Software Component Models

Kung-Kiu Lau and Zheng Wang

Abstract—Component-based development (CBD) is an important emerging topic in software engineering, promising long-sought-after benefits like increased reuse, reduced time to market, and, hence, reduced software production cost. The cornerstone of a CBD technology is its underlying software component model, which defines components and their composition mechanisms. Current models use objects or architectural units as components. These are not ideal for component reuse or systematic composition. In this paper, we survey and analyze current component models and classify them into a taxonomy based on commonly accepted desiderata for CBD. For each category in the taxonomy, we describe its key characteristics and evaluate them with respect to these desiderata.

Index Terms—Software components, software component models, component life cycle, component syntax, component semantics, component composition.

1 Introduction

In software engineering, component-based development (CBD) [1], [2], [3] is an important emerging topic. CBD aims to compose systems from prebuilt software units or components. A system is developed as a composite of subparts rather than a monolithic entity. Such an approach reduces production cost by composing a system from existing components instead of building it from scratch. It also enables software reuse since components can be reused in many systems. Thus, CBD promises the benefits of increased reuse, reduced production cost, and shorter time to market, which have long been sought after by the software industry. To realize these benefits, it is crucial to have components that are easy to reuse and composition mechanisms that can be applied systematically. For systematic composition, composites, as well as their composition, must be well defined.

The cornerstone of any CBD methodology is its underlying *software component model* [4], [5], which defines what components are, how they can be constructed, how they can be composed or assembled, and how they can be deployed, as well as, ideally, how we can reason about all these operations on components so that quality certification may be tractable. Current component models can largely be divided into two categories [4], [5]: 1) models where components are *objects*, as in object-oriented programming, and 2) models where components are *architectural units*, as in software architectures [6], [7]. Exemplars of these categories are Enterprise JavaBeans (EJB) [8], [9] and architecture description languages (ADLs) [10], [11], respectively. In general, current models are not ideal for fulfilling CBD's promise [4], [12] because they use components that are not easy to reuse and/or composition mechanisms that are not well defined and/or are hard to apply systematically. In this paper, we will show why this is so by analyzing these models.

• *The authors are with the School of Computer Science, University of Manchester, Manchester M13 9PL, UK.*
E-mail: {kung-kiu, zw}@cs.man.ac.uk.

Manuscript received 4 Aug. 2006; revised 27 May 2007; accepted 26 June 2007; published online 12 July 2007.
Recommended for acceptance by R. Taylor.
For information on obtaining reprints of this article, please send e-mail to: tse@computer.org, and reference IEEECS Log Number TSE-0186-0806.
Digital Object Identifier no. 10.1109/TSE.2007.70726.

Our analysis is based on an idealized component life cycle [4], [5] such that, if components were constructed and composed in the manner defined in the life cycle, then we should be able to meet the success criteria for CBD. The definition of the idealized component life cycle is based on the widely accepted desiderata for CBD's success [2], [3], [13], [14]. These are given as follows: First, components should be preexisting reusable software units which developers can reuse to compose software for different applications more quickly than writing all of the code from scratch for each application. Second, components should be produced and used by independent parties. That is, component developers need not be the same people as component customers, such as system developers. This is important for ensuring that components are truly reusable by third parties. Third, it should be possible to copy and instantiate components so that their reuse can be maximized, both in terms of code reuse and in terms of the components' scope of deployment. Fourth, components should be composable into composite components which, in turn, can be composed with (composite) components into even larger composites (or subsystems), and so on. Composition means not only reuse but also a systematic approach to system construction.

However, to achieve these desiderata, it would be crucial to identify the key prerequisites for component models. We believe that a good starting point for this endeavor is a study of current component models. In this paper, we present such a study. In the study, we survey the 13 major current component models, analyze them, and classify them into a taxonomy with four categories. For each category, we describe the key characteristics of the models and evaluate them with respect to the desiderata for CBD by the idealized component life cycle. This paper thus distills and presents a comprehensive knowledge and analysis of current software component models.

This paper is organized as follows: In Section 2, we define and explain the elements of software component models in general and define the idealized component life cycle. In Section 3, we survey and categorize the 13 current major component models according to these elements. In Section 4, we present a taxonomy based on the idealized component life cycle.

2 Software Component Models

In this section, we present a reference framework for software component models.

Currently, there is no universally accepted terminology for CBD [13]. In particular, there are no standard criteria for what constitutes a software component and current major component technologies do not all use the same kind of components. However, at present, the more widely adopted definitions of components tend to be given without software component models.

For example, the widely accepted definition by Szyperski et al. [3]: "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 parties," is not given in the context of a component model, and neither is the following definition by Meyer [14]:

"A component is a software element (modular unit) satisfying the following conditions:

1. It can be used by other software elements, its "clients."
2. It possesses an official usage description, which is sufficient for a client author to use it.
3. It is not tied to any fixed set of clients."

At the same time, standard component models like EJB, COM [15], and CCM (Corba Component Model) [16] adopt slightly different component definitions from "standard" ones like Szyperski et al.'s and from each other.

For uniformity, in this section, we present a reference framework for component models which defines and explains terms of reference that we will use throughout this paper. The definitions are general and should therefore be universally applicable. Furthermore, by and large, they follow (what we perceive as) consensus views and therefore should not be contentious or controversial.

A *software component model* is a definition of

- the *semantics* of components, that is, what components are meant to be,
- the *syntax* of components, that is, how they are defined, constructed, and represented, and
- the *composition* of components, that is, how they are composed or assembled.

This is exemplified by Heineman and Councill's definition of a component [2]:

> A [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.

In this paper, we will examine the elements of 13 software component models: JavaBeans [17], EJB, COM, .NET [18], CCM, Web Services [19], Koala [20], [21], KobrA [22], SOFA [23], [24], Acme-like ADLs [10], UML 2.0 [25], [26], PECOS [27], [28], and Fractal [29], [30], [31], [32]. By necessity, we have to omit other models, but we believe that the chosen set is representative of the main categories of current models. An important criterion for inclusion is that a model should have reached a mature stage of development, with sufficient documentation available. Also, by necessity, we have to focus on representative models of the categories rather than all of the models.

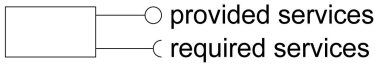

Fig. 1. A software component.

2.1 The Semantics of Software Components

A generally accepted view of a software component is that it is a software unit with *provided services* and *required services* (Fig. 1). The provided services are operations performed by the component. The required services are the services needed by the component to produce the provided services. The interface of a component consists of the specifications of its provided and required services. It should specify any *dependencies* between its provided and required services. To specify these dependencies precisely, it is necessary to match the required services to the corresponding provided services.

Note that, in the literature, a component can often have multiple interfaces, with each interface being a different set of services. Here, we have used a single interface as a collective entity for all such interfaces.

In current component models where components are objects in the sense of object-oriented programming, the methods of these objects are the provided services. Because they cannot specify their required services, these objects are usually hosted in an environment, for example, a container, which handles access to and interactions between components. As a result, the semantics of these components is an enhanced version of that of the corresponding objects. In particular, they can interact with one another via mechanisms provided by the environment.

For example, in JavaBeans and EJB, although, syntactically, they are both Java classes, JavaBeans and EJB are different semantically. Semantically, a JavaBean is a Java class that is hosted by a container such as BeanBox [33]. JavaBeans interact with one another via adapter classes generated by the container. Adapter classes link beans via events. An EJB, on the other hand, is a Java class that is hosted and managed by an EJB container provided by a Java 2 Enterprise Edition (J2EE) server [34] via two interfaces, the home interface and the remote interface, for the enterprise bean. Enterprise beans interact directly via method delegation within the EJB container and, through their remote and home interfaces, with remote clients, also via method delegation.

In current component models where components are architectural units, services are represented as ports. Ports on different units can be linked by connectors and, in a linked pair of ports of connected units, the port of one unit represents not only the provided service of that unit but also the required service of the other unit and vice versa. In some models, for example, UML2.0 and CCM, ports for provided services are distinguished from those for required services.

2.2 The Syntax of Software Components

Once the semantics of components has been fixed in a component model, components can be defined and constructed. The definition of components requires a *component definition language* which may be distinct from the implementation language, that is, programming language, for components. Clearly, for a given definition language, components can be implemented in different programming languages. Therefore, we refer to the syntax of components as the syntax of the component definition language. In a component model, this language must be

specified, whereas the implementation language(s) may be left open.

In current component models where components are classes, the definition language for components coincides with the implementation language. For example, in both JavaBeans and EJB, a component (respectively, a bean and an enterprise bean) is defined as a Java class. In current models where components are objects, the definition language is a programming language with interface definition language (IDL) mappings. For example, COM uses Microsoft IDL [15] and CCM uses Object Management Group (OMG) IDL [35].

On the other hand, in component models where components are architectural units, the definition language is an ADL or an ADL-like language. In models with pure ADLs being definition languages, no implementation language is specified. Examples are Acme [36] and UML2.0. In models with ADL-like languages being definition languages, the implementation language is different from the definition language. For example, in Koala, the definition language is CDL and the implementation language is C. In PECOS, the definition language is CoCo and the implementation language is C++ or Java.

2.3 The Composition of Software Components

In CBD, composition is a central issue since components are supposed to be used as building blocks from a repository and assembled or plugged together into larger blocks or systems. Typically, components are hosted by servers and, so, composition takes place on the server side. Client applications are built from server-side components, as exemplified by Web service orchestration [37].

In order to define composition, we need a composition language, for example, [38]. The composition language should have suitable semantics and syntax that are compatible with those of components in the component model. In most of the current component models, there is no composition language. JavaBeans, EJB, COM, .NET, and CCM have no composition languages. Web Services are composed by orchestration, typically using a workflow language like Business Process Execution Language (BPEL) [39]. Koala uses connectors as glue code for composition. KobrA and UML2.0 use the UML notation. ADLs are, of course, formal composition languages [40] and PECOS has an ADL-like composition language CoCo (which doubles as the CDL).

In order to reason about composition, we need a composition theory (see the discussion in [41]). Such a theory allows us to calculate and, thus, predict the result of applying a composition operator to components. Current component models tend not to have composition theories, even those with a composition language.

Composition can take place during different stages of the *life cycle* of components [42], namely, 1) the *design* phase, during which components are designed, defined, and constructed in the source code, and possibly compiled into binaries, 2) the *deployment* phase, during which binaries of components are deployed into the target execution environment for the system under construction, and 3) the *runtime* phase, during which component binaries are instantiated with data and these instances are executed in the running system.

This life cycle is distinct from but embedded in the system construction process that we have assumed. By the deployment phase of the component life cycle, we assume that the system has already been constructed in a component-based manner, with placeholders for the selected components.

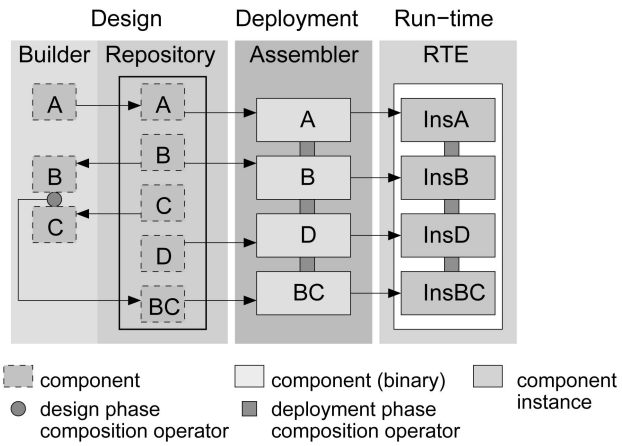

Fig. 2. An idealized component life cycle.

Of course, it is possible to consider a different system construction process, in particular one where the system can be dynamically constructed or reconfigured at runtime, for example, [43]. We choose not to follow this process, primarily because it demands binary-level composition for component instances, for which it is hard to define meaningful composition operators other than glue code.

Ideally, composition should be possible in both the design and the deployment phases of the component life cycle while the system is being constructed. Composition means component reuse and, therefore, composition in both phases will maximize reuse. It also means design flexibility in the sense that the deployed components, in particular composite components, can be designed, by composition, in either phase.

Accordingly, we have defined an idealized component life cycle [5], [12] and the kind of composition that is possible and meaningful in both phases.

2.3.1 An Idealized Component Life Cycle

The idealized life cycle is depicted in Fig. 2. This is based on the commonly accepted desiderata of CBD [2], [3], [13], [14], as described in Section 1. That components should be preexisting reusable software units necessitates the use of a repository in the design phase. That components should be produced and used by independent parties requires the use of builder and assembler tools that can interact with a repository in the design and deployment phases, respectively. That it should be possible to copy and instantiate components means that components should be distinguished from their instances and, therefore, we distinguish the design and deployment phases from the runtime phase. That it should be possible to compose components into composite components, which, in turn, can be composed with (composite) components into even larger composites (or subsystems), and so on, requires that composites can be deposited in and retrieved from a repository, just like any components. All current component models attempt to meet these criteria, with varying degrees of success, as we will show later.

2.3.2 Composition in the Design Phase

In the design phase, components have to be constructed, cataloged, and stored in a repository. The main requirement for a repository is that it should provide storage and management for depositing and retrieving the components. A repository could be a registry or a directory, but it must

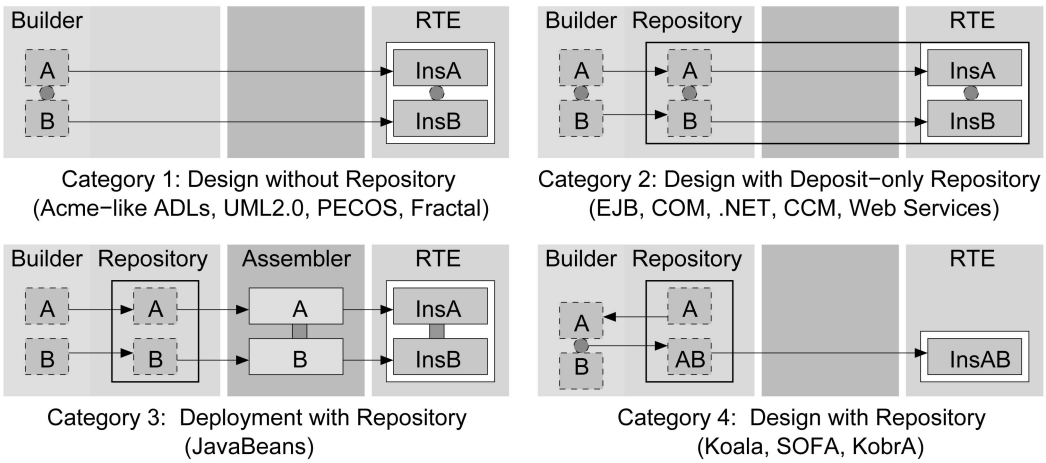

Fig. 3. Categories based on composition.

store and catalog the deposited components with suitable identities and provide a search facility for retrieving these components.

In the idealized life cycle, components in the repository are in source code or they may have been compiled into binary. They can be retrieved and composed into a composite that is, in turn, deposited into the repository. A new composite has a unique identity which enables the composite to be retrieved subsequently.

A *builder* tool (Fig. 2) can be used to 1) construct new components and then deposit them in the *repository*, for example, A in Fig. 2, and 2) retrieve components from the repository, compose them, and deposit them back in the repository. For example, in Fig. 2, B and C are composed into a composite BC that is deposited in the repository.

For example, in JavaBeans, the container provides a repository for beans, for example, ToolBox in the BDK (Beans Development Kit) [33], [44], which are stored as JAR files. However, no composition is possible in the design phase and, therefore, no composite beans can be formed.

2.3.3 Composition in the Deployment Phase

In the deployment phase, components have to be retrieved from the repository and compiled to binary code. These binary components can be composed into a system that will be executable once the binaries have been instantiated with data.

An *assembler* tool (Fig. 2) can be used to retrieve components from a repository and, if necessary, compile them into binary code, then assemble them into a system. For example, in Fig. 2, binaries of A, B, D, and BC are created and composed into a system. In the runtime phase, the system is executable in the runtime environment once the binaries of A, B, D, and BC are instantiated with data.

The result of the deployment phase composition is a whole system in binary code; thus, this is the end result of system design and implementation and, therefore, systems should not be stored in the repository.

For example, in JavaBeans, in the deployment phase, bean instances are created from beans in the repository and these can be composed (linked) graphically by using the builder tool.

Finally, it is worth pointing out that, although they do not figure explicitly in Fig. 2, *containers* play an important part in the component life cycle. Indeed, they are considered to be implementations of component models in that they provide an execution environment for components and their assemblies. With reference to Fig. 2, a container may be a repository (in the design phase) and a runtime environment, for example, the EJB container. Alternatively, it may be an assembler and a runtime environment, for example, the BeanBox in JavaBeans.

3 Current Software Component Models

In this section, we survey existing software component models. Rather than discussing them in some random sequential order, we survey the models in meaningful groups. Clearly, we can group the models according to component semantics, syntax, or composition. However, since composition is a key issue in CBD, we focus on groupings based on composition. We present four such categories that cover the 13 models and, for each category, we explain a representative model in detail, with an example, but only outline the other models (in the Appendix) for completeness.

To define categories based on composition, we consider composition in an ideal life cycle, as described in Section 2.3 (Fig. 2), that provides a basis for comparing and classifying composition in existing component models. For instance, some component models do not have composition in the design phase, whereas some models do. Some have composition in the deployment, whereas some do not. Thus, many categories are possible. Fig. 3 gives four categories that cover all 13 major existing component models.

In Category 1, in the design phase, there is no repository. Therefore, components are all constructed from scratch. Composition is possible. In the deployment phase, no new composition is possible: The composition of the component instances (in the runtime phase) is the same as that of the components in the design phase. All simple Acme-like ADLs belong to this category, as do models such as UML2.0, PECOS, and Fractal, which are based on Acme-like ADLs. This category can be described as Design without Repository.

In Category 2, in the design phase, new components can be deposited in a repository but cannot be retrieved from it. Composition is possible, that is, composites can be formed, but composites cannot be retrieved from the repository because they do not have identities of their own. In the deployment phase, no new composition is possible: The composition of the component instances (in the runtime phase) is the same as that of the components in the design

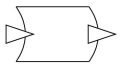

Fig. 4. An Acme component.

phase. This category includes EJB, COM, .NET, CCM, and Web Services. It can be described as Design with Deposit-only Repository.

In Category 3, in the design phase, new components can be deposited in a repository but cannot be retrieved from it. Composition is not possible in the design phase, that is, no composites can be formed and, so, no composites can be deposited in the repository. In the deployment phase, components can be retrieved from the repository and their binaries can be formed and composed. The sole member of this category is JavaBeans. This category can be described as Deployment with Repository.

In Category 4, in the design phase, new components can be deposited in a repository and components can be retrieved from the repository. Composition is possible and composites can be deposited in the repository. In the deployment phase, no new composition is possible: The composition of the component instances (in the runtime phase) is the same as that of the components in the design phase. Koala, SOFA, and KobrA belong to this category. This category can be described as Design with Repository.

3.1 Category 1: Design without Repository

This category includes all simple Acme-like ADLs, UML2.0, PECOS, and Fractal. Acme is the representative example in this category since all the of other models are based on Acme-like ADLs.

In Acme, as in any ADL [10], [11], a component is an *architectural unit* (a box) that represents a primary computational element and data store of a system (Fig. 4). The interface of a component is defined by a set of ports, through which the component's functionalities are exposed. Each port identifies a point of interaction between the component and its environment. A port can have different roles, such as sink (receive) and source (send).

Components and their ports can have different types. These types can be defined in a family of type definitions which define the design vocabulary of a system. Both functional and extrafunctional attributes of a component can be specified by property types defined in that component.

Consider a simple bank system which has just one ATM that serves one bank consortium with two bank branches, Bank1 and Bank2. In Acme [36], the bank system can be implemented using the ATM, BankConsortium, Bank1, and Bank2 components, as shown in Fig. 5. Components ATM, BankConsortium, Bank1, and Bank2 specify their ports and Bank1 and Bank2 specify their bank identity properties.

In Acme, again, as in any ADL, components are composed of *connectors* (lines) that mediate the communication and coordination activities among components (boxes; see Fig. 6). A connector can connect two or more components. Similarly to components, connector interfaces are defined by sets of roles. Each role defines a participant of the interaction represented by the connector. A connector may have multiple interfaces by using different types of roles. The types of connectors and their roles may be defined as a family. Both connector and role attributes can be specified by property types defined in that connector.

In Acme, there is no repository for components. In the design phase, components and connectors have to be constructed from scratch. The system architecture is then

```
Component ATM = {
    Port send;
}
```

```
Component BankConsortium = {
    Port receive;
    Port send;
}
```

```
Component Bank1 = {
    Port receive;
    Property bankid : String = "Bank1";
}
```

```
Component Bank2 = {
    Port receive;
    Property bankid : String = "Bank2";
}
```

Fig. 5. Examples of Acme components.

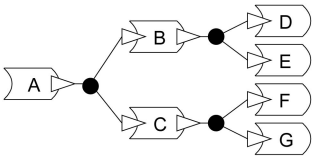

Fig. 6. Acme connectors.

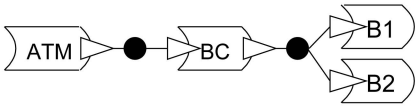

Fig. 7. Architecture of bank system in Acme.

created by connecting the components, possibly using a visual builder tool, for example, AcmeStudio [45] for Acme. For example, in Acme, the architecture for the bank system is defined graphically in Fig. 7, where BC stands for BankConsortium and B1 and B2 stand for Bank1 and Bank2, respectively. The components are those specified in Fig. 5 and the connectors are specified in Fig. 8. The specification for the architecture is defined in Fig. 9. The connections are defined by a set of attachments. Each attachment represents an interaction between a port (of a connector) and some role of a connector.

In ADLs, in general, after the design phase comes the runtime phase, that is, there is no deployment phase (Fig. 3). However, the implementation of components and connectors is not always specified. Therefore, the implementation of the architecture is a separate activity from its design and has to be done manually somehow in some programming language. The implementation is then executed in the runtime environment for the chosen programming language.

In Acme, system specifications constructed in the design phase can be compiled to a system in a programming language directly, provided that the code for the components and connectors is available in that language. For example, in Acme, architectures can be specified in ArchJava [46], [47] and can be compiled into Java, provided that the Java code for the components and the connectors is available [48]. The compiled system is then run on the Java Virtual Machine.

The other models in this category, UML2.0, PECOS, and Fractal, are outlined in Appendix A.1. In these models, the builder is a programming environment which is often a visual builder tool like AcmeStudio for the Acme ADL, whereas the runtime environment is usually that for the implementation language, for example, the Java Virtual Machine for Fractal.

```
Connector ATMtoBankCon = {
        Role request;
        Role produce;
}

Connector BankContoB1 = {
        Role request;
        Role produce;
}

Connector BankContoB2 = {
        Role request;
        Role produce;
}
```

Fig. 8. Examples of Acme connectors.

```
System BankSys = {
    Component ATM = {. . .}
    Component BankConsortium = {. . .}
    Component Bank1 = {. . .}
    Component Bank2 = {. . .}
    Connector ATMtoBankCon = {. . .}
    Connector BankContoB1 = {. . .}
    Connector BankContoB2 = {. . .}
  Attachments {
    ATM.send to ATMtoBankCon.request;
    ATMtoBankCon.produce to BankConsortium.receive;
    BankConsortium.send to BankContoB1.request;
    BankContoB1.produce to Bank1.receive;
    BankConsortium.send to BankContoB2.request;
    BankContoB2.produce to Bank2.receive;
} }
```

Fig. 9. Example of component composition in Acme.

3.2 Category 2: Design with Deposit-Only Repository

This category includes EJB, COM, .NET, CCM, and Web Services. The representative example is EJB.

In EJB [49], [50], a component is an *enterprise bean*, which is a Java class that is hosted and managed by an EJB container provided by a J2EE server. An EJB container manages the execution of enterprise beans and handles security, transaction management, Java Naming and Directory Interface (JNDI) lookups, and remote connectivity.

The Java class for an enterprise bean defines the methods of the bean. It must be accompanied by code for two interfaces, the *home interface* and the *remote interface*, that the EJB container uses to manage and run the bean. These interfaces expose the capabilities of the bean and provide all of the methods needed for client applications to access the bean. The home interface represents the life-cycle methods of the bean, such as *create*, *destroy*, and *locate* a bean instance, whereas the remote interface represents the tasks performed by the bean.

There are three different kinds of enterprise beans [49]: *entity beans*, *session beans*, and *message-driven beans*. Entity beans model business data: An entity bean represents a persistent business object whose data is stored in a database. Session beans model business processes: A session bean represents a business process or an agent that performs a service. Message-driven beans model message-related business processes: A message-driven bean represents a business process that can only be triggered by receiving messages from other beans.

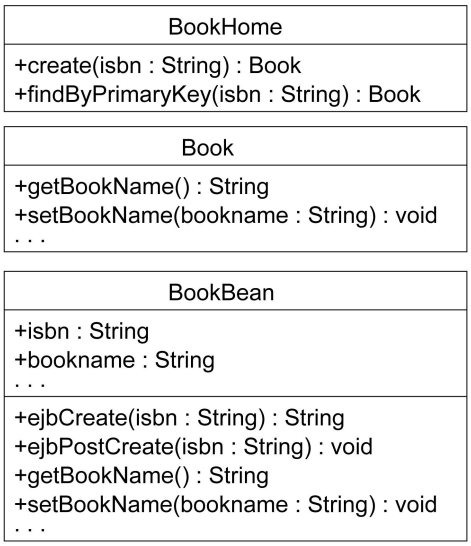

Fig. 10. Example of entity bean.

As an example, consider a bookstore that wishes to maintain a database of its book stock. Suppose books can be purchased and have their details added to the database by any shop assistant. Then, the bookstore can use a set of enterprise beans to implement a system that allows multiple clients to access and update the database.

An entity bean, like the one in Fig. 10, can represent the table of books in a database. This entity bean consists of one class and two interfaces: 1) BookBean is the Java class that defines the methods of the entity bean, 2) BookHome is the home interface of the entity bean, and 3) Book is the remote interface of the entity bean. Each instance of this entity bean represents a row of the table of books in a database. Methods defined in the home interface BookHome are life-cycle methods: "create" and "findByPrimaryKey." Thus, the home interface helps create an instance of this entity bean and locate an instance of BookBean by its primary key (isbn). Methods in BookBean correspond to methods defined in both the home interface BookHome and the remote interface Book.

A session bean, like the one in Fig. 11, can be used to add details of a set of books into the table of books in the database. This session bean consists of the class BookStoreBean, the home interface BookStoreHome, and the remote interface BookStore (and its helper class Books). The only method defined in the home interface BookStoreHome of BookStoreBean is a life-cycle method: "create." The only method defined in the remote interface BookStore of BookStoreBean is a task performed to add details of a set of books into the database: "addBook." As in an entity bean, methods in a session bean correspond to methods in its home and remote interfaces.

In general, enterprise beans are Java classes and interfaces and bean composition is by delegation of method calls. In the design phase, enterprise beans can be constructed in a Java programming environment such as Eclipse [51] and their JAR files containing the enterprise bean implementation class, the home and remote interfaces, and the deployment descriptor are deposited in an EJB container on a J2EE server that is the repository of enterprise beans.

The EJB container in Fig. 12 shows an example of the design phase composition of two session beans, SessionBeanA and

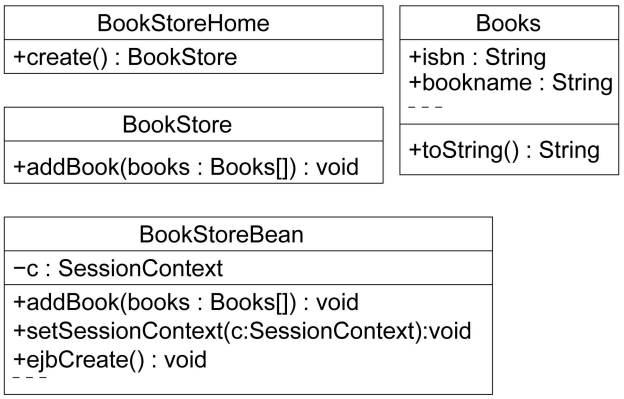

Fig. 11. Example of session bean.

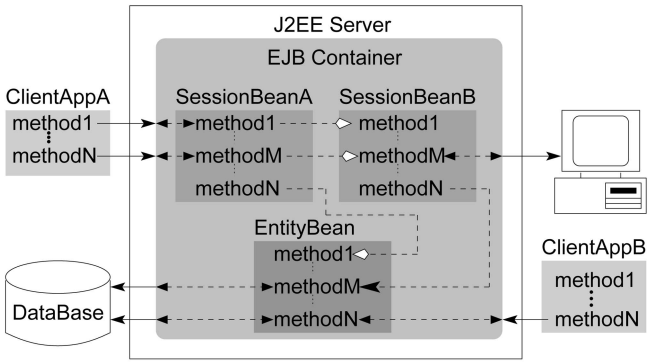

Fig. 12. Composition of Enterprise JavaBeans.

SessionBeanB, and one entity bean, EntityBean. SessionBeanA takes client calls, and EntityBean writes to the database.

In the design phase, although bean composition can produce composite beans "on the fly" in the EJB container, it is not possible to retrieve such a bean and reuse it as a single bean for further composition. However, individual beans in the container in the design phase are reusable in the sense that every bean is accessible to clients in the deployment phase, regardless of what other beans it is linked to, and, whatever method a client calls, the correct links will automatically be followed.

Thus, although it is a repository for enterprise beans that supports the design phase composition, the EJB container does not support the storage or retrieval of composite components as identifiable units. The reason is that the container is also the execution environment for beans.

In EJB, after the design phase comes the runtime phase. Thus, no new composition is possible after the design phase (for example, see [52]). In the runtime phase, bean instances are created and executed in the EJB container.

Consider the bookstore example again. In the design phase, an assembly of the session bean BookStoreBean and the entity bean BookBean is deposited into the EJB container. The composition is defined by BookStoreBean calling the methods "create" and "set" defined in the home interface BookHome and the remote interface Book of BookBean (Fig. 13).

In the runtime phase, this system is looked up and instantiated. In this instance of the system, the BookStoreClient calls the method "create" defined in the home interface BookStoreHome of the BookStoreBean, which returns an instance of BookStore, the remote interface of

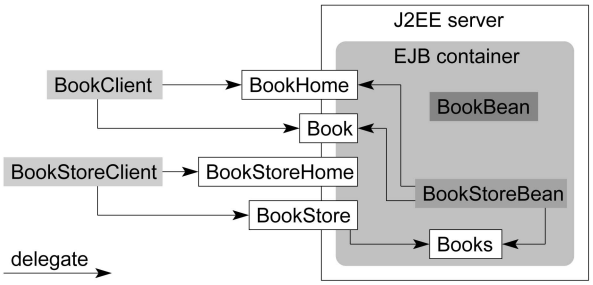

Fig. 13. Example of EJB composition.

the BookStoreBean. Then, the BookStoreClient calls the "addBook" method defined in the remote interface BookStore of the BookStoreBean. Within this call, the BookStoreBean adds books by iteratively calling the method "create" defined in the home interface BookHome of the BookBean, which returns instances of the remote interface Book of the BookBean. Then, the BookStoreBean calls the "set" methods defined in the remote interface Book of the BookBean to set BookName, Author, Publisher, and Price of these books iteratively, resulting in rows of books inserted into the table books.

The other models in this category, COM, .NET, CCM, and Web Services, are outlined in Appendix A.2. In these models, the builder is usually a programming environment, like Eclipse for EJB, the deposit-only repository is either a container, like the EJB container and the CCM container, or a server, like the COM server, the Windows server for .NET, and Web servers for Web Services.

3.3 Category 3: Deployment with Repository

This category contains only JavaBeans.

In JavaBeans [17], a component is a *bean*, which is just any Java class that has *methods*, *events*, and *properties*. A bean is intended to be constructed and manipulated in a visual builder tool.

For example, consider a simple bean MessageBox, which is a Java class that has a method for displaying a message, mouse events such as "mousePressed," and the message that it displays is a property of the bean.

Properties are local to a component, so these do not figure in the bean's interface. Events can be source or target events. Source events in one bean can trigger (target) methods in another bean. More precisely, an event listener (for a target event) in a bean, when notified by an external source event (that is, a source event in another bean), triggers a corresponding method in the bean. Thus, in a bean, target events and methods are provided services in the bean's interface and external source events are the required services, that is, required services are event sinks.

Individual JavaBeans are constructed in a Java programming environment such as Java Development Kit and deposited in the ToolBox of the BDK [33], [44], which is the repository for JavaBeans. To execute or compose JavaBeans, the beans have to be dragged into a container like BeanBox. More precisely, for each bean, a JAR file containing the bean implementation class, the event state object, and the event listener interface are deposited in the ToolBox of BDK. Binaries of a bean can be dragged from the ToolBox, composed, instantiated, and executed in the BeanBox.

Although the ToolBox acts like a repository, it does not support composition of beans. Thus, the only composition

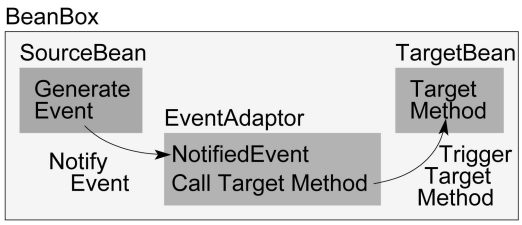

Fig. 14. Composition of JavaBeans.

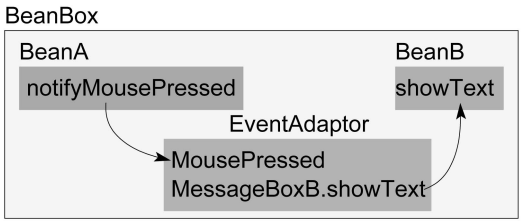

Fig. 15. Example of composition of JavaBeans.

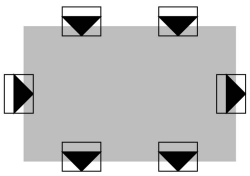

Fig. 16. A Koala component.

possible in JavaBeans is the composition of beans in the deployment phase.

Deployment phase composition is handled by the Java delegation event model, which specifies how a bean sends a message to other beans without knowing the exact methods that the other bean implements. To compose two chosen JavaBean instances in BeanBox, one bean instance must act as a source bean that can generate a source event and a method must be chosen in the other (target) bean instance, which will be triggered by the source event. Of course, the target method of the target or listener bean must match the event type of the source bean's method. The communication between the source and target beans is indirect, but this is handled by BeanBox automatically: BeanBox generates, compiles, and loads an event adapter class which routes messages from the source bean to the target bean to connect the source bean's event to the target bean's event handler method (Fig. 14).

Consider the composition of two beans, MessageBoxA and MessageBoxB. In the design phase, these two beans are developed and deposited into the ToolBox of BeanBox. No composition is possible at this stage. In the deployment phase, suppose an instance of MessageBoxA (Bean A is created with the property "Hello, I'm bean A") and an instance of MessageBoxB (Bean B is created with the property "Hello, I'm bean B") by dragging MessageBoxA and MessageBoxB from the ToolBox. To compose beans A and B, we choose bean B to be the event source and Bean A to be the event target and we choose a source event "mousePressed" in bean B. Next, bean B is linked to bean A and a target event or method in bean A "showText" is chosen which will be a listener for the source event that has been selected in bean B. BeanBox effects this composition by automatically generating and compiling an adapter class that connects beans A and B (Fig. 15). More precisely, BeanBox creates a class that calls the "showText" method in bean A whenever the event "mousePressed" occurs in bean B. It then associates the source and target beans with this "adapter" object. Therefore, when bean B is selected, and the mouse is pressed, bean A displays the message "Hello, I'm bean A."

3.4 Category 4: Design with Repository

This category includes Koala, SOFA, and KobrA. The representative example is Koala.

In Koala [20], [21], a component is a unit of design which has a specification and an implementation. Semantically, Koala components are units of computation and control (and data) connected in an architecture. Therefore, syntactically, Koala components are defined in an ADL-like language, namely, Koala, consisting of an IDL for defining component interfaces, a CDL for defining components, and a Data Definition Language (DDL) for specifying local data in components. Koala component definitions are compiled by the Koala compiler to their implementation in a programming language, for example, C.

In Koala, a component can have multiple interfaces implementing different functions and is represented in Fig. 16. Each interface specifies the signature of the function that it implements and is represented as a square containing a triangle. The tip of the triangle represents the direction of that function call.

For example, consider a Stopwatch device that is used to count down from a specific number, for example, 100. The Stopwatch device is comprised of a Countdown component and a Display component (Fig. 17). The interfaces of the Countdown and Display components are specified in Koala IDL and their component definitions are in Koala CDL. The ICount interface of the Countdown component specifies the signature of the function "count" that the Countdown component implements. The Countdown component definition defines its provides interface ICount and its implementation "c_impl." The ICount and ISignal interfaces of the Display component specify the signatures of the functions "count" and "display" that the Display component implements. The Display component definition defines its requires and provides interfaces and its implementation "d_impl."

Koala components' definition files are deposited in a repository, namely, the KoalaModel WorkSpace, which is a file system. In the design phase, Koala components are composed by method calls through connectors. There are three kinds of connectors: binding, glue code, and switch.

- *Binding* is used to connect the requires interface of a component to a provides interface of the same type of another component.
- *Glue code* serves as an adapter that connects the requires interface of a component to a provides interface of a different type of another component.
- *Switch* is special glue code that switches binding between components.

Any combination of components is, again, a component; that is, the combination of components is a composite component. A composite component is represented as in Fig. 18. Connector binding is represented as a line, glue code is represented as a node, with "m" for modules, and a switch connector is represented as a "switch" node.

In Koala, composite components can be deposited back into the repository.

In the deployment phase, Koala components are compiled into binaries of a programming language, for example, C. However, no new composition of component

```
interface ICount {
        int count(void);
   }
component Countdown {
        provides Icount cp;
        contains module c_impl present;
        connects cp = c_impl;
   }
```

```
interface ICount {
        int count(int x);
   }
interface ISignal {
        void display(int signal);
   }
component Display {
        requires ICount dr;
        provides ISignal dp;
        contains module d_impl present;
        connects dr = d_impl;
                  d_impl = dp;
   }
```

Fig. 17. Example of Koala components.

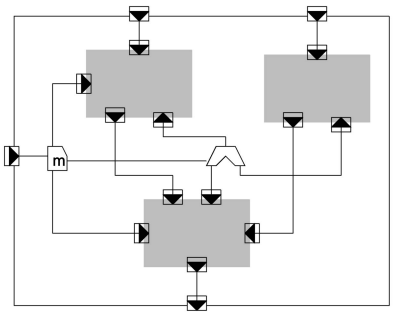

Fig. 18. Component composition in Koala.

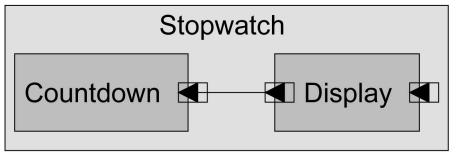

Fig. 19. A stopwatch device.

```
component Stopwatch {
        contains component Countdown c;
        contains component Display d;
        connects d.dr = c.cp;
   }
```

Fig. 20. Example of component composition in Koala.

binaries is possible. In the runtime phase, component instances are executed in the runtime environment of the chosen programming language.

Consider the Stopwatch device again. In the design phase, the Stopwatch device (Fig. 19) is being implemented by constructing a new Countdown component and composing it with a Display component from the repository. The definition files for the Display component are retrieved from the repository (definition files contain the definitions of interfaces, components and data). Then, the definition files for the Countdown component are constructed. Using their definition files, Countdown and Display are composed by method calls. This yields a definition file for Stopwatch (Fig. 20). The definition files for Countdown and Stopwatch are deposited into the KoalaModel WorkSpace. In the deployment phase, the definition files of Stopwatch Countdown and Display are compiled by the Koala compiler to C header files. Then, the programmer has to write C files and compile these with the header files to binary C code for Stopwatch.

In general, the Koala component model is used to build a product population for consumer electronics from repositories of preexisting components, that is, product lines.

The other models in this category, that is, SOFA and KobrA, are outlined in Appendix A.3. In these models, the builder is a programming environment, that is, SOFANode for SOFA and the UML Visual Builder tool for KobrA, the repository is a file system, for example, the Template Repository for SOFA, and the runtime environment is that of the implementation language.

4 A Taxonomy

In this section, we propose a taxonomy for the 13 component models described in the previous section (and in the Appendix). Clearly, such a taxonomy could be based on component semantics, syntax, or composition. We will base the taxonomy on composition, following the categories defined in the previous section, and we will argue why these categories are more meaningful than those based on component semantics or syntax.

4.1 Categories Based on Component Semantics

Based on semantics, current component models can be grouped into three categories: 1) component models in which components are classes, 2) models in which components are objects, and 3) those in which components are architectural units (Fig. 21).

Component models that belong to 1) are JavaBeans and EJB since, semantically, components in these models are special Java classes, that is, classes hosted by containers.

Component models that belong to 2) are COM, .NET, CCM, Web Services, and Fractal since, semantically, components in these models are runtime entities that behave like objects. In COM, a component is a piece of compiled code that provides some services which is hosted by a COM server. In .NET, a component is an executable DLL that is hosted by a Common Language Runtime (CLR). In CCM, a component is a Corba metatype that is an extension and specialization of a Corba object which is hosted by a CCM container on a CCM platform such as OpenCCM. In Web Services, a component is a piece of binary code that provides some services which is hosted by a Web server. In Fractal, a component is an object-like runtime entity in languages with mappings from the chosen IDL.

Component models that belong to 3) are Acme-like ADLs, UML2.0, KobrA, Koala, SOFA, and PECOS. Semantically, components in these models are units of computation and control (and data) connected in an architecture. In ADLs, a component is an architectural unit that represents a primary computational element and data store of a system. In UML2.0, a component is a modular unit of a system with well-defined interfaces which is replaceable within its environment. In KobrA, components are UML components. In Koala, SOFA, and PECOS, a component is a unit of design which has a specification and an implementation.

Component Semantics	Models
Classes	JavaBeans, EJB
Objects	COM, .NET, CCM, Web Services, Fractal
Architectural Units	Acme−like ADLs, UML2.0, KobrA, Koala, SOFA, PECOS

Fig. 21. Categories based on semantics.

Component Syntax	Models
Object−oriented Programming Languages	JavaBeans, EJB
Programming Languages with IDL mappings	COM, .NET, CCM, Web Services, Fractal
Architecture Description Languages	Acme−like ADLs, UML2.0, KobrA, Koala, SOFA, PECOS

Fig. 22. Categories based on syntax.

4.2 Categories Based on Component Syntax

Based on component syntax, current models fall into three categories: 1) models in which components are defined by object-oriented programming languages, 2) those in which an IDL is used and in which components can be defined in programming languages with mappings from the IDL, and 3) those in which components are defined by ADLs (Fig. 22).

Component models that belong to 1) are JavaBeans and EJB, where components are implemented in Java.

Component models that belong to 2) are COM, .NET, CCM, Web Services, and Fractal. These models use IDLs to define generic interfaces that can be implemented by components in specific programming languages. COM and .NET use the Microsoft IDL [15], CCM uses the OMG IDL [35], and Web Services use Web Service Description Language (WSDL), whereas Fractal can use any IDL.

Component models that belong to 3) are Acme-like ADLs, UML2.0, KobrA, Koala, SOFA, and PECOS. Obviously, in all ADLs, components are defined in ADLs. In UML2.0 and KobrA, the UML notation is used as a kind of ADL and components are defined by UML diagrams. In Koala and SOFA, components are defined in ADL-like languages. In PECOS, components are defined in CoCo, which is essentially an ADL, too.

The main difference between these categories is that components in 1) and 2) are directly executable in their respective programming languages, whereas components in 3) are only specifications which have to be implemented by somehow using suitable programming languages.

4.3 A Taxonomy Based on Composition

We have seen three groupings of categories based on component semantics, syntax, and composition. The question is whether it is possible or meaningful to combine them into a single taxonomy. Looking at the categories based on semantics (Fig. 21) and those based on syntax (Fig. 22), it is obvious that they can be merged straightforwardly into two groups:

- *Object* based: JavaBeans, EJB, COM, .NET, CCM, Web Services, and Fractal.
- *Architecture* based: Acme-like ADLs, UML2.0, KobrA, Koala, SOFA, and PECOS.

However, comparing these two groups with the categories based on composition in the component life cycle (Fig. 3), it is clear that there is no meaningful way of merging the former with the latter. Of the object-based group of the former, EJB, COM, .NET, CCM, and Web Services belong to different categories from JavaBeans and from Fractal in the latter. Of the architecture-based group of the former, KobrA, Koala, and SOFA belong to different categories from Acme-like ADLs, UML2.0, and PECOS in the latter. Conversely, the categories based on composition are not simply divided between object-based models and architecture-based models. For example, in these categories, Fractal, which is object-based, belongs to the same category as the architecture-based models Acme-like ADLs, UML2.0, and PECOS.

In view of this, we believe that the only meaningful taxonomy is one based on composition in the idealized component life cycle. Composition is the central issue in CBD after all. Moreover, in the idealized life cycle, composition takes place in both the design and deployment phases. By contrast, object-based models and architecture-based models tend to be heavily biased toward one phase or the other. In object-based models like COM, .NET, CCM, and Fractal, components are objects that are executable binaries and are therefore more of deployment phase entities than design phase entities. On the other hand, in architecture-based models such as UML2.0, components are expressly design entities by definition, with or without binary components in the deployment phase.

Thus, we propose the taxonomy of software component models shown in Fig. 23 based on component composition in the idealized component life cycle, as discussed in Section 3. The categories are the same as those in Fig. 3, but Fig. 23 shows more details about their characteristics.

In Fig. 23, the first four columns of characteristics are design phase characteristics, whereas the last one refers to the deployment phase characteristics. In the design phase, "Deposit-N" stands for "new components can be deposited in a repository," "Retrieve" stands for "components can be retrieved from the repository," "Compose" stands for "composition is possible," and "Deposit-C" stands for "composite components can be deposited in the

Category	Models	Design				Deploy
		Deposit−N	Retrieve	Compose	Deposit−C	Compose
Design without Repository	Acme−like ADLs, UML2.0, PECOS, Fractal	×	×	✓	×	×
Design with Deposit−only Repository	EJB, COM, .NET, CCM, Web Services	✓	×	✓	×	×
Deployment with Repository	JavaBeans	✓	×	×	×	✓
Design with Repository	Koala, SOFA, KobrA	✓	✓	✓	✓	×

Fig. 23. A taxonomy based on composition.

repository." In the deployment phase, "Compose" stands for "composition is possible."

In the Design without Repository category, in the design phase, there is no repository for components. Therefore, neither new components can be deposited nor can existing components be retrieved from a repository. Consequently, components are all constructed from scratch. However, composition is possible. In the deployment phase, since no repository is available, no new composition is possible.

In the Design with Deposit-only Repository category, in the design phase, new components can be deposited in a repository but cannot be retrieved from it. Composition is possible, that is, composites can be formed, but composites cannot be deposited in (and, hence, retrieved) from the repository. In the deployment phase, no new composition is possible.

In the Deployment with Repository category, in the design phase, new components can be deposited in a repository but cannot be retrieved from it. Composition is not possible in the design phase, that is, no composites can be formed, and, so, no composites can be deposited in the repository. In the deployment phase, components can be retrieved from the repository and their instances can be formed and composed.

In the Design with Repository category, in the design phase, new components can be deposited in a repository and components can be retrieved from the repository. Composition is possible and composites can be deposited in the repository. In the deployment phase, no new composition is possible: The composition of the component instances is the same as in that of the components in the design phase.

5 Discussion

The basis for the taxonomy in Fig. 23 is the idealized component life cycle, which is based on the commonly accepted desiderata of CBD, as discussed in Section 2.3. It is interesting to note that models in the Design with Repository category meet the requirements of the idealized life cycle better than the other categories. This is not surprising since Koala and KobrA use product line engineering, which has proved to be the most successful approach for software reuse in practice [53]. The main reason for its success is precisely its use of repositories of families of preexisting components, that is, product lines.

At the other end of the scale, models in the Design without Repository do not "perform" so well, mainly because they are based on Acme-like ADLs and are therefore focused on designing (systems and) components from scratch rather than reusing existing components.

Models in the other two categories are "middle of the road." They also use repositories, but they behave differently from those in the Design with Repository category in that the former store binary compiled code, whereas the latter store units of design in the repository, which are more generic and, hence, more reusable.

The taxonomy also reveals that no existing model has composition in both the design and the deployment phases. No model can retrieve composites for further composition in the deployment phase, not even those in the Design with Repository category. That is, the ideal category would be Design and Deployment with Repository, but this does currently not exist. Thus, there is room for improvement and better component models are possible. Indeed, the taxonomy that we have presented in this paper has been used by other researchers to evaluate and improve their component models, for example, [54].

Apart from adding composition and repository to both the design and deployment phases, there are two other important issues: 1) how components that are easy to reuse can be designed and 2) how composition mechanisms that will enable systematic composition can be designed. On both points, current component models do not perform well. All of the models (where components are either objects or architectural units) use message passing as the composition mechanism: Objects communicate by *direct* message passing, whereas architectural units use *indirect* message passing. Message passing relies on and induces close coupling between components. This hampers component reuse, in particular the reuse of composite components.

Current models also do not have well-defined composition theories that support systematic composition. Objects do not have a composition theory: The "composition" of two objects by direct message passing is not supported by a composition theory and the result is not a single object but just the two objects calling each other's methods. Architectural units compose via their ports and have a simple composition theory. However, this theory does not support systematic composition. In particular, it is usually defined at the level of ports and used for type checking connected ports [47] rather than at the level of whole components, in particular composite components.

We believe that to overcome these problems, components should have the key properties of *encapsulation* and *compositionality* and we have formulated such a component model [55], [56]. Our model improves reuse and defines composition hierarchically. Components encapsulate their

own data, control, and computation and their composition results in composites that preserve encapsulation. Encapsulation makes reuse easier because it removes coupling between components. Composition that preserves encapsulation leads to composite components being *self-similar* to their subcomponents. Consequently, composition is hierarchical in our model.

Objects and architectural units are both lacking in encapsulation and compositionality. Objects encapsulate data but not control or computation. They are not compositional. Architectural units are compositional and can encapsulate data, but they do not encapsulate control or computation.

Encapsulation has the potential to counter *complexity*. In our model, encapsulation occurs at every level of composition and it encapsulates every composite into just an *interface*. This interface is all we need to know about the composite in order to use it for further composition. This means that we can encapsulate much larger composites at each step and, by so doing, we are able to subsequently compose much larger composites without regard to their size or complexity.

Finally, we have not included our model in this survey because it has not yet reached a mature stage of development.

6 Conclusion

In this paper, we have presented a survey of software component models. As far as possible, we have tried to use a unified terminology in the context of an idealized scenario for CBD. We have deliberately avoided adopting terminology from any one component model. For example, we use the term "builder" in the design phase and the term "assembler" in the deployment phase to refer to composition tools in these phases rather than "builder tools" that are specific to any component models because the latter do not follow a unified terminology.

We have proposed a taxonomy by generalizing from software component models that we have examined and compared. The taxonomy clearly reveals the characteristics of categories of existing component models with regard to the desiderata of CBD. Thus, it shows what desirable features a component model should have in order to achieve these desiderata.

The ideal model does not yet exist. Such a model would allow composition in both the design and deployment phases, together with the use of a repository. It would also use components that are easier to reuse, as well as composition operators that better enable systematic composition than are afforded by current models. We believe that the ideal model should have the key characteristics of encapsulation and compositionality.

Appendix A

This appendix contains the outlines of component models that are not covered in detail in the main text.

A.1 Category 1: Design without Repository

A.1.1 UML2.0

In UML2.0 [25], [26], a component is a modular unit of a system with well-defined interfaces that is replaceable within its environment. A component defines its behavior by one or more requires and provides interfaces (ports) which implement its required and provided services. Every required service is represented by a socket and every provided service by a lollipop (Fig. 24). Components can be constructed in a visual builder tool such as Visual UML.

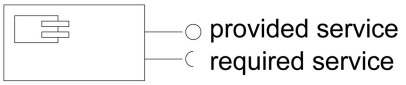

Fig. 24. A UML component.

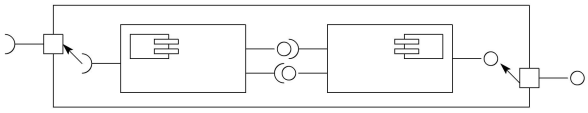

Fig. 25. Component composition in UML.

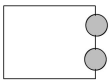

Fig. 26. A PECOS component.

In the design phase, UML2.0 components are composed of UML connectors (Fig. 25). There are two kinds of connectors:

- An *assembly connector* (lollipop in socket) is used to connect the required interface of a component to the provided interface of another component.
- A *delegation connector* (arrow) is used to forward requested and provided services from inside the environment of a composite component to outside the component.

Like ADLs, there is no deployment phase and implementations of the components and connectors in the system constructed in the design phase have to be constructed manually somehow. Some tools can perform code generation from UML2.0 specifications. Either way, the implemented system is executed in the runtime environment for the chosen programming language.

A.1.2 PECOS

In PECOS [27], [28], a component is a unit of design which has a specification and an implementation. The inputs and outputs of a component are represented as ports (Fig. 26). Components are composed by linking their ports with connectors.

Every component in PECOS has a name, a number of property bundles, a set of ports, and behavior. Ports are for data exchange, which is the only form of interaction between components with their environment (and, hence, other components). A port is specified with a unique name within a component, the type of the data passed over the port, the range of values that can be passed on this port, and the direction of the port, that is, *in*, *out*, and *inout*. A port can only be connected to another port having the same type and complementary direction.

The behavior of a component is a function or an algorithm that takes data available on the component ports or some internal data and produces data on the component ports.

In PECOS, there is no component repository. In the design phase, each system or component is specified in the CoCo language [27] in a top-down manner, that is, in terms of compositions of subcomponents. Like ADLs, CoCo specifies only the properties and ports of a component and its connectors but not its behavior. The behavior of a component has to be filled in (implemented) by the programmer. Since

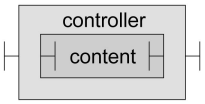

Fig. 27. A Fractal component.

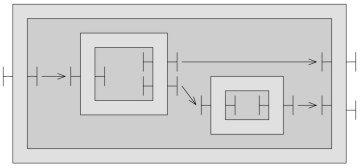

Fig. 28. Component composition in Fractal.

CoCo does not specify the behavior of components, the CoCo specification of the entire system is a syntactic specification of the composition of the subcomponents. This composition is the composition in the design phase. The (sub)components have no implementation at this stage.

Components are composed of connectors that link their ports. A connector describes a data-sharing relationship between ports. It is described by its name, its type, and a list of ports that it connects. A connector may only connect two ports if the in port and the out port have compatible data types. Ports of components can only be connected if they belong to the same parent component, that is, connectors may not cross component boundaries.

PECOS is used for specifying and developing embedded systems of field devices represented as software components. A complete system typically represents a device running in a control loop. It must have a schedule that specifies the order in which its behavior and the behavior of its subcomponents are run. Therefore, to realize an executable system from its CoCo specification, the behavior of subcomponents has to be implemented by the programmer and a schedule must then be provided for the system and any subsystem that contains subcomponents.

A.1.3 Fractal

In Fractal [30], [31], [32], a component is a runtime entity that behaves like an object. IDLs (for example, OMG IDL) are used to define generic interfaces that can be implemented by components in specific programming languages. The current Fractal API is extended and modified from Java API with JavaBeans-like introspection facilities.

A Fractal component is comprised of a content and a controller (Fig. 27). The content of a component contains its interfaces and implementation. The interfaces of a component are the only access points for other components to invoke operations defined by the component. The controller of a component defines the control behavior associated with this component. In particular, it intercepts incoming and outgoing operation invocations and operation returns targeting or originating in the component's content that it controls.

A Fractal component can implement multiple interfaces. A parametric component allows attributes to be set by its clients (via its AttributeController interface).

In the design phase, components are constructed in a programming environment with Fractal APIs and are composed by method calls through connectors (Fig. 28). The Java Virtual Machine serves as the runtime environment for Fractal components.

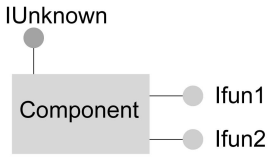

Fig. 29. A COM component.

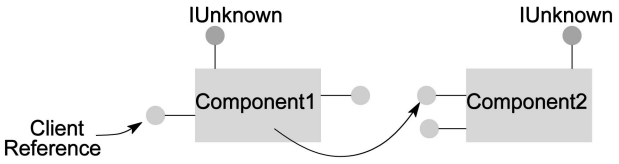

Fig. 30. Component composition in COM.

A.2 Category 2: Design with Deposit-Only Repository

A.2.1 COM

In Microsoft's COM [15], a component is a unit of compiled code (binary object) on a COM server. The language for the source code of a component can be any programming language that is supported by Microsoft IDL [15], for example, C, C++, and Ada. Services in a component are invoked via pointers to the functions that implement them. For each service provided by a component, there is an interface [57]. A COM component can implement multiple interfaces (Fig. 29). COM interfaces are specified in Microsoft IDL and every component must implement an IUnknown interface.

COM components are constructed in a programming environment such as Microsoft Visual Studio .NET, which provides a builder for COM components. The COM server is the repository and there is no assembler.

In the design phase, COM components are composed by method calls through interface pointers (Fig. 30). After the design phase comes the runtime phase, with the COM server providing the runtime environment.

A.2.2 .NET Component Model

In Microsoft's .NET [18], a component is an assembly that is a binary unit supported by CLR [58]. CLR is the runtime environment of .NET that loads, executes, and manages .NET types in the Intermediate Language (IL) into which all .NET languages are compiled. Types of all .NET languages are implemented within the Common Language Specification (CLS), which is a subset of the language features that are adopted by CLR to realize cross-language interoperability. Thus, a .NET component is implemented in any .NET language, including C#, VB, C++, that is constrained by CLS (only features included in CLS can be used) and compiled to IL code, whatever the implementation language is.

A .NET component is made up of metadata and IL code (Fig. 31). The metadata of a .NET component contains the following information:

- *Description of assembly*: Assembly identity, including name, version, and so forth, the files, types, and other resources that make up the assembly, any other assemblies that this assembly depends on, and the set of permissions that are required to run.
- *Description of types*: Name, visibility, base class, interfaces implemented, and members, including methods, fields, properties, events, and nested types.

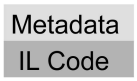

Fig. 31. A .NET component.

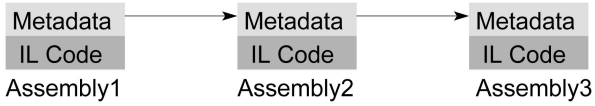

Fig. 32. Component composition in .NET.

Fig. 33. A Corba component.

- *Attributes*: Garbage collection; security attributes; version binding, and so forth.

The IL code of a .NET component is the output of a number of compilers of .NET languages (the IL code is actually the CPU-independent instruction set) that is used as the input to a just-in-time compiler in CLR (the IL code is converted to native CPU-specific code by the CLR). Therefore, the metadata is the interface of a .NET component.

.NET components are constructed in Microsoft Visual Studio .NET, which therefore provides a builder for these components. The CLR is the repository and runtime environment of .NET components.

In the design phase, .NET components are composed by method calls through references via metadata (Fig. 32). After the design phase comes the runtime phase, with the CLR providing the runtime environment.

A.2.3 CCM

In CCM [35], [59], a component is a metatype in Corba [60], which is an extension and specialization of a Corba object, hosted by a CCM container on a CCM platform such as OpenCCM [61]. Component types are specific named collections of features that can be described in OMG IDL 3. Component interfaces (Fig. 33) are made up of ports.

CCM supports four kinds of ports:

- *Facets* are distinct named interfaces that are provided by the component for client interaction. They are the provided operation interfaces of the component.
- *Receptacles* are named connection points that describe the component's ability to use a reference supplied by some external agent. They are the required operation interfaces of the component.
- *Event Sources* are named connection points that emit events of a specified type to one or more interested event consumers or to an event channel. They publish or emit events.
- *Event Sinks* are named connection points into which events of a specified type may be pushed. They consume events.

Corba components have homes that are component factories that manage a component instance's life cycle, including creation, destruction, and retrieval. Each component instance must be managed by one home instance.

Corba components are constructed in a programming environment such as Open Production Tool Chain and deposited into a CCM container hosted and managed by a CCM platform such as OpenCCM. The programming environment is the builder and the CCM container is the repository.

Fig. 34. A Web service.

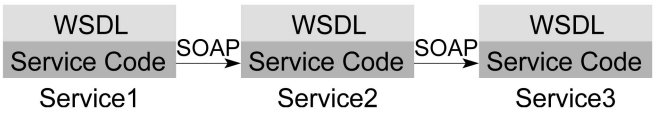

Fig. 35. Web service composition.

In the design phase, Corba components are assembled by method and event delegations in such a way that facets match receptacles and event sources match event sinks. In the runtime phase, the CCM container provides the runtime environment for Corba component instances.

A.2.4 Web Services

Web Services [62] are fundamental elements of distributed applications in Service-Oriented Computing [19]. A Web service (Fig. 34) is a piece of binary code designed to support interoperable machine-to-machine interactions for resource sharing over a network. It has an interface described in a machine processable format, specifically in the WSDL [63]. Web services interact with one another via SOAP messages [62], typically conveyed by using HTTP with an XML serialization in conjunction with other Web-related standards. The manner in which a Web Service handles SOAP messages is prescribed by its WSDL interface. WSDL defines the message formats, data types, transport protocols, and transport serialization formats that should be used between services.

Services can be implemented in any programming language and deployed on server machines that are publicly available. Interfaces of services are published in a Universal Description, Discovery, and Integration (UDDI) [62].

In the design phase, Web services are composed by delegation of method calls through SOAP messages (Fig. 35). For a service to be composed with another service, it first locates the server machine for this service by physically specifying its address in the code so that the two services could send and receive SOAP messages in the design phase.

Services are deployed in the design phase, so there is no separate deployment phase. In the runtime phase, the server of each service provides the runtime environment for the service.

Note that by service composition we mean server-side composition and not what is called Web service orchestration on the client side. The latter is used by the Web services community to create client-side applications by defining their workflows in the BPEL [39].

A.3 Category 4: Design with Repository

A.3.1 SOFA

In SOFA [23], [24], a component is a unit of design that has a specification and an implementation. It is specified by its frame and architecture. The frame defines provides and requires interfaces (Fig. 36) and properties of the component which can be implemented by more than one architecture. The architecture describes the structure of the component. SOFA components are defined in an ADL-like language, that is, SOFA CDL, which is used to define interfaces, frames, and architectures of SOFA components. SOFA components definitions are compiled by the SOFA

Fig. 36. A SOFA component.

CDL compiler to their implementations in a programming language, for example, Java.

SOFA components are constructed in the SOFANode and deposited into the Template Repository. The SOFANode is the builder for SOFA components and the Template Repository is the repository of SOFA components. There is no assembler.

In the design phase, SOFA components are composed by method calls through connectors. In the deployment phase, the SOFANode provides the runtime environment for SOFA components.

A.3.2 KobrA

In KobrA [22], a component is a UML stereotype. Every KobrA component has a specification and an implementation. The specification describes what a component does and, thus, it is the interface of the component, and the implementation describes how it does it.

KobrA components can be constructed in a visual builder tool such as Visual UML and deposited into a file system. The visual builder tool is the builder for KobrA components and the file system is the repository of KobrA components. There is no assembler.

In the design phase, KobrA components are composed by direct method calls. In the deployment phase, component implementations can be refined from their specifications. No new composition of component instances is possible.

ACKNOWLEDGMENTS

The authors wish to thank Ivica Crnkovic, David Garlan, Dirk Muthig, Oscar Nierstrasz, Bastiaan Schonhage, and Kurt Wallnau for information and helpful discussions. They are also grateful to the reviewers and the editor for illuminating feedback and constructive comments, which enabled them to make considerable improvements to the paper. Z. Wang would like to thank Universities UK for the Overseas Research Students (ORS) Award and the School of Computer Science, University of Manchester, for the Departmental Studentship. Parts of the material in this paper were presented as tutorials at the 21st IEEE/ACM International Conference on Automated Software Engineering (ASE 2006) and the 28th International Conference on Software Engineering (ICSE 2006) [4], as well as in a conference proceeding [12].

REFERENCES

[1] F. Bachmann, L. Bass, C. Buhman, S. Comella-Dorda, F. Long, J. Robert, R. Seacord, and K. Wallnau, "Volume II: Technical Concepts of Component-Based Software Engineering," second ed., Technical Report CMU/SEI-2000-TR-008, Software Eng. Inst., Carnegie Mellon Univ., 2000.

[2] *Component-Based Software Engineering: Putting the Pieces Together,* G. Heineman and W. Councill, eds. Addison-Wesley, 2001.

[3] C. Szyperski, D. Gruntz, and S. Murer, *Component Software: Beyond Object-Oriented Programming,* second ed. Addison-Wesley, 2002.

[4] K.-K. Lau, "Software Component Models," *Proc. 28th Int'l Conf. Software Eng. (ICSE '06),* pp. 1081-1082, 2006.

[5] K.-K. Lau and Z. Wang, *A Survey of Software Component Models,* second ed., School of Computer Science, Univ. of Manchester, http://www.cs.man.ac.uk/cspreprints/PrePrints/cspp38.pdf, May 2006.

[6] M. Shaw and D. Garlan, *Software Architecture: Perspectives on an Emerging Discipline.* Prentice Hall, 1996.

[7] L. Bass, P. Clements, and R. Kazman, *Software Architecture in Practice,* second ed. Addison-Wesley, 2003.

[8] L. DeMichiel, L. Yalçinalp, and S. Krishnan, *Enterprise JavaBeans Specification Version 2.0,* 2001.

[9] R. Monson-Haefel, *Enterprise JavaBeans,* fourth ed. O'Reilly & Assoc., 2004.

[10] P. Clements, "A Survey of Architecture Description Languages," *Proc. Eighth Int'l Workshop Software Specification and Design (IWSSD '96),* pp. 16-25, 1996.

[11] 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.

[12] K.-K. Lau and Z. Wang, "A Taxonomy of Software Component Models," *Proc. 31st Euromicro Conf. Software Eng. and Advanced Applications (SEAA '05),* pp. 88-95, 2005.

[13] M. Broy, A. Deimel, J. Henn, K. Koskimies, F. Plasil, G. Pomberger, W. Pree, M. Stal, and C. Szyperski, "What Characterizes a Software Component?" *Software—Concepts and Tools,* vol. 19, no. 1, pp. 49-56, 1998.

[14] B. Meyer, "The Grand Challenge of Trusted Components," *Proc. 25th Int'l Conf. Software Eng. (ICSE '03),* pp. 660-667, 2003.

[15] D. Box, *Essential COM.* Addison-Wesley, 1998.

[16] *CORBA Component Model, V3.0,* OMG, http://www.omg.org/technology/documents/formal/components.htm, 2002.

[17] *JavaBeans Specification.* Sun Microsystems, http://java.sun.com/products/javabeans/docs/spec.html, 1997.

[18] A. Wigley, M. Sutton, R. MacLeod, R. Burbidge, and S. Wheelwright, *Microsoft .NET Compact Framework (Core Reference).* Microsoft Press, Jan. 2003.

[19] G. Alonso, F. Casati, H. Kuno, and V. Machiraju, *Web Services: Concepts, Architectures and Applications.* Springer-Verlag, 2004.

[20] 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, Mar. 2000.

[21] R. van Ommering, "The Koala Component Model," *Building Reliable Component-Based Software Systems,* I. Crnkovic and M. Larsson, eds., pp. 223-236, Artech House, 2002.

[22] 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, 2001.

[23] F. Plášil, D. Balek, and R. Janecek, "SOFA/DCUP: Architecture for Component Trading and Dynamic Updating," *Proc. Fourth Int'l Conf. Configurable Distributed Systems (ICCDS '98),* pp. 43-52, 1998.

[24] F. Plášil, M. Besta, and S. Visnovsky, "Bounding Component Behavior via Protocols," *Proc. Technology of Object-Oriented Languages and Systems (TOOLS 31),* pp. 387-398, 1999.

[25] *UML 2.0 Superstructure Specification,* OMG, http://www.omg.org/cgi-bin/doc?ptc/2003-08-02, 2007.

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

[27] T. Genssler, A. Christoph, B. Schulz, M. Winter, C. Stich, C. Zeidler, P. Müller, A. Stelter, O. Nierstrasz, S. Ducasse, G. Arévalo, R. Wuyts, P. Liang, B. Schönhage, and R. van den Born, *PECOS in a Nutshell,* http://www.pecos-project.org/, Sept. 2002.

[28] O. Nierstrasz, G. Arévalo, S. Ducasse, R. Wuyts, A. Black, P. Müller, C. Zeidler, T. Genssler, and R. van den Born, "A Component Model for Field Devices," *Proc. First Int'l IFIP/ACM Working Conf. Component Deployment (CD '02),* pp. 200-209, 2002.

[29] "The Fractal Project Web Page," http://fractal.objectweb.org/, 2007.

[30] E. Bruneton, T. Coupaye, and J. Stefani, "Recursive and Dynamic Software Composition with Sharing," *Proc. Seventh Int'l Workshop Component-Oriented Programming (WCOP '02),* 2002.

[31] E. Bruneton, T. Coupaye, and J. Stefani, "The Fractal Component Model," ObjectWeb Consortium, Technical Report Specification V2, 2003.

[32] E. Bruneton, T. Coupaye, and M. Leclercq, "An Open Component Model and Its Support in Java," *Proc. Seventh Int'l Symp. Component-Based Software Eng. (CBSE '04),* pp. 7-22, 2004.

[33] *JavaBeans Architecture: BDK Download.* Sun Microsystems, http://java.sun.com/products/javabeans/software/bdk_download.html, 2003.
[34] *Java 2 Platform, Enterprise Edition* Sun Microsystems, http://java.sun.com/j2ee/, 2007.
[35] *Common Object Request Broker Architecture: Core Specification, Version 3.0.3,* http://www.omg.org/technology/documents/corba_spec_catalog.htm, Mar. 2004.
[36] D. Garlan, R. Monroe, and D. Wile, "Acme: Architectural Description of Component-Based Systems," *Foundations of Component-Based Systems,* G. Leavens and M. Sitaraman, eds., pp. 47-68, Cambridge Univ. Press, 2000.
[37] C. Peltz, "Web Services Orchestration and Choreography," *Computer,* vol. 36, no. 10, pp. 46-52, Oct. 2003.
[38] M. Lumpe, F. Achermann, and O. Nierstrasz, "A Formal Language for Composition," *Foundations of Component Based Systems,* G. Leavens and M. Sitaraman, eds., pp. 69-90, Cambridge Univ. Press, 2000.
[39] T. Andrews, F. Curbera, H. Dholakia, Y. Goland, J. Klein, F. Leymann, K. Liu, D. Roller, D. Smith, S. Thatte, I. Trickovic, and S. Weeragwarana, *Business Process Execution Language for Web Services (BPEL4WS) Version 1.1.* IBM, http://www-106.ibm.com/developerworks/library/ws-bpel/, 2004.
[40] R. Allen and D. Garlan, "A Formal Basis for Architectural Connection," *ACM Trans. Software Eng. and Methodology,* vol. 6, no. 3, pp. 213-249, 1997.
[41] I. Crnkovic, H. Schmidt, J. Stafford, and K. Wallnau Proc. Sixth Workshop Component-Based Software Eng.: Automated Reasoning and Prediction (CBSE '04). *ACM SIGSOFT Software Eng. Notes,* vol. 29, no. 3, pp. 1-7, May 2004.
[42] B. Christiansson, L. Jakobsson, and I. Crnkovic, "CBD Process," *Building Reliable Component-Based Software Systems,* I. Crnkovic and M. Larsson, eds., pp. 89-113, Artech House, 2002.
[43] B. Warboys, B. Snowdon, R. Greenwood, W. Seet, I. Robertson, R. Morrison, D. Balasubramaniam, G. Kirby, and K. Mickan, "An Active Architecture Approach to COTS Integration," *IEEE Software,* special issue on incorporating COTS into the development process, vol. 22, no. 4, pp. 20-27, July/Aug. 2005.
[44] *The Bean Builder,* Sun Microsystems, https://bean-builder.dev.java.net/, 2007.
[45] *AcmeStudio 2.1 User Manual,* Carnegie Mellon Univ., http://www-2.cs.cmu.edu/~acme/Manual/AcmeStudio-2.1.htm, 1998.
[46] J. Aldrich, C. Chambers, and D. Notkin, "ArchJava: Connecting Software Architecture to Implementation," *Proc. 24th Int'l Conf. Software Eng. (ICSE '02),* pp. 187-197, 2002.
[47] J. Aldrich, C. Chambers, and D. Notkin, "Architectural Reasoning in ArchJava," *Proc. 16th European Conf. Object-Oriented Programming (ECOOP '02),* pp. 334-367, 2002.
[48] J. Aldrich, D. Garlan, B. Schmerl, and T. Tseng, "Modeling and Implementing Software Architecture with Acme and ArchJava," *Proc. Companion 19th Ann. ACM SIGPLAN Conf. Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA '04),* pp. 156-157, 2004.
[49] L. DeMichiel and M. Keith, *Enterprise JavaBeans, Version 3.0.* Sun Microsystems, 2006.
[50] R. Monson-Haefel, *Enterprise JavaBeans 3.0,* fifth ed. O'Reilly & Associates, 2006.
[51] "Eclipse Web Page," http://www.eclipse.org/, 2007.
[52] Y. Choi, O. Kwon, and G. Shin, "An Approach to Composition of EJB Components Using C2 Style," *Proc. 28th Euromicro Conf. (Euromicro '02),* pp. 40-46, 2002.
[53] P. Clements and L. Northrop, *Software Product Lines: Practices and Patterns.* Addison Wesley, 2001.
[54] P. Hnětynka and F. Plášil, "Dynamic Reconfiguration and Access to Services in Hierarchical Component Models," *Proc. Ninth Int'l Symp. Component-Based Software Eng. (CBSE '06),* I. Gorton et al., eds., pp. 352-359, 2006.
[55] K.-K. Lau, P. Velasco Elizondo, and Z. Wang, "Exogenous Connectors for Software Components," *Proc. Eighth Int'l Symp. Component-Based Software Eng. (CBSE '05),* I. Gorton et al., eds., pp. 90-106, 2005.
[56] K.-K. Lau, M. Ornaghi, and Z. Wang, "A Software Component Model and its Preliminary Formalisation," *Proc. Fourth Int'l Symp. Formal Methods for Components and Objects (FMCO '06),* F. de Boer et al., eds., pp. 1-21, 2006.
[57] A. Major, *COM IDL and Interface Design.* John Wiley & Sons, Feb. 1999.
[58] M. Barnett and W. Schulte, "Runtime Verification of .Net Contracts," *Systems and Software,* vol. 65, no. 2003, pp. 199-208, 2003.
[59] BEA Systems et al., "CORBA Components," *Object Management Group,* Technical Report orbos/99-02-05, 1999.
[60] R. Natan, *CORBA: A Guide to Common Object Request Broker Architecture.* McGraw-Hill, 1995.
[61] *OpenCCM User's Guide,* ObjectWeb—Open Source Middleware, http://openccm.objectweb.org/doc/0.8.1/user_guide.html, 2007.
[62] E. Newcomer, *Understanding Web Services: XML, WSDL, SOAP, and UDDI.* Addison-Wesley, 2002.
[63] Ariba, Microsoft, and IBM, *Web Services Description Language (WSDL) Version 1.1.,* http://www.w3.org/TR/2001/NOTE-wsdl-20010315, 2001.

Kung-Kiu Lau received the BSc and PhD degrees from the University of Leeds, United Kingdom. He is currently a senior lecturer in the School of Computer Science at the University of Manchester, United Kingdom. He is the series editor of a book series on component-based software development published by World Scientific. He is an area editor (for logic and software engineering) of the *Journal of Applied Logic.* He has served on the program committees of numerous international conferences, including ASE, CBSE, and SC. He has also delivered invited talks and tutorials at many international meetings, including an invited keynote talk at FMCO 2005 and tutorials on software component models at ASE 2005 and ICSE 2006. His main research interest is component-based software development.

Zheng Wang received the BSc degree from the University of Manchester, United Kingdom, where he is currently working toward the PhD degree in the School of Computer Science. He participated in and completed the project European Network of Excellence in Computational Logic (CologNet), where he maintained the Web site of Component-Based Software Development. He coorganized the 2004 and 2005 Workshop on Predictable Software Component Assembly. His research interests are component-based software engineering and software verification.