infrastructure services via two products.

- Security, the Microsoft Transaction Service (MTS), which provides security and transaction management services.
- Asynchronous communication, The Microsoft Message Queue (MSMQ), which provide applications.

COM is a platform-independent, distributed, object-oriented system for creating binary software components support for asynchronous communication between component-based that can interact. COM+ is the next evolution of Microsoft Component Object Model (COM) and Microsoft Transaction Server (MTS). It handles many of the resource management tasks that developers previously had to program, such as thread allocation and security. It can be used to develop enterprise-wide, mission-critical, distributed applications based on the Microsoft Windows platform.

3.3. JAVABEANS COMPONENT MODEL

The JavaBeans component model was proposed by Sun Microsystems in 1997. The main objective of JavaBeans APIs is to define a software component model for java so that third party can create and ship java components that can be composed together into applications by end users. “A JavaBean is defined, as a reusable software component, that can be manipulated visually in a builder tool and can be usable at run-time within generated application” [7]. JavaBeans is one of the few models in which the component is explicitly designed to interact in two different contexts: at composition time, within the builder tool, and at execution time, in the run-time environment.

A second distinct component model, Enterprise Java Beans (EJB) [7], from Sun Microsystems, is a component-architecture for server-side components used to build distributed systems with multiple clients and servers. A Java Bean is a reusable component which supports persistency and can interact across all platforms supported by Java. EJB provides support for transactions and security over a neutral object communication protocol, which gives the user the opportunity to implement the application on top of a protocol of choice. EJB is part of the
Java 2 Platform Enterprise Edition (J2EE) [13] which includes many other technologies such as remote method invocation (RMI), naming and directory interface (JNDI), database connectivity (JDBC), server pages (JSP) and messaging services (JMS).

A Java Bean is a special case of an ordinary Java class. To make a Java Bean an Enterprise bean the Java Bean must conform to the specification of EJB by implementing and exposing a few required methods. These methods enable the EJB container to manage beans in a uniform way for creation, transactions etc. There are two different kinds of enterprise beans: session and entity beans. Session beans live as long as the client code which calls it. They represent the business process and are used to implement business logic, business rules and workflow. Entity beans model data and are often used by session beans to represent the data they use.

EJB is designed to interact with CORBA implementations and access CORBA objects transparently. There is also a bridge between COM and EJB which can be used to make systems even more open. The adapter pattern [13] can normally be used to implement various bridges between object models.

3.4 .NET COMPONENT MODEL

.NET [6], the latest component model from Microsoft, emphasizes on language interoperability and introspection. It defines an internal language Microsoft Intermediate Language (MSIL), which is very similar to Java Byte Code and it’s interpreter with introspection capabilities: the Common Language Runtime (CLR), which is very similar to a Java Virtual Machine.

.NET technology provides the ability to quickly build, deploy, manage, and use connected, security-enhanced solutions with Web services. .NET-connected solutions enable businesses to integrate their systems more rapidly and in a more agile manner and help them realize the promise of information anytime, anywhere, on any device.

.NET represents the programming language approach for component programming. It means that the program contains the information related to the relationships with other components and that the compiler is responsible for generating the information needed at execution.

.NET model includes the visibility control, which allows assemblies and their modules to be local to an application and thus different dynamic link libraries (DLLs) with the same name can run simultaneously. Each assembly keeps track of versioning information about itself and about the assemblies it depends on, provided either in the form of attributes in the code source or as command-line switches when building the component descriptor.

Version control is supervised by the dynamic loader, which selects the right version, based on the assembly’s version information and on a set of default rules. Both at the machine-wide level and at each application level, the default rules can be altered using XML configuration files. These features significantly improve application packaging and deployment control with respect to the traditional Windows application deployment.

3.5. THE OSGI COMPONENT MODEL

The Open Services Gateway Initiative (OSGI) created the specification of the OSGI services platform to ease the deployment and management of services in a coordinated way. It was founded in 1999 with the objective of creating “open specifications for the delivery of multiple services over wide area networks to local networks and devices.” [8].

The OSGI emphasis is on a lightweight framework that can be executed in low-memory devices. Actually, the OSGI targets products such as set-top boxes, cable modems, routers, consumer electronics and so on. The OSGI relies on java to ensure portability on different hardware.

A very important aspect of the paradigm of OSGI is that services may appear or disappear at any time during the execution of an application i.e. it has been tailored to support dynamic evolution of the system architecture. Components can be downloaded, updated and removed dynamically, without even stopping the system. The OSGI relies on java to ensure portability on different hardware. It also allows for remote administration of the system via the network.

4. DEPLOYMENT IN COMPONENT MODELS

To partially achieve the static service integration, component models such as CORBA Component Model (CCM) and Enterprise Java Beans (EJB) have emerged. Those standards specify how some services can be statically plugged into components. One of the most important contributions in component models is to separate application programming from deployment. Indeed, deployment descriptors allow component programmers to give information about which services to use. Then the deployer has to customize the deployment descriptor in order to adapt the component to the specificity of the runtime environment (transaction, persistency, security, database support, etc.). According to a deployment descriptor, generators provided by platforms generate adequate interposition code. So, the integration of services by the deployer is basically done through a parameter file. As the definition of the services is integrated into the platform, their evolution and composition is handled by the platform. Consequently,
we could suppose that the deployer does not have to deal neither with the application code nor with the generated code.

5. DEPLOYMENT DRAWBACKS IN COMPONENT MODELS

However, the code generators support only services provided by the platform. For using new services like replicated EJB for example, the deployer must either modify the generated code or specialize the generator if it is open-source [14], [15]. So, it is a difficult task, because service calls must be plugged either in the generated code or in the code to generate. This makes maintenance, evolution and service composition quite impossible to manage. Moreover, as the code integration is predefined, the deployer has no high level way to adapt the code generation nor to control the composition of existing services. This is needed for example in order to modify or to trace access to persistent data. Finally, these component models do not allow neither dynamic service integration nor dynamic customizations of a single component instance.

6. RESEARCH DIRECTIONS

The component – based approach opens up several new areas for research. A component – oriented world calls for determining methodologies that can allow component builders and users to agree on the tasks to be carried out by a given component. Research in this field suggests that a component must be endowed with a series of additional information (apart from that making up its interface) that allows it, in a certain sense to be formed semantically. This information can be used by the user in the phases of different development cycles.

More specifically, in the development of a component based application we must initially focus on identifying that or those components that provide the basic functionalities. That is to say, that we must elicit the functional and non-functional requirements for these basic components. When candidate components are found, we can test them, against the specified requirements, and choose the best for our objectives. After having identified the first basic components, we can go toward the expansion of the application functionalities, in second directions and look for new components. The specifications for the new searched components must include the features of the components already acquired. This cycle is repeated until all the application functionalities are covered.

The specification of extra functional properties of components is still an open area of research, and it is uncertain what impact it will have on the future of software component specification.

7. CONCLUSION

Component models and technologies addresses primarily execution and technological issues, including interoperability, heterogeneity, distribution, efficiency, fast life cycle, deployment, and multiprogrammer development. These technologies often make the distinction between the component model specification and the implementation of the standards. They also provide the distinction between the in-house developed component, and the component available in the market place. In current practices, component specification techniques specify techniques only syntactically and the use of UML and OCL to specify components represents a step toward semantic specifications.

REFERENCES


