April 2005

A Survey of Software Component Models

Kung-Kiu Lau and Zheng Wang

School of Computer Science
The University of Manchester
Preprint Series
CSPP-30

A Survey of Software Component Models

Kung-Kiu Lau and Zheng Wang

April 2005

Abstract

In Software Engineering, Component-based Development (CBD) is an important emerging topic, promising long sought after benefits like increased reuse and reduced time-to-market (and hence software production cost). However, there are at present many obstacles to overcome before CBD can succeed. For one thing, CBD success is predicated on a standardised market place for software components, which does not yet exist. In fact currently CBD even lacks a universally accepted terminology. Existing component models adopt different component definitions and composition operators. Therefore much research remains to be done. We believe that the starting point for this endeavour should be a thorough study of current component models, identifying their key characteristics and comparing their strengths and weaknesses. A desirable side-effect would be clarifying and unifying the CBD terminology. In this report, we present a clear and concise exposition of all the current major software component models, including a taxonomy. The purpose is to distill and present knowledge of current software component models, as well as to present an analysis of their properties with respect to commonly accepted criteria for CBD. The taxonomy also provides a starting point for a unified terminology.

Keywords: software component model, repository, life-cycle, component syntax, component semantics, component composition, predictable assembly

Contents

1 Introduction **3**

2 An Abstract Software Component Model **4**
2.1 The Syntax of Software Components 4
2.2 The Semantics of Software Components 4
2.3 The Composition of Software Components 6
2.3.1 The Life Cycle of Components 6
2.3.2 Composition in the Design Phase 7
2.3.3 Composition in the Deployment Phase 8

3 Current Software Component Models **8**
3.1 JavaBeans 8
3.1.1 The Semantics and Syntax of Java Beans 8
3.1.2 Composition of Java Beans 9
3.1.3 Summary 11
3.2 Enterprise JavaBeans 11
3.2.1 The Semantics and Syntax of Enterprise Java Beans 11
3.2.2 Composition of Enterprise Java Beans 15
3.2.3 Summary 16
3.3 Component Object Model 17
3.3.1 The Semantics and Syntax of COM Components 17
3.3.2 Composition of COM Components 18
3.3.3 Summary 20
3.4 CORBA Component Model 20
3.4.1 The Semantics and Syntax of CORBA Components 20
3.4.2 Composition of CORBA Components 22
3.4.3 Summary 23
3.5 Koala 23
3.5.1 The Semantics and Syntax of Koala Components 23
3.5.2 Composition of Koala Components 26
3.5.3 Summary 27
3.6 SOFA 27
3.6.1 The Semantics and Syntax of SOFA Components 27
3.6.2 Composition of SOFA Components 29
3.6.3 Summary 30
3.7 KobrA 30
3.7.1 The Semantics and Syntax of KobrA Components 30
3.7.2 Composition of KobrA Components 31
3.7.3 Summary 32
3.8 Architecture Description Languages 32
3.8.1 The Semantics and Syntax of ADL Components 32
3.8.2 Composition of ADL Components 32
3.8.3 Summary 34
3.9 UML 2.0 35
3.9.1 The Semantics and Syntax of UML 2.0 Components 35

3.9.2 Composition of UML 2.0 Components . . . 37
3.9.3 Summary . . . 37
3.10 PECOS . . . 38
3.10.1 The Semantics and Syntax of PECOS Components . . . 38
3.10.2 Composition of PECOS Components . . . 39
3.10.3 Summary . . . 41
3.11 Pin . . . 41
3.11.1 The Semantics and Syntax of Pin Components . . . 41
3.11.2 Composition of Pin Components . . . 42
3.11.3 Summary . . . 44
3.12 Fractal . . . 44
3.12.1 The Semantics and Syntax of Fractal Components . . . 44
3.12.2 Composition of Fractal Components . . . 45
3.12.3 Summary . . . 46

4 Towards A Taxonomy **46**
4.1 Categories based on Component Syntax . . . 46
4.2 Categories based on Component Semantics . . . 48
4.3 Categories based on Component Composition . . . 48
4.4 A Taxonomy of Software Component Models . . . 50
4.5 Discussion . . . 51

5 Conclusion **52**

6 Acknowledgements **53**

References **53**

1 Introduction

In Software Engineering, Component-based Development (CBD) is widely accepted as a promising approach that could potentially yield the long sought after benefits of software reuse, reduced time-to-market (and hence software production cost), interoperability, as well as tractable quality certification [49]. However, the success of CBD in realising these benefits is predicated on a standardised market place for software components, and at present there are many obstacles to overcome before this can be achieved.

As yet, there is not even a universally accepted terminology for CBD [9]. 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.

The cornerstone of any CBD methodology is its underlying *component model*, which defines what components are, how they can be constructed, how they can be composed or assembled, how they can be deployed and how to reason about all these operations on components. This is exemplified by Heineman and Councill's definition of a component [29] as:

> "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."

However, at present the more widely adopted definitions of components tend to be given without component models, whilst at the same time, the more widely used component models do not adopt these definitions properly. For example, the widely accepted definition due to Szyperski [49]:

> "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 due to Meyer [35]:

> "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 [25, 36], COM [8] and CCM [6, 40] adopt slightly different component definitions from 'standard' ones like Szyperski's, and from each other.

For future research it would therefore be crucial to clarify and unify the CBD terminology and its definition. We believe that the starting point for this endeavour should be a thorough study of current component models. In this report, we take this first step and present a clear and concise exposition of these component models, including a taxonomy. The purpose is to distill and present knowledge of current software component models, as well as to present an analysis of their properties with respect to commonly accepted criteria for CBD. The taxonomy also provides a starting point for a unified terminology.

The report is organised as follows. In Section 2 we present an abstract component model which will serve as a reference model. In Section 3 we give a clear and concise exposition of all the major current software component models. This is done in a uniform manner, following the abstract model. Based on this exposition, in Section 4, we propose and discuss a taxonomy of current software component models. In Section 5 we make our conclusion.

2 An Abstract Software Component Model

First we present an abstract component model as a reference framework for current component models, which defines (and explains) terms of reference that we will use throughout this report. 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* should define:

- the *syntax* of components, i.e. how they are constructed and represented;
- the *semantics* of components, i.e. what components are meant to be;
- the *composition* of components, i.e. how they are composed or assembled.

We will consider the following software component models: JavaBeans [48], EJB, COM, CCM, Koala [51, 50], KobrA [5], SOFA [42, 43], Architecture Description Languages [20, 34], UML 2.0 [39, 16], PECOS [28, 37], Pin [30] and Fractal [11, 12, 10].

2.1 The Syntax of Software Components

Obviously the language used for constructing components determines the syntactic rules for the components. In a component model, therefore this language should be specified.

In current component models, the language for components tends to be a programming language. For example in both JavaBeans and EJB, a component, respectively a bean and an enterprise bean, is defined as a Java class.

2.2 The Semantics of Software Components

A generally accepted view of a software component is that it is a software unit consisting of (i) a name; (ii) an interface; and (iii) code (Figure 1 (a)). The code implements the services

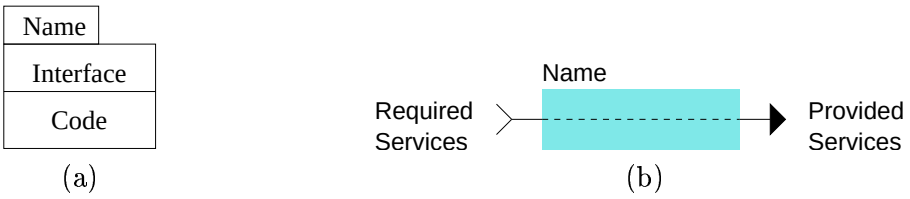


Figure 1: A software component.

provided, or operations performed, by the component, and is not visible or accessible from outside. The interface is the only point of access to the component, so it should provide all the

information that is necessary to use the component. In particular, it should specify the services required by the component in order to produce the services it provides (Figure 1 (b)). Roughly speaking, the provided services of a component are its output, while the required services are its input. Required services are typically input values for parameters of the provides services. The interface of a component thus specifies the *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. This matching could be expressed by listing corresponding services as ordered pairs $\langle r_1, p_1 \rangle, \ldots, \langle r_n, p_n \rangle$, where each r_i and p_i is a set of services. In the example in Figure 1 (b) this matching is made explicit by a dotted line joining corresponding services.[1]

Note that in Figure 1 (b), our notation, whilst similar to UML, is different from UML. Whereas in UML, every required service (represented by a socket) and every provided service (represented by a lollipop) is regarded as a separate interface, we regard the interface as a single entity that is the combined specification of all the required and provided services. Another difference with UML is that we can explicitly specify dependencies (using a dotted line, as shown in Figure 1 (b)) between provided and required services, whereas in UML such intra-component dependencies cannot be represented.

Components are (sub)parts of a system, and as such are incomplete functional units. A system, on the other hand, is a functionally complete unit that can be deployed in an environment. In a system, the required services of a component may be provided by other components, but they may also be inputs that must come from sources outside the system, i.e. they may be *system inputs*. Similarly, the provided services of a component may go to other components, or they may go outside the system, i.e. they may be *system outputs*. It is useful to distinguish these services, so we will use the notation in Figure 2.

(a) component required service

(b) component provided service

(c) system required service (system input)

(d) system provided service (system output)

Figure 2: Required and provided services.

For example, the component in Figure 3 (a) has both component required and component provided services, whereas the component in Figure 3 (b) behaves like a system by itself.

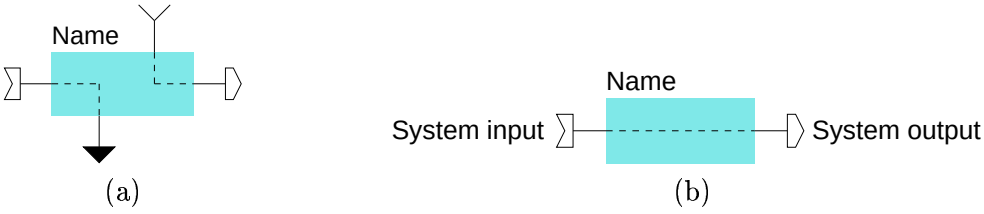


Figure 3: Examples of components.

Note that a component can have multiple required and multiple provided services, with various combinations of dependencies. For example, the component in Figure 3 (a) requires a

[1] For simplicity we have just one (set of) required and one (set of) provided service(s) in this example.

system input for a service it provides for another component, and it requires a component service for a system output it provides. For simplicity, in this report, we will use simple components with just one required and one provided service.

In current component models, components tend to be 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, e.g. 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, Java beans and enterprise Java beans are different semantically. Semantically a Java bean is a Java class that is hosted by a container such as BeanBox [47]. Java beans interact with one another via adaptor classes generated by the container. Adpator classes link beans via events. An enterprise Java bean, on the other hand, is a Java class that is hosted and managed by an EJB container provided by a J2EE server [46], 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.

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. In order to define composition, we need a composition language, e.g. [32]. 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, CCM have no composition languages. Koala uses connectors (and glue code) for composition. KobrA and UML 2.0 use UML notation. Architecture description languages (ADLs) are of course (formal) composition languages [4], and PECOS and Pin have ADL-like composition languages, viz. CoCo [28] and CCL [55] respectively.

In order to reason about composition, we need a composition theory (see discussion in [24]). 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.

2.3.1 The Life Cycle of Components

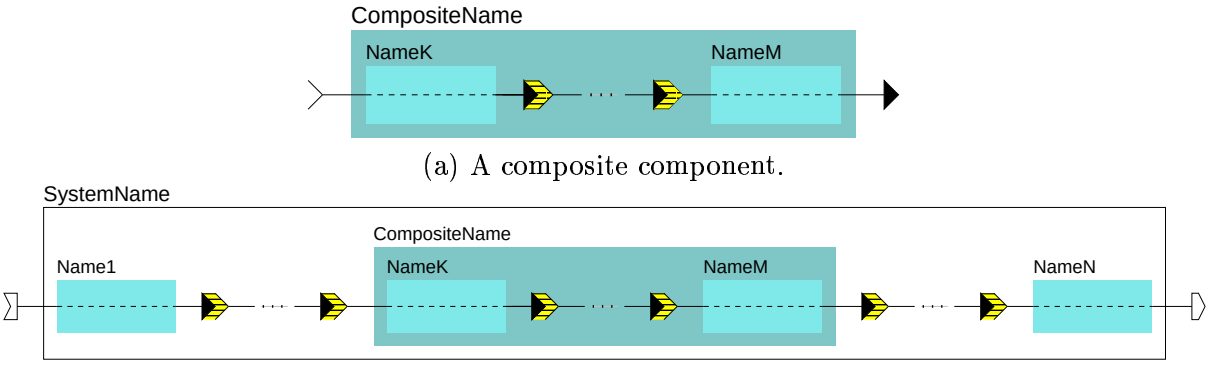
Composition can take place during different stages of the *life cycle* of components [18]. We identify two main stages in this cycle, the *design* phase and the *deployment* phase.

- *Design Phase.*
  In this phase, components are designed and constructed, and then may be deposited in a repository if there is one. Components constructed in this phase are *stateless*, since they are just templates (like classes) that cannot execute. The only data that they can contain at this stage are constants. Nevertheless, (unlike classes) they can be composed into composite components. If there is a component repository, then the constructed components, including composite ones, have to be catalogued and stored in the repository in such a way that they can be retrieved later, as and when needed.

- *Deployment Phase.*
  In this stage, component instances are created by instantiating (composite) components with initial data, so that they have *states* and are ready for execution. If there is a component repository, then components have to be retrieved from the repository before instantiation.

2.3.2 Composition in the Design Phase

In the design phase, components can be composed into *composite components* or *system templates*. A *composite component* is one with component required and component provided services only, and is built from components that also only have component required and component pro-



(a) A composite component.

(b) A system template.

Figure 4: Design phase composition.

vided services (Figure 4 (a), where composition is represented by the *connector* ⩥). A composite component is therefore a component of the type shown in Figure 1 (b). Composite components form sub-parts of a system, and as such are useful for designing a system.[2]

A *system template* is a composition of (possibly composite) components which forms a system, i.e. with system inputs and system outputs only (Figure 4 (b)). A system template is therefore a component of the type shown in Figure 3 (a).

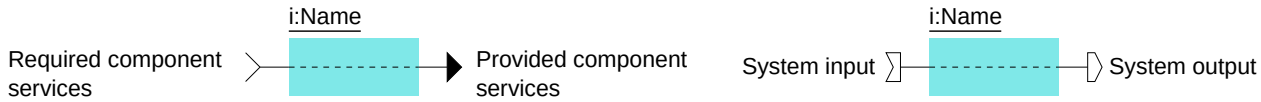
Composite components and system templates are both *stateless*, and can be stored in the repository, and retrieved later. Composite components form sub-parts of a system, and are therefore useful for designing a system. A system template represents the design of a whole system. Therefore, composition in the design phase is concerned with the design of systems and their sub-parts. If there is a component repository, then composite components can also be stored in the repository, and retrieved later, like any components. In the design phase, a builder tool can be used to (i) construct new components, and then deposit them in the repository; and (ii) retrieve components from the repository, compose them and deposit them back in the repository.

For example, in JavaBeans, the container provides a repository for beans, e.g. ToolBox in the Beans Development Kit (BDK) [47, 45], which are stored as JAR files. However, no composition is possible in the design phase, and therefore no composite beans can be formed.

[2] Again, for simplicity the composite component here has only one (set of) required and provided service(s).

2.3.3 Composition in the Deployment Phase

In the deployment phase, composite components and system templates are retrieved from the repository, and instantiated with data. These instances are thus initialised with states and are ready for execution (like objects). Following UML, we use underlined names for instances (Figure 5).



(a) A component instance. (b) A system.

Figure 5: Instances of components.

Composition of these instances is done in the same way as component composition in the Design Phase (Figure 6).

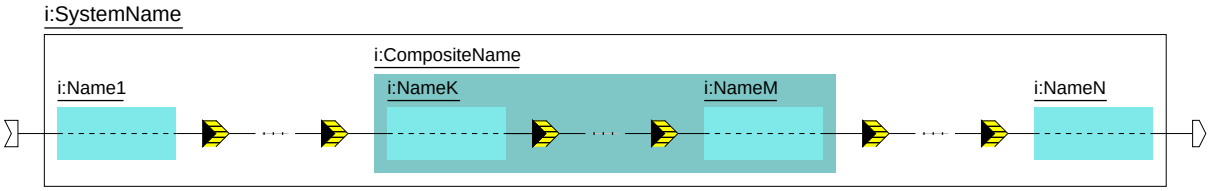


Figure 6: Deployment phase composition.

Component instances can be composed by an *assembler* tool and the result of composition is a whole executable system. Instances of components or systems cannot be stored in the repository. Rather, they are the end result of system design (design phase) and implementation (deployment phase).

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

3 Current Software Component Models

In this section, we describe existing software component models. Our description of each model follows that of the abstract model in the previous section. For each model, we also give an example to illustrate the semantics and syntax of the components, and component composition in both the design and deployment phases. Thus we describe all the models in a uniform way.

3.1 JavaBeans

3.1.1 The Semantics and Syntax of Java Beans

In Java Beans, a component is a *bean*, which is just any Java class that has:

- methods
- events

- properties.

A bean is intended to be constructed and manipulated in a visual builder tool.

Example 3.1.1 For example, consider a simple bean MessageBox that displays a message when it is notified of the event 'mousePressed' by another bean. The MessageBox bean is a Java class that has a method for displaying a message, mouse events such as 'mousePressed', and the message it displays is a property of the bean which can be set by the programmer.

Note that MessageBox can be a complete system by itself, and this is true of any bean in general.

Properties are local to a component, so 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 (i.e. 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, i.e. required services are event sinks (Figure 7 (a)).

Figure 7: Java beans.

Event sinks and event sources (and target methods) may also be system required services and system provided services respectively. In particular, a bean may be a complete system by itself (Figure 7 (b)).

So even though in general, in object-oriented programming, classes and objects just provide services (methods) without specifying their required services (methods), in JavaBeans the beans can specify their required services as events.

3.1.2 Composition of Java Beans

Individual Java beans are constructed as Java classes in a Java programming environment such as Java Development Kit, and deposited in the ToolBox of the BDK, which is the repository for Java beans. To execute or compose Java beans, 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 is deposited in the ToolBox of BDK. Instances of a bean can be created by dragging the bean from the ToolBox, and these can be executed or composed in the BeanBox.

Although the ToolBox acts like a repository, it does not support composition of beans. So the only composition possible in JavaBeans is the composition of bean instances 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 Java bean 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 is handled by BeanBox automatically: BeanBox generates, compiles, and loads an event adaptor class, that 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 (Figure 8).

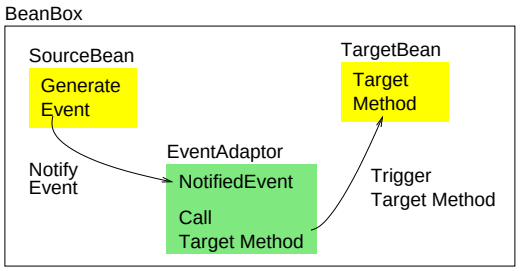
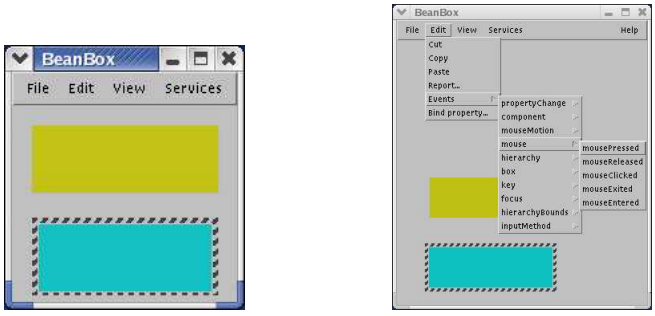


Figure 8: Deployment phase composition of Java beans.

Example 3.1.2 Consider the composition of two beans MessageBoxA and MessageBoxB, which are examples of the MessageBox bean in Example A.1. In the design phase, these two beans are developed and deposited into the ToolBox of BeanBox. No composition is possible at this stage.

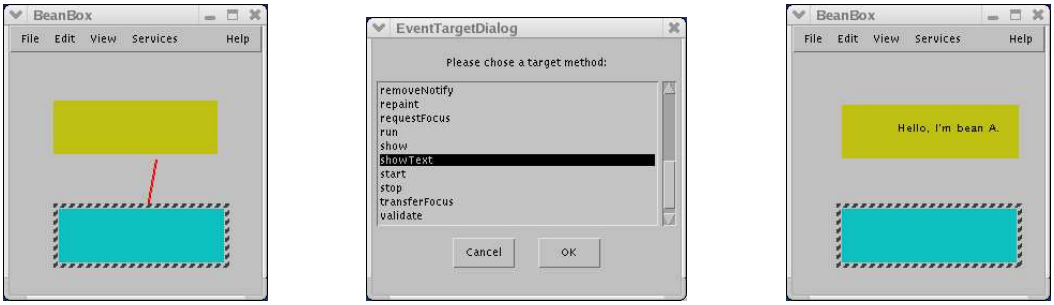
In the deployment phase, suppose an instance of MessageBoxA, which we will call Bean A, is created with the property 'Hello, I'm Bean A', and an instance of MessageBoxB, which we will call Bean B, is created with the property 'Hello, I'm Bean B', by dragging MessageBoxA and MessageBoxB from the ToolBox (Figure 9 (a)).



(a) (b)

Figure 9: Example of bean composition in JavaBeans.

To compose Beans A and B in BeanBox, we need to choose a bean to be the event source, and the other the event target. Suppose we choose Bean B to be the event source (and therefore Bean A is the event target). To do so, we select Bean B (indicated by highlighted border). Then a source event in Bean B is chosen. Suppose the chosen event is 'mousePressed' (Figure 9 (b)).



(c) (d) (e)

Figure 10: Example of bean composition in JavaBeans (continued).

Next Bean B is linked to Bean A (Figure 10 (c)). To do this a target event or method in Bean A is chosen that will be a listener for the source event that has been selected in Bean B. BeanBox shows all the candidate target events or methods in Bean A in the EventTargetDialog box (Figure 10 (d)). Suppose the target method 'showText' is chosen.

The composition of Beans A and B is now complete, and BeanBox effects this composition by automatically generating and compiling an adaptor class that connects Beans A and B. More precisely, BeanBox creates a class that calls the 'showText' method in the target bean (Bean A) whenever the event 'mousePressed' occurs in the source bean (Bean B). It then associates the source and target beans with this "adaptor" object. The adaptor class thus serves as a connector that calls the method 'showText' defined in Bean A whenever the event 'mousePressed' is fired in Bean B (see also Figure 10). Therefore, when Bean B is selected and the mouse is pressed, Bean A displays the message 'Hello, I'm beanA' (Figure 10 (e)).

Conversely, if Bean B is chosen as the event target, and Bean A the event source, then Bean B will display the message 'Hello, I'm bean B', when Bean A is selected and the mouse is pressed.

3.1.3 Summary

In JavaBeans, a component is a Java class with methods, events and properties, intended to be constructed and used in a visual builder tool. In the design phase, Java beans can be constructed in a Java programming environment such as Java Development Kit and JAR files containing class files for Java beans are deposited in the ToolBox of the BDK, which is the repository for Java beans. There is no bean composition in the design phase. In the deployment phase, the assembler is the BeanBox of the BDK, which can be used to compose bean instances by the Java delegation event model.

3.2 Enterprise JavaBeans

3.2.1 The Semantics and Syntax of Enterprise Java Beans

In Enterprise JavaBeans (EJB), 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. JNDI lookup services

provide support for a client to locate the target enterprise beans. The J2EE remote connectivity model handles the remote communication between the client and enterprise beans. Figure 11 shows an overview of EJB.

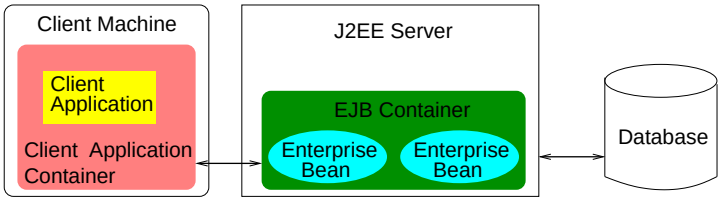


Figure 11: Overview of EJB.

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 the methods needed for (remote) client applications to access the bean (over a network). The home interface represents the life-cycle methods of the bean such as *create*, *destroy* and *locate* a bean instance, while the remote interface represents the tasks performed by the bean.

There are three different kinds of enterprise beans [25]:

- **Entity beans**
  Entity beans model business data; they are Java objects that cache database information. An entity bean represents a persistent business object whose data is stored in a database, and adds behaviour specific to that data.

- **Session beans**
  Session beans model business processes; they are Java objects that act as agents performing tasks. A session bean represents a business process or an agent that performs a service. It is different from an entity bean in that it does not represent persistent data.

- **Message-Driven beans**
  Message-driven beans model message-related business processes; they are Java objects that act as message listeners. A message-driven bean represents a business process that can only be triggered by receiving messages from other beans. It is different from a session bean in that it cannot be accessed through an interface.

Entity beans represent business objects in a persistent storage mechanism. Persistence means that the entity bean's state exists beyond the lifetime of the application. Typically, each entity bean has an underlying table in a relational database, and each instance of the bean corresponds to a row in that table. Entity beans can be shared by multiple clients and each entity bean has a unique object identifier that is its primary key in the database. There are two types of entity beans, corresponding to Container-Managed Persistence (CMP) and Bean-Managed Persistence (BMP). With CMP, the container manages the persistence of the entity bean. With BMP, the entity bean contains database access code and is responsible for reading and writing its own state to the database.

Session beans are used to manage interactions with entity beans, access resources, and generally perform tasks on behalf of the client. There are two types of session bean: Stateless

and Stateful. The state of an object consists of the values of its instance variables. Stateless session beans do not even maintain a conversational state for the client, while stateful beans retain conversational state for the duration of the client-bean session. The state is held in secondary storage and is not persistent.

Message-Driven beans act as a Java Message Service (JMS) listeners that allow J2EE applications to process messages asynchronously. They are similar to event listeners except that they receive JMS messages instead of events. A message-driven bean has only a bean implementation class without home and remote interfaces.

Example 3.2.1 Consider a book store which 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 book store can use a set of enterprise beans to implement a system that allows multiple clients to access and update the database. For example, an entity bean, like the one in Figure 12, can represent the table of books in a database.

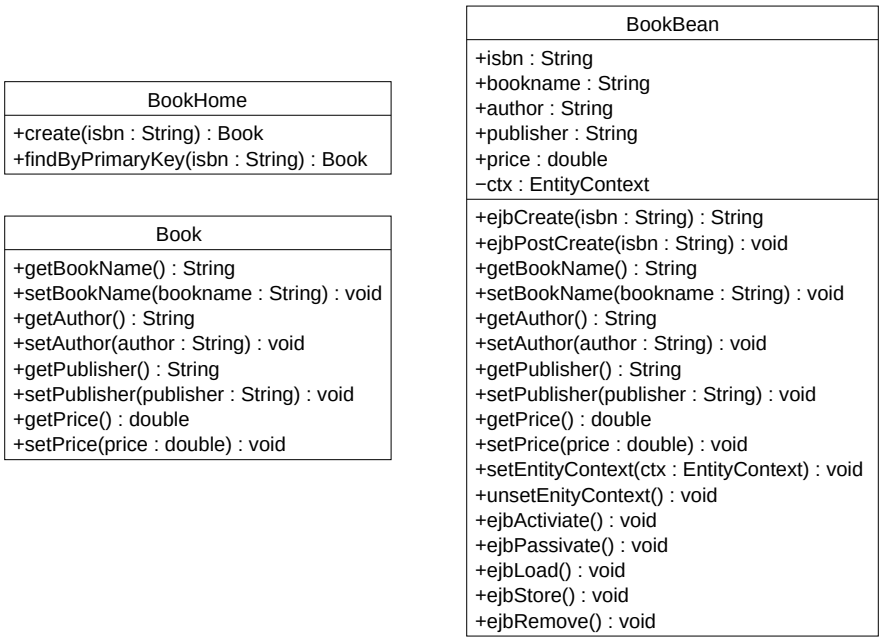


Figure 12: Entity bean for book store example.

The entity bean in Figure 12 consists of one class and two interfaces: (a) BookBean is the Java class that defines the methods of the entity bean; (b) BookHome is the home interface of the entity bean; and (c) 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 to 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.[3]

Note that this entity bean can behave like a system by itself. For instance, when adding a book into the table of books, the client can call the method 'create' defined in the home interface

[3]Persistence is managed by the bean if the 'ejbLoad' and 'ejbStore' methods in BookBean are implemented by the bean provider. Otherwise, persistence is managed by the EJB container by default.

BookHome, which returns an instance of the remote interface Book, and then the client can call the 'set' methods defined in Book to set BookName, Author, Publisher and Price of this book, resulting in a new row inserted into the table of books in the database.

For this example, apart from the above entity bean, there may also be a session bean, like the one in Figure 13, consisting of the class BookStoreBean, the home interface BookStoreHome

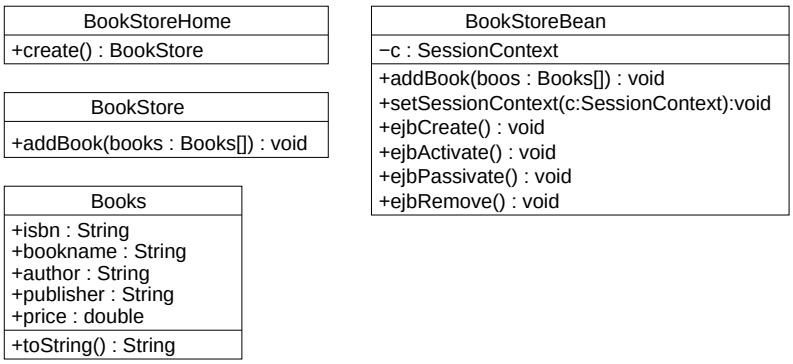


Figure 13: Session bean for book store example.

and the remote interface BookStore (and its helper class Books). BookStoreBean is used to add details of a set of books into the table of books in the database. It therefore defines tasks performed on the table of books. Since the latter is represented by the entity bean BookBean, BookStoreBean is a stateless session bean that calls methods in the entity bean to complete its tasks. The only method defined in the home interface BookStoreHome of BookStoreBean is a life-cycle method: 'create', which creates an instance of BookStoreBean. 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, method in a session bean correspond to methods in its home and remote interfaces.

Note that although BookStoreBean cannot behave like a system by itself, in general a session bean can do so if it does not have to perform tasks on the database, i.e. if it does not have to call the methods of an entity bean. For example, if a session bean computes some numbers and outputs them directly to the client, then it behaves like a system.

In general, in EJB a client can access an enterprise bean only through the methods defined in the bean's home and remote interfaces. These interfaces thus specify the *provided services* of an enterprise bean. Some session beans may require methods defined in other beans to accomplish a task, so the *required services* of an enterprise bean are external method calls (Figure 14 (a)).

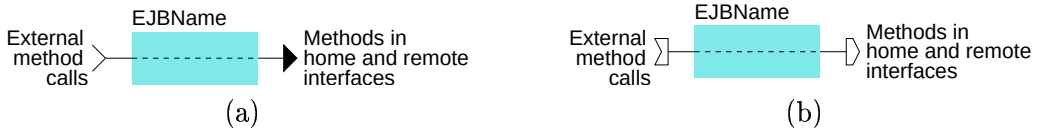


Figure 14: Enterprise Java beans.

Instead of coming from other beans, external method calls may come from a client, i.e. they may be system required services. Similarly, methods in home and remote interfaces can also produce outputs for the client or the database, i.e. they can be system provided services. So,

a single enterprise bean may behave like a system by itself (Figure 14 (b)).

More precisely, an entity bean can take method calls from a client or another enterprise bean. It can call the methods of another enterprise bean, but it must send its output to the database. A session bean can take method calls from a client or from another enterprise bean, and it can call the methods of another enterprise bean and send outputs to the client.

3.2.2 Composition of Enterprise Java Beans

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 [26] and their JAR files containing the enterprise bean implementation class, the home and remote interfaces, and the deployment descriptor [4] are deposited in an EJB container on a J2EE server that is the repository of enterprise beans. If a bean, say bean B, contains method calls to another bean, say bean A, then A must be deposited in the container before B can be deposited (and A and B must be linked by a JNDI name). This is of course to avoid dangling method calls, and it implies that in an empty EJB container, the first bean that can be deposited must not contain method calls to other beans, i.e. it must be a bean that only produces outputs (to the client or the database). Thus in the design phase, composition of enterprise beans must happen successively as one bean is added to the container at a time, in such a way that the end result is a bean assembly that is a complete system template, with input from and output to the client or database. The EJB container in Figure 15 shows an example of design phase composition of two session beans, SessionBeanA and SessionBeanB, and one entity bean, EntityBean, into a system template: SessionBeanA takes client calls and EntityBean writes to the database.

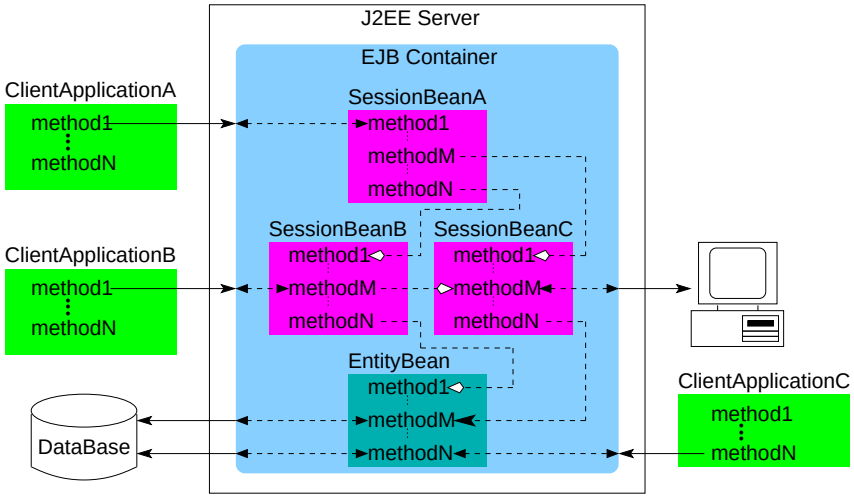


Figure 15: Design phase composition of enterprise Java beans.

In the design phase, although bean composition can produce composite beans 'on the fly' in the EJB container, it is not possible to store such a bean with its own identity and reuse it as a single bean (with this identity) for further composition. However, individual beans in the

[4] Apart from the class files for the bean and its home and remote interfaces, a JAR file also contains the XML deployment descriptor, an XML Document Type Definition which is used to specify security, persistence and transactions for the bean.

container in the repository 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 be automatically followed. This is illustrated by Figure 15. When ClientApplicaitonA calls method1 in SessionBeanA, this causes SessionBeanA to call method1 in SessionBeanB, which in turn causes SessionBeanB to call method1 in EntityBean. Finally, EnitityBean writes to the database.

When ClientApplicationB calls methodM in SessionBeanB, the latter writes directly to an output device.

When ClientApplicationC calls methodN in EntityBean, the latter writes to the database.

Thus although it is a repository for enterprise beans that supports 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 the deployment phase, bean instances are created and executed in the EJB container; there is no composition for bean instances.

Example 3.2.2 Consider the book store example in Example 3.2.1 again. In the design phase, a system is deposited into the EJB container, consisting of an assembly of the session bean BookStoreBean and the entity bean BookBean. 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 (Figure 16).

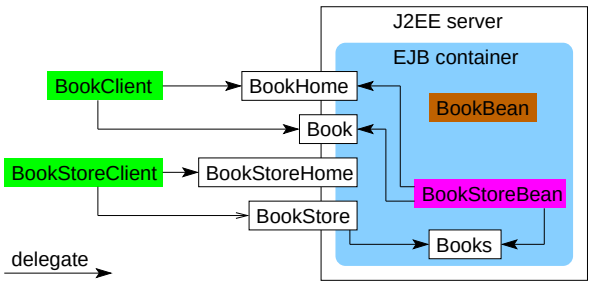


Figure 16: Example of EJB composition.

In the deployment 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 the BookStoreBean. Then the BookStoreClient calls the 'addBook' method defined in the remote interface BookStore of the BookStoreBean. Within this call, the BookStoreBean adds three 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 three rows of books inserted into the table books.

3.2.3 Summary

In EJB, a component is an enterprise Java bean, which is a Java class in an EJB container on a J2EE server, together with two Java interfaces that the container uses to manage and execute the Java class and its instances. Enterprise beans can be constructed in a Java programming

environment such as Eclipse. In the design phase, enterprise beans are composed by method and event delegation, and JAR files for enterprise beans that are assembled into complete system templates are deposited in the EJB container, which is the repository of enterprise beans. In the deployment phase, bean instances are created and used by client applications. No new composition is possible (see e.g. [17]), and so there is no assembler. The EJB container provides the run-time environment for the bean instances.

3.3 Component Object Model

3.3.1 The Semantics and Syntax of COM Components

In Microsoft's COM (Component Object Model), a component is a unit of compiled code (binary object) on a COM server. The code of a COM component provides some services, possibly by invoking the services provided by other components.[5] The language for the source code of a component can be any programming language that supports function calls via pointers, e.g. C, C++ and Ada.[6] Services in a component are invoked via pointers to the functions that implement the services.

A COM component has an interface [33] for every service it provides (it is said to implement the interface), and the set of interfaces is the only point of access to the component's services. In COM, a component is a box with lollipops that represent the interfaces it implements, e.g.

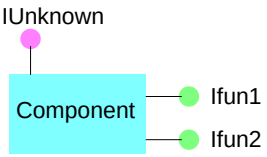


Figure 17: A COM component.

the component in Figure 17 implements 3 interfaces.[7] COM interfaces are specified in Microsoft IDL [8] (also known as COM IDL). Each interface specifies the signatures of the functions it implements. A COM component is identified by globally unique identifier (GUID) that is either a CLSID (a unique identifier for a component/class) or an IID (an identifier for an interface) [22].

A COM component can implement multiple interfaces. Every component must implement an IUnknown interface (e.g. the component in Figure 17). IUnknown is a special interface that implements some essential functionality such as reference counting methods that can determine if a component is being called by other components, and query methods that allow users to dynamically discover what interfaces a component supports.

Example 3.3.1 For example, consider a spell checker system that is used to check the spelling of a word. When a word is typed into the spell checker, it firstly looks for the corresponding correctly spelt word in its dictionary, then compares the input word with this correct one, and finally tells the user whether the input word is spelt correctly or not. This spell checker system comprises a checker component and a dictionary component (Figure 18). The interfaces

[5] Somewhat confusingly, a COM component is called a component object, although it is not necessarily an object in the object-oriented sense.

[6] Implementation language of COM components should be supported by Microsoft IDL.

[7] In COM, all interface names start with 'I' by convention.

of the checker and dictionary components are specified in Microsoft IDL. From the interfaces

```
import "unknwn.idl";
[object, uuid(CAB357AE−1204−4783−AC3F−A7E4CA19EF6C)]
interface ISpellCheck : IUnknown {
        HRESULT CheckSpelling([in, string] char *word,
                              [out, retval] BOOL *isCorrect);
 }
[uuid(0EE7AE7−A357−4a04−B6D6−CE4DFD5CCAAF)]
library SpellcheckerLib {
        [uuid(49FA65CD−8CF6−4876−8443−37A75A267A7D)]
        coclass CSpellCheck {
                interface ISpellCheck;
          }
 };
```

Checker component interface − ISpellCheck

```
import "unknwn.idl";
[object, uuid(D66AB784−75C8−4f52−8EB2−C5BE9796ABEF)]
interface IUseCustomDictionary : IUnknown {
        HRESULT UseCustomDictionary([out, retval] vector <string>* dict);
 }
[uuid(1C381680−CF29−46b1−8060−1237C36EA6C7)]
library CustomdictionaryLib {
        [uuid(C51815AF−CB06−4028−956C−C5F3E5781780)]
        coclass CCustomDictionary {
                interface IUseCustomDictionary;
          }
 };
```

Dictionary component interface − IUseCustomDictionary

Figure 18: Examples of COM components.

of the checker and dictionary components, it is clear that they both implement the IUnknown interface. The ISpellCheck interface of the checker component specifies the signature of the CheckSpelling function that the checker component implements. The IUseCustomDictionary interface of the dictionary component specifies the signature of the UseCustomDictionary function that the dictionary component implements.

In general, in COM, functions defined in a component's interfaces are services provided by the component, whilst external function calls from other components via interface pointers are required services (Figure 19 (a)). Of course, external function calls maybe made by client applications and the results returned to the clients, in which case a component behaves like

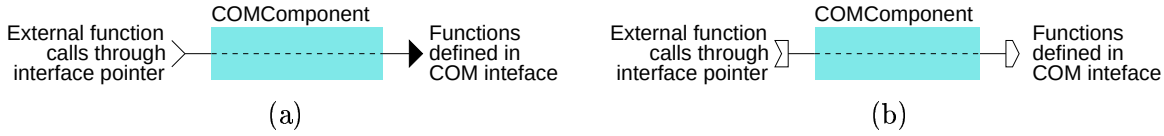


Figure 19: COM components.

a complete system, with client function calls as system required services and the functions it implements as system provided services (Figure 19 (b)).

3.3.2 Composition of COM Components

COM components cannot directly interact with each other. Components always access other components through interface pointers. An interface pointer is a pointer through which the client of the component can access its functions in the interface. So, an interface of a component is the only thing that can be publicly visible and the only point of communication with other components. Thus, COM components are composed via method calls through their interface pointers (Figure 20).

In the design phase, COM components are constructed in a programming environment such as Microsoft Visual Studio .NET [53] and they are deposited in the COM server, which is the repository of COM components. Components are composed by method calls through interface pointers in the design phase. In the deployment phase, no new composition is possible, so there is no assembler, and the COM server provides the run-time environment.

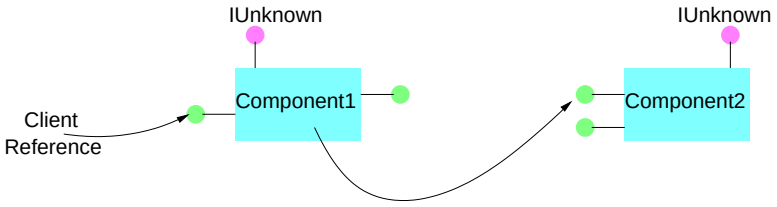


Figure 20: Composition of COM components.

As shown in Figure 20, COM components can be composed to a composite component or a system template in the design phase. However, because the COM server does not support storage of composite components and retrieval of components, so only atomic components and system templates are exist in the repository. Component instances are not possible to composed again in the deployment phase.

Example 3.3.2 Consider the spell checker system in Example 3.3.1 again. In the design phase, a system is deposited into the COM server, consisting of an assembly of the checker and dictionary component. The composition is defined by the checker component calling the method 'UseCustomDictionary' defined in the interface IUseCustomDictionary of the dictionary component (Figure 21).

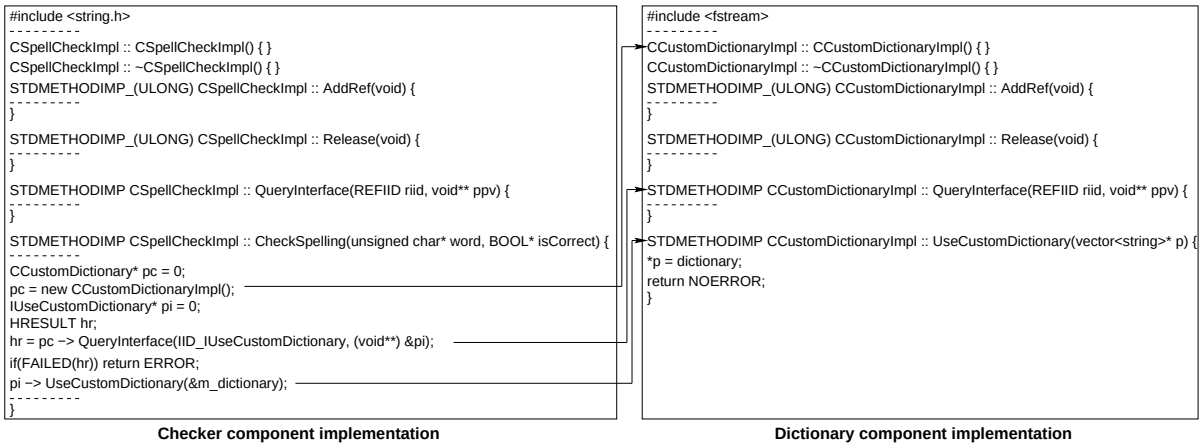


Figure 21: Example of composition of COM component.

In the deployment phase, this system is looked up and instantiated. In this instance of the system, the spell checker client defines an interface pointer of the interface ISpellCheck of the checker component, through which the client calls the function 'CheckSpelling' implemented by the checker component. Then the checker component instance calls the function 'UseCustomDictionary' implemented by the dictionary component through an interface pointer of the interface IUseCustomDictionary of the dictionary component. Within this call, the checker component compares the spelling of the input word with the words provided the dictionary iteratively, and tells the user that the correctness of the spelling of the word input.

As an aside, we give a summary of COM-based technologies. Distributed Component Object Model (DCOM) [44] is that part of COM that is concerned with enabling COM-based software

components to be used over a network. DCOM is quite similar to COM but with security functionalities. Microsoft Transaction Server (MTS) [15] is distributed runtime environment for COM or the COM application server. Windows Distributed InterNet Applications Architecture (Windows DNA) [7] is a distributed applications development model that uses COM as its integration technology, and uses Internet Information Server (IIS) [23], MTS, and Internet Explorer to provide integration with the World Wide Web. COM+ [41] is the second generation of COM. It includes a new set of features that integrate DCOM and MTS. ActiveX [14] is a COM-based technology that has utility on the World Wide Web. OLE [14] is COM compound document technology. .NET [56] is the platform for COM components and COM based products to develop and deploy. So, the definition of component and the mechanisms of assembly for all those COM related Microsoft technologies are the same as COM. Therefore, in this survey, we present COM as a standard Microsoft software component model.

3.3.3 Summary

In COM, a component is a piece of compiled code that provides some services, that is hosted by a COM server. COM components are constructed in a programming environment such as Microsoft Visual Studio .NET. In the design phase, COM components are composed by method calls through interface pointers and they are deposited in the COM server, which is the repository of COM components. In the deployment phase, no new composition is possible, so there is no assembler, and the COM server provides the run-time environment.

3.4 CORBA Component Model

3.4.1 The Semantics and Syntax of CORBA Components

In CORBA Component Model (CCM), a component is a CORBA meta-type that is an extension and specialisation of CORBA Object, that is hosted by a CCM container on a CCM platform such as OpenCCM [38]. Component types are specific, named collections of features that can be described in OMG IDL 3. A component is denoted by a component reference, which is represented by an object reference. A component definition is a specialisation and extension of an interface definition, but only supports single inheritance. A component type encapsulates its internal state and implementation. A component type can be instantiated to create concrete entities (instances) with state and identity. Components can be constructed in any object-oriented programming languages that have mappings from OMG IDL 3.

Component interfaces are made up of ports through which clients and other components may interact with them (Figure 22). The CORBA Component Model supports four kinds of

Figure 22: A CORBA component.

ports:

- **Facets** are distinct named interfaces 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 to manage a component instance life cycle including creation, destruction, and retrieval of component instances. Every component instance must be managed by a home instance.

Example 3.4.1 Consider a simple bank system which has just one ATM that serves one bank consortium. The bank consortium has two bank branches Bank1 and Bank2. This bank system is being implemented by ATM, BankConsortium, Bank1 and Bank2 components (Figure 23). The attributes, facets, receptacles, event sinks and event sources of components ATM,

```
interface Bank {
   string getBankID(string cardno);
   void deposit(string cardno);
   void withdraw(string cardno);
   void checkBalance(string cardno);
  }
```

```
enum BankState {
   IsCustomer, NotCustomer
  };
eventtype AccountInfo {
   public string cardno;
   public BankState customerinfo;
  };
```

```
component ATM {
    attribute string atmid;
    uses Bank getBankID;
    consumes AccountInfo customer;
  };
home ATMhome manages ATM {
    factory new(in string atmid);
  };
```

```
component BankConsortium {
    attribute string bankconsortiumid;
    provides Bank getBankID;
    uses Bank deposit;
    uses Bank withdraw;
    provides Bank checkBalance;
    publishes AccountInfo customer;
  };
home BankConhome manages BankConsortium {
    factory new(in string bankconsortiumid);
  };
```

```
component Bank {
    attribute string bankid;
    provides Bank deposit;
    provides Bank withdraw;
    provides Bank checkBalance;
  };
home Bankhome manages Bank {
    factory new(in string bankid);
  };
```

Figure 23: Examples of CORBA components.

BankConsortium, Bank1 and Bank2 are specified in OMG IDL 3 respectively. The instantiation of those components are managed by their component homes.

In general, the provided services of a CORBA component are specified in its Facets and Event Sources and the required services of a CORBA component are specified in its Receptacles and Event Sinks. A CORBA component may have its local state consisting of the values of its variables. Thus in CORBA Component Model, facets and event sources are provided services

in a CORBA component interface, whilst receptacles and event sinks are required services (Figure 24 (a)). A component in CCM may have system inputs and system outputs, so a

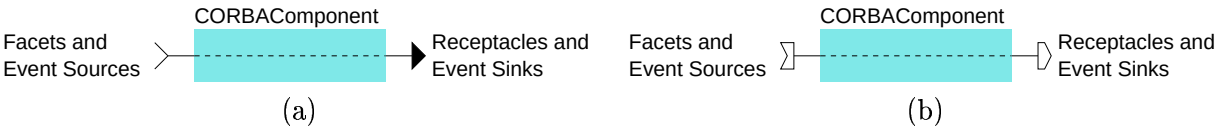


Figure 24: Required and provided services of CORBA components.

CORBA component can behave like a system by itself (Figure 24 (b)).

3.4.2 Composition of CORBA Components

In CCM, components are constructed in a programming environment such as Open Production Tool Chain hosted and managed by a CCM platform such as OpenCCM. The repository of CORBA components is a CCM container hosted and managed by an application server, and CORBA components are assembled by method and event delegations in a way that facets match receptacles and Event sources match Event sinks in the design phase (Figure 25). In

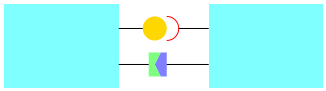


Figure 25: Composition of CORBA components.

the deployment phase, the CCM container provides the run-time environment. The CCM container encapsulate CORBA components from client applications. Any component access is done through a CCM container provided by an application server that provides system services like a run-time environment, multiprocessing, load-balancing, device access, provide naming and transaction services and make containers visible. CORBA components use a CCM container to implement component access to system services.

There are two basic types of containers:

- **transient containers** that may contain transient, non-persistent components whose states are not saved at all; and
- **persistent containers** that contain persistent components whose states are saved between invocations.

Containers are defined in terms of how they use the underlying CORBA infrastructure and thus are capable of handling services like transactions, security, events, persistence, life-cycle services, and so on.

The container is a running piece of code that is installed on the server machine. CORBA components are server-side objects; the system administrator installs components into the container, which takes charge of them when they run. CCM containers are transactional, secure, and persistent (Figure 26). Client applications find the CCM container that contains the CORBA component through CORBA COS Naming Service. They create, or obtain a reference to the component's container through the component's home interface. They then make use of the CCM Container to invoke the component methods.

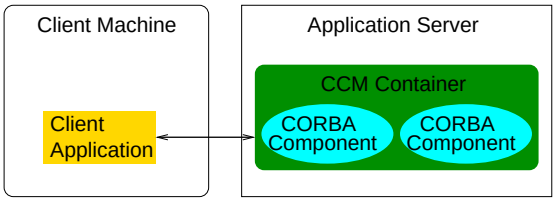


Figure 26: CCM container.

Example 3.4.2 Consider the simple bank system in Example 3.4.1 again. In the design phase, components ATM, BankConsortium, Bank1 and Bank2 are constructed and deposited into a CCM container. Components are composed by method and event delegations in a way that facets match receptacles and Event sources match Event sinks in the design phase (Figure 27). The composition of CORBA components is specified in Component Assembly Descriptor[8] that

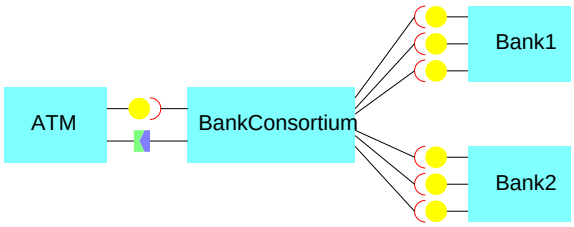


Figure 27: Example of composition of CORBA components.

is an XML file (Figure 28). In the deployment phase, instances of ATM, BankConsortium, Bank1 and Bank2 components are composed within the assembler in the same manner as their composition in the design phase. The CCM container provides the run-time environment for CORBA component instances.

3.4.3 Summary

In CORBA Component Model (CCM), a component is a CORBA meta-type that is an extension and specialisation of CORBA object, that is hosted by a CCM container on a CCM platform such as OpenCCM. In the design phase, components are constructed in a programming environment such as Open Production Tool Chain hosted and managed by a CCM platform such as OpenCCM. The repository of CORBA components is a CCM container hosted and managed by an application server, and CORBA components are assembled by method and event delegations in a way that facets match receptacles and Event sources match Event sinks in the design phase. In the deployment phase, the CCM container provides the run-time environment for CORBA component instances.

3.5 Koala

3.5.1 The Semantics and Syntax of Koala Components

In Koala (C[K]omponent Organizer and Linking Assistant) component model, a component is a unit of design which has a specification and an implementation. Koala components are defined

[8]Because OMG IDL 3 cannot specify the actual components assembly. A component assembly descriptor is a file, which is in XML, describing a set of component files, component instantiation and their assembly.

```
<?xml version = "1.0"?>
<!DOCTYPE component assembly BANKSYSTEM "componentassembly.dtd">
<component assembly id = "banksys">
  <description> bank assembly descriptor</description>
  <componentfiles>
    <componnetfile id = "ATM component">
    <filearchive name = "ATM.csd">
    </componentfile>
    <componnetfile id = "BankConsortium component">
    <filearchive name = "BankConsortium.csd">
    </componentfile>
    <componnetfile id = "Bank component">
    <filearchive name = "Bank.csd">
    </componentfile>
  </componentfiles>
  <partitioning>
    <homereplacement id = "ATMHome">
      <componentfileref idref = "ATM Component"/>
      <componentinstantiation id = "atm">
      <registerwithnaming name = "ATMHome"/>
    </homereplacement>
    <homereplacement id = "BankConsortiumHome">
      <componentfileref idref = "BankConsortium Component"/>
      <componentinstantiation id = "bankconsortium">
      <registerwithnaming name = "BankConsortiumHome"/>
    </homereplacement>
    <homereplacement id = "BankHome">
      <componentfileref idref = "Bank Component"/>
      <componentinstantiation id = "bank1">
      <componentinstantiation id = "bank2">
      <registerwithnaming name = "BankHome"/>
    </homereplacement>
  </partitioning>
  <connections>
    <connectinterface>
      <usesport>
        <usesidentifier>getBankID</usesidentifier>
        <componentinstantiationref idref = "atm"/>
        <usesidentifier>deposit</usesidentifier>
        <usesidentifier>withdraw</usesidentifier>
        <usesidentifier>checkBalance</usesidentifier>
        <componentinstantiationref idref = "bankcon"/>
      </usesport>
      <providesport>
        <providesidentifier>getBankID</providesidentifier>
        <componentinstantiationref idref = "bankcon"/>
        <providesidentifier>deposit</providesidentifier>
        <providesidentifier>withdraw</providesidentifier>
        <providesidentifier>checkBalance</providesidentifier>
        <componentinstantiationref idref = "bank"/>
      </providesport>
    </connectinterface>
    <connectevent>
      <publishesport>
        <publishesidentifier>customer</publishesidentifier>
        <componentinstantiationref idref = "bankcon"/>
      </publishesport>
      <consumesport>
        <consumesidentifier>customer</consumesidentifier>
        <componentinstantiationref idref = "atm"/>
      </consumesport>
    </connectevent>
  </connections>
</component assembly>
```

Figure 28: The component assembly descriptor of the bank system.

in its Interface Definition Language (IDL), Component Definition Language (CDL) and Data Definition Language (DDL) that are similar to Architecture Description Languages (ADLs). Koala IDL is used to specify Koala component interfaces, its CDL is used to define Koala components and its DDL specifies local data of Koala components. Semantically, components in Koala are units of computation and control (and data) connected together in an architecture. Syntactically, components in Koala are defined in an ADL-like language. Koala components definitions are compiled by Koala compiler to their implementations in a programming language, e.g. C.

In Koala, a component is represented as Figure 29. Interfaces are represented as squares with triangle, the tip of triangle represents the direction of function call. A Koala component's interface specifies the signature of a set of functions implemented by the component.

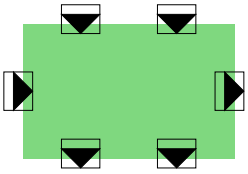


Figure 29: A Koala component.

Example 3.5.1 For example, consider a Stopwatch device that is used to count down from a specific number, e.g. 100. The Stopwatch device comprises a Countdown component and a Display component. The interfaces of the Countdown and Display components are specified in Koala IDL and their component definitions are in Koala CDL (Figure 30). The ICount

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

Countdown component

```
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;
    }
```

Display component

Figure 30: Examples of Koala components.

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 's_impl'. The ICount and ISignal interfaces of the Display component specify the signatures of the function 'count' and 'display' that the Display component implements. The Display component definition defines its requires and provides interfaces and its implementation 'd_impl'.

In general, in Koala, the provided services of a Koala component are its provides interfaces and the required services of a Koala component are its requires interfaces as shown in Figure 31 (a). Alternatively, provides interfaces can also be system provided services, requires interfaces

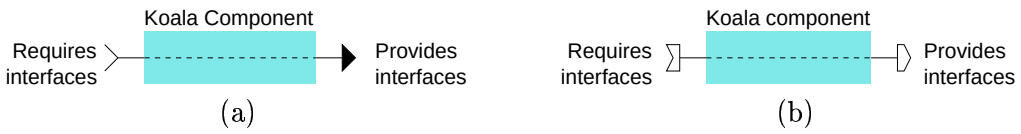


Figure 31: Required and provided interfaces of Koala components.

can also be system required services. A Koala component may behave like a system by itself (Figure 31 (b)).

3.5.2 Composition of Koala Components

In Koala, components are definition files that represent design units in the Koala language. The repository for Koala components is 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 required interface of a component to the provided interface of the same type of another component.
- **Glue code** serves as an adaptor that connects the required interface of a component to a provided interface of a different type of another component.
- **Switch** is a special glue code that switches binding between components.

Any combination of components is again a component, so, in fact, the combination of components is a composite component. A Koala configuration is a list of components (part list) and a list of connectors (net list) between components. In fact, a configuration is a system template.

In Koala, connector binding is represented as a line, glue code is represented as a note with "m" (for modules) and a switch connector is represented as a "switch" note. So, a composite component is represented as shown in Figure 32.

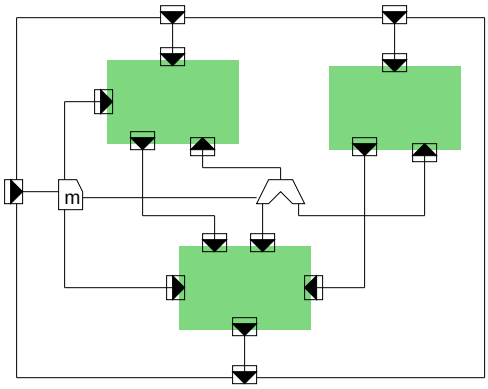


Figure 32: Composition of Koala components.

Koala components are constructed as component definition files and deposited into KoalaModel Workspace in the design phase. Koala components can also be retrieved from KoalaModel Workspace and composed with other components to a composite component or a system template that is then deposited back to the repository.

In the deployment phase, Koala components are compiled into a programming language, e.g. C, and executed in the run-time environment of that language. No new composition of component instances is possible.

Example 3.5.2 Consider the Stopwatch device in Example 3.5.1 again. In the design phase, the Stopwatch device (Figure 33) is 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

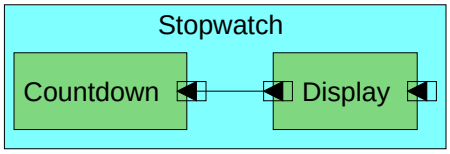


Figure 33: Stopwatch device.

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 (Figure 34). The definition files for Countdown and Stopwatch are deposited into the KoalaModel WorkSpace.

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

Stopwatch configuration

Figure 34: Example of composition of Koala components.

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, Koala component model is used to build a product population for consumer electronics from repositories of pre-existing components, i.e. product lines.

3.5.3 Summary

In Koala, a component is a unit of design which has a specification and an implementation. Koala components are constructed as component definition files and deposited into KoalaModel Workspace, which is the repository of Koala components. In the design phase, Koala components are composed by method calls through connectors. Koala components can also be retrieved from KoalaModel Workspace and composed with other components to a composite component or a system template that is then deposited back to the repository in the design phase. In the deployment phase, no new composition is possible, so there is no assembler, and Koala components are compiled into a programming language, e.g. C, and executed in the run-time environment of that language.

3.6 SOFA

3.6.1 The Semantics and Syntax of SOFA Components

In SOFA (SOFtware Appliances) component model, a component is a unit of design which has a specification and an implementation. A SOFA component is specified by its frame and architecture. The frame defines provides and requires interfaces, and properties of the component. The frame can be implemented by more than one architecture. The architecture describes the

structure of the component. Semantically, components in SOFA are units of computation and control (and data) connected together in an architecture.

SOFA components are defined in its Component Definition Language (CDL) that is similar to Architecture Description Languages (ADLs). SOFA CDL is used to define interfaces, frames and architectures of SOFA components. Syntactically, components in SOFA are defined in an ADL-like language. SOFA components definitions are compiled by SOFA CDL compiler to their implementations in a programming language, e.g. Java.

In SOFA, a component is represented as Figure 35. Interfaces are represented as rectangle with black rectangle for provides interfaces and white rectangle for requires interfaces.

Figure 35: A SOFA component.

Example 3.6.1 For example, consider a Stopwatch device that is used to count down from a specific number, e.g. 100. The Stopwatch device comprises a Countdown component and a Display component. The Countdown and Display components are specified in SOFA CDL (Figure 36). The CountInterface of the Countdown component specifies the signature of the function

```
interface CountInterface {
        int count(void);
    };
frame Countdown {
        provides:
            CountInterface Count;
    };
architecture CUNI Countdown
                    version "1.0" primitive;
```

Countdown component

```
interface CountInterface {
        int count(int x);
    };
interface SignalInterface {
        void display(int signal);
    };
frame Display {
        requires:
            CountInterface Count;
        provides:
            SignalInterface Signal;
    };
architecture CUNI Display
                    version "1.0" primitive;
```

Display component

Figure 36: Examples of SOFA components.

'count' that the Countdown component implements. The Countdown component frame defines its provides interface Count. The CountInterface and SignalInterface of the Display component specify the signatures of the function 'count' and 'display' that the Display component implements. The Display component frame defines its requires and provides interfaces Count and Signal. Since both Countdown and Display are primitive components in SOFA, so their architectures only specify their version rather than their structure.

In general, in SOFA, the provided services of a SOFA component is its provides interfaces and the required services of a SOFA component is its requires interfaces as shown in Figure 37 (a). Alternatively, provides interfaces can also be system provided services, requires interfaces can also be system required services. A SOFA component may behave like a system by itself (Figure 37 (b)).

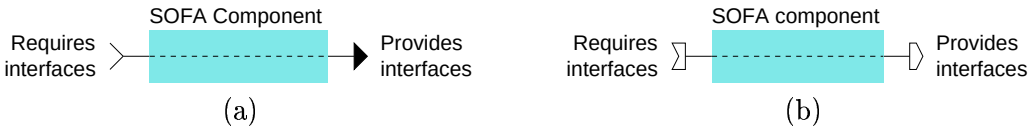


Figure 37: Required and provided interfaces of SOFA components.

3.6.2 Composition of SOFA Components

In SOFA, components are constructed in the builder tool SOFAnode. The repository of SOFA components is the Template Repository. In the design phase, SOFA component composition is by method calls through connectors. There are three types of predefined connectors: CSProcCall, EventDelivery and DataStream, and users can define their own connectors as well. A combination of components can be a composed component, which is a composite component and can also be a system, which is a system template.

In SOFA, connector is represented as an arrow, the direction of an arrow from a requires interface to a provides interface represents the direction of method calls. So, a composite component is represented as shown in Figure 38.

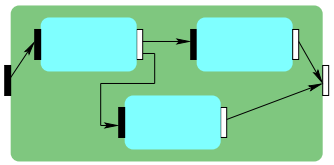


Figure 38: Composition of SOFA components.

In the deployment phase, SOFAnode provides the run-time environment for SOFA components.

Example 3.6.2 Consider the Stopwatch device in Example 3.6.1 again. In the design phase, the Stopwatch device (Figure 39) is implemented by constructing a new Countdown component

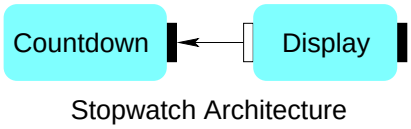


Figure 39: Stopwatch device.

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, frames and architectures.) 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 (Figure 40). The definition files for Countdown and Stopwatch are deposited into the Template Repository.

In the deployment phase, the definition files of Stopwatch, Countdown and Display are compiled by the SOFA CDL compiler to its default implementation, and it is complied to binary code for Stopwatch.

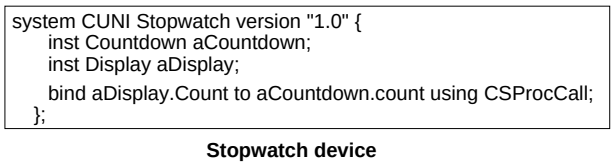


Figure 40: Example of composition of SOFA components.

3.6.3 Summary

In SOFA, a component is a unit of design which has a specification and an implementation. SOFA components are constructed as component definition files and deposited into Template Repository of SOFAnode, which is the repository of SOFA components. In the design phase, SOFA components are composed by method calls through connectors. SOFA components can also be retrieved from Template Repository and composed with other components to a composite component or a system template that is then deposited back to the repository in the design phase. In the deployment phase, no new composition is possible, so there is no assembler, SOFAnode provides the run-time environment for SOFA components.

3.7 KobrA

3.7.1 The Semantics and Syntax of KobrA Components

In KobrA Component Model, a component is a UML component. Every KobrA component has a specification, which describes what a component does, and an implementation, which describes how it does it. The specification of a KorbrA component describes all the properties of its instances that are visible to other component instances, including the set of services that the instances make available to other components (supplied or server interface), and the set of server instances that the component instances need to acquire (imported, supplied or used interface). So, the specification of a component is the interface of this component. The specification of a component also defines the behaviour of the component including its local state.

In KobrA, a component is represented in UML notation that is used as a kind of architecture description language.

Example 3.7.1 For example, consider a book store which wishes to maintain a database of its book stock and sell its books by an Automatic Teller Machine (ATM). Suppose books details can be added to the database by any shop assistant. The specification of the BookStore

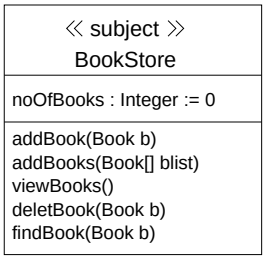


Figure 41: BookStore component.

component is in UML class diagram that specifies what the BookStore component does (Figure 41), it specifies the signature of methods that this BookStore component implements and also its local state (instance variable noOfBooks).

In general, the provided services of a KobrA component is its supplied or server interfaces. The required services of a KobrA component is its imported, supplied or used interfaces (Figure 42 (a)). Alternatively, imported, supplied or used interfaces can also be system required services,

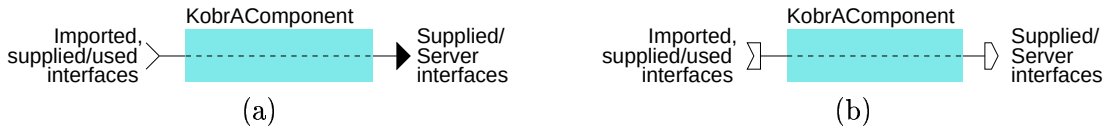


Figure 42: KobrA components.

supplied or server interfaces can also be system provided services. A KobrA component may behave like a system by itself (Figure 42 (b)).

3.7.2 Composition of KobrA Components

In KobrA, components are composed by direct method calls in the design phase. KobrA components can be constructed in a visual builder tool such as Visual UML [52]. The repository of KobrA components is a file system that stores a set of UML diagrams. It is not possible to form a composite component in KobrA.

In the deployment phase, component implementations can be refined from their specifications. No new composition of component instances is possible.

Example 3.7.2 Consider the book store in Example 3.7.1 again, suppose the book store system is being implemented by constructing a new ATM component and composing it with BookStore and Book components from the repository. The specifications of the BookStore and Book component are retrieved from the repository. Then the specification of the ATM component is constructed. The components ATM, BookStore and Book are composed by direct

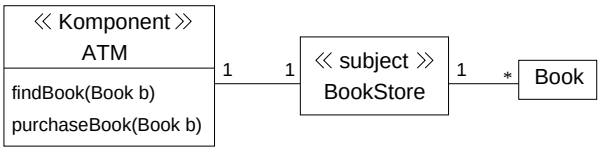


Figure 43: Example of composition of KobrA components.

method calls to a book store system (Figure 43).

In general, KobrA component model is used to build a product family from repositories of pre-existing components, i.e. product lines. KobrA systems are represented with many commonalities and few differences. This idea captures the variability of system which enables it to cope with evolution.

3.7.3 Summary

In KobrA, a component is a UML component. KobrA components are constructed as component specifications and deposited into the repository, which is a file system that stores a set of UML diagrams. In the design phase, KobrA components are composed by direct method calls. KobrA components can also be retrieved from the repository and composed with other components to a system template that is then deposited back to the repository in the design phase. It is not possible to form a composite component in KobrA. In the deployment phase, no new composition is possible, so there is no assembler, and KobrA component implementations are refined from their specifications into a programming language and executed in the run-time environment of that language.

3.8 Architecture Description Languages

3.8.1 The Semantics and Syntax of ADL Components

In Architecture Description Languages (ADLs), a component is an architectural unit that represents a primary computational element and data store of a system. Obviously in all ADLs, components are defined in architecture description languages.

Component interfaces are defined by a set of ports through which their functionalities are exposed. Each port identifies a point of interaction between the component and its environment. A component may have multiple interfaces by using different types of ports, which express a set of operations available on that component (Figure 44). Types are mechanisms for the

Figure 44: An ADL component.

specification of recurring component, connector, port and role structures. Types of components and their ports can be defined in a family that is a set of type definitions, which define the design vocabulary of a system. Both functional and extra-functional attributes of a component can be specified by property types defined in that component.

Example 3.8.1 Consider a simple bank system which has just one ATM that serves one bank consortium. The bank consortium has two bank branches Bank1 and Bank2. This bank system is implemented by the ATM, BankConsortium, Bank1 and Bank2 components in Acme (Figure 45). Components ATM, BankConsortium, Bank1 and Bank2 specify their ports, and Bank1 and Bank2 specify their bankid properties.

In general, the required services of a component are input ports and the provided services of a component are output ports in ADL (Figure 46 (a)). Alternatively, input ports can also be system required services and output ports can also be system provided services. An ADL component can behave like a system by itself (Figure 46 (b)).

3.8.2 Composition of ADL Components

In ADLs, components are composed by connectors that model communication and interaction between components. Connectors mediate the communication and coordination activities

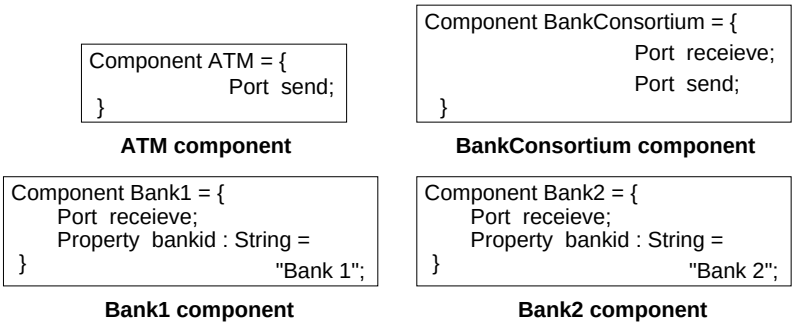


Figure 45: Example of an ADL component in Acme.

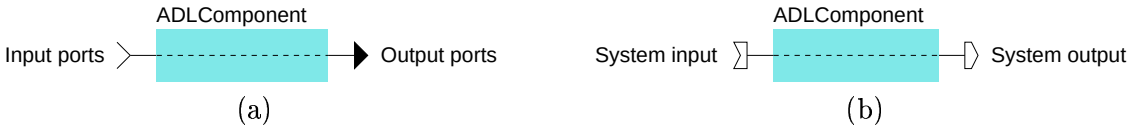


Figure 46: ADL components.

among components. Similar to components, connector interfaces are defined by a set of roles. Each role of a connector defines a participant of the interaction represented by the connector. There are both binary and n-nary connectors. A connector may have multiple interfaces by using different types of roles. The types of connectors and their roles may be defined in a family. Both connector and role attributes can be specified by property types defined in that connector.

In ADLs, there is no repository. In the design phase, components and connectors are (possibly) constructed in a visual builder tool, e.g. AcmeStudio [13]. Unlike other component models, component instances and their composition are not always defined, and their implementation is not always specified. In the deployment phase, the implementation of components and connectors can be done in various programming languages, and so the run-time environment in the deployment phase is that for the chosen programming language.

Example 3.8.2 Consider the simple bank system in Example 3.8.1 again. In the design phase, the architecture for the whole system is designed (Figure 47). This is done by using

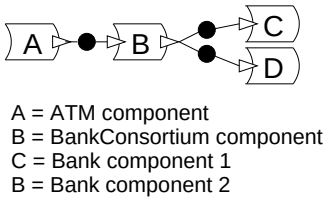


Figure 47: Architecture of bank system.

components and connectors in Acme (Figure 48): components are units of computation (and storage), whereas connectors define the interactions between the units (Figure 49).

In the deployment phase, implementations of the components and connectors in the system are constructed from scratch, or alternatively mapped from specifications in Acme to implementations in ArchJava [1, 2] that can be compiled to instances of the components [3]. Then

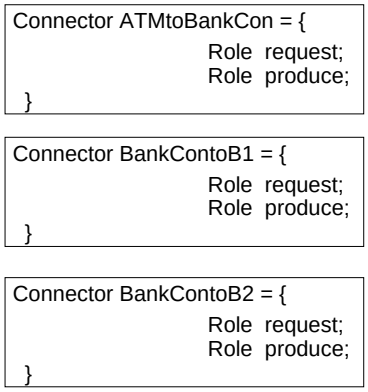


Figure 48: Examples of ADL connectors in Acme.

instances of ATM, BankConsortium, Bank1 and Bank2 are composed within the assembler in the same manner as their composition in the design phase. If ArchJava components are used, then the run-time environment is the Java Virtual Machine.

In general, in the design phase, components can be assembled by connectors to a composite component or a system template.

For a composite component, a binding provides a way of associating a port on that component with some port within its representation (components that form this composite component). System templates may define properties which describe system-level attributes and represent properties of the environment in which systems are operating. The graph of a system (how everything is connected) is defined by a set of attachments. Each attachment represents an interaction between a port and some role of a connector.

The ADL type model dictates that all instances of a type must define the properties declared by the type, and include any structure mandated by the type. Typically, a family may define a set of design rules that constrain how components can be assembled together by a set of connectors, and they are encoded as properties, for using the family.

An Open Semantic Framework may be provided for reasoning about ADL specification such as Acme [27]. In this framework, an ADL specification represents a derived predicate, called its prescription. This predicate can be reasoned about using standard first-order logical machinery or it can be compared for fidelity with real world artifacts that the specification is intended to describe.

3.8.3 Summary

In Architecture Description Languages (ADLs), a component is an architectural unit that represents a primary computational element and data store of a system. Obviously in all ADLs, components are defined in architecture description languages. Components are composed by connectors that mediate the communication and coordination activities among components. In ADLs, there is no repository. In the design phase, components and connectors are constructed possibly in a visual builder tool. Components can be assembled by connectors to a representation (composite component) or a configuration (system template) in design phase. In the deployment phase, no new composition is possible, so there is no assembler, the implementation of components and connectors can be done in various programming languages, and so the

```
System BankSys = {
   Component ATM = {
                  Port  send;
    };
   Component BankConsortium = {
                  Port  receieve;
                  Port  send;
    };
   Component Bank1 = {
                  Port  receieve;
                   Property  bankid : String =
                                          "Bank 1";
    };
   Component Bank2 = {
                  Port  receieve;
                   Property  bankid : String =
                                          "Bank 2";
    };

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

   Connector BankContoB1 = {
                   Role  request;
                    Role  produce;
    };
   Connector BankContoB2 = {
                   Role  request;
                    Role  produce;
    };
    Attachments {
          ATM.send to ATMtoBankCon.request;
          ATMtoBankCon.produce to BankConsortium.receieve;
          BankConsortium.send to BankContoB1.request;
          BankContoB1.produce to Bank1.receieve;
          BankConsortium.send to BankContoB2.request;
          BankContoB2.produce to Bank2.receieve;
    }
}
```

Figure 49: Example of composition of ADL components in Acme.

run-time environment in the deployment phase is that for the chosen programming language.

3.9 UML 2.0

3.9.1 The Semantics and Syntax of UML 2.0 Components

In UML2.0, a component is a modular unit of a system with well-defined interfaces that is replaceable within its environment. A component defines its behaviour by one or more required and provided interfaces (ports), every required service is represented by a socket and every provided service is represented by a lollipop as shown in Figure 50. The required and provided interfaces specify a formal contract of services for the component that requires from other components in the environment or provides to its clients. Thus, they are the only access points to components. In other words, components encapsulate services through their required and provided interfaces. In UML2.0, a component is represented in UML notation that is used as a kind of architecture description language.

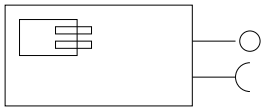


Figure 50: A UML component.

Example 3.9.1 Consider a simple bank system which has just one ATM that serves one bank consortium. The bank consortium has two bank branches Bank1 and Bank2. This bank system is implemented by ATM, BankConsortium, Bank1 and Bank2 components (Figure 51). Components ATM, BankConsortium, Bank1 and Bank2 specify their required and provided

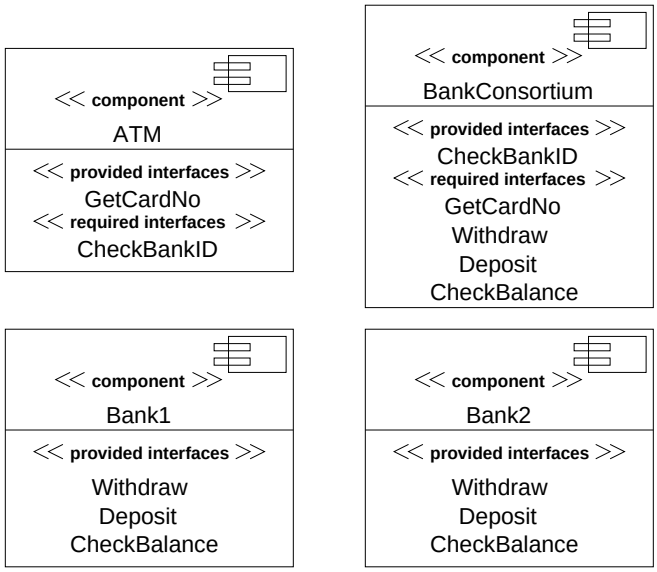


Figure 51: Examples of UML 2.0 components.

interfaces.

In general, the required services of a component are defined by its required interfaces and the provided services of a component are defined by its provided interfaces (Figure 52 (a). Alternatively, provided interfaces can also be the system provided services and required inter-

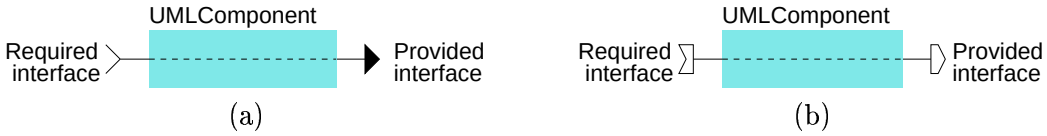


Figure 52: Required and provided services of UML 2.0 components.

faces can also be the system required services. A UML 2.0 component can behave like a system by itself (Figure 52 (b)).

3.9.2 Composition of UML 2.0 Components

In UML2.0, components can be constructed in a visual builder tool such as Visual UML. In the design phase, UML components are composed by UML connectors (Figure 53). Generally,

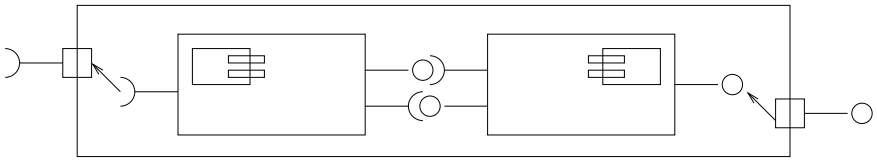


Figure 53: Composition of UML 2.0 components.

there are two kinds of connectors:

- **Assembly connector** is used to connect the required interface of a component to the provided interface of another component.
- **Delegation connector** is used to forward requested operations from the environment of a component and provides services outside the component.

Like some ADLs, UML2.0 only specifies components and connectors, but does not provide support for their implementation in the deployment phase.

Example 3.9.2 Consider the simple bank system in Example 3.9.1 again. In the design phase, the architecture for the whole system is designed. This is done by using components and assembly connectors (Figure 54).

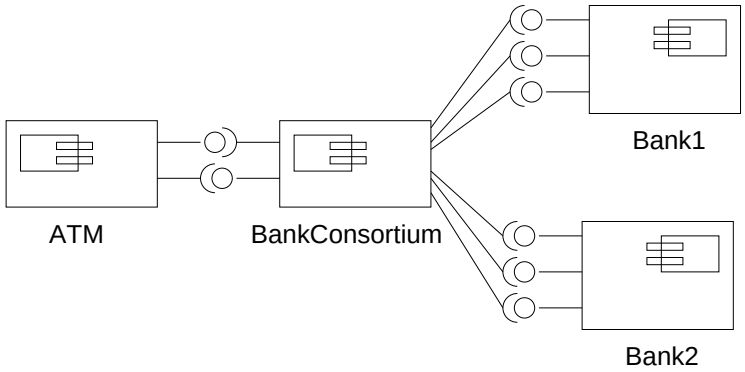


Figure 54: Example of composition of UML components.

In the deployment phase, implementations of the components and connectors in the system are constructed from scratch, or alternatively mapped from specifications in UML2.0 to implementations in a programming language, and so the run-time environment in the deployment phase is that for the chosen programming language.

3.9.3 Summary

In UML 2.0, a component is a modular unit of a system with well-defined interfaces that is replaceable within its environment. Components are represented in UML notation that

is used as a kind of architecture description language. In UML 2.0, there is no repository. In the design phase, components can be constructed in a visual builder tool such as Visual UML. Components are composed by UML connectors: delegation connectors and assembly connectors. Components can be composed by assembly connectors to a composite component or by delegation and assembly connectors to a system template. In the deployment phase, no new composition is possible, so there is no assembler, the implementation of components and connectors can be done in various programming languages, and so the run-time environment in the deployment phase is that for the chosen programming language.

3.10 PECOS

3.10.1 The Semantics and Syntax of PECOS Components

In PECOS (Pervasive Component Systems) Component Model, 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. Components are composed by linking their ports with connectors. The model is used for specifying and developing embedded systems of field devices represented as software components.

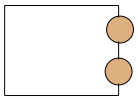


Figure 55: A PECOS component in PECOS notation.

Every component in PECOS has a name, a number of property bundles,[9] a set of ports, and behaviour. In PECOS, a component is represented by a box with ports (Figure 55). 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, viz. *in*, *out* and *inout*. A port can only be connected to another port having the same type and complementary direction.

The behaviour 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. Depending on how their behaviour is triggered and where they are run, components are classified into three kinds:

- **Passive Components**
  A passive component does not have its own thread of control. Passive components are typically used to encapsulate a piece of behaviour that executes synchronously and completes in a short time-cycle.

- **Active Components**
  An active component is a component with its own thread of control. Active components are typically used to model either very fast or very slow activities such as reading out hardware registers or writing to slow memory.

[9] Used to store meta information about the component, such as worst-case execution times, memory consumption, or scheduling information.

- **Event Components**
  An event component is like an active component, but the execution of the behaviour is triggered by an event. Whenever the event fires, the behaviour is executed immediately.

In PECOS, the CoCo language (Component Composition Language) [28] is used to specify components. Like ADLs, CoCo specifies only the properties and ports of a component,[10] but not its behaviour. The behaviour of a component has to be filled in (implemented) by the programmer.

Example 3.10.1 A clock component may be specified in CoCo as follows [28]:

```
component Clock {
   properties {
     memSize = 32;
     description = "This is my first clock.";
   }
   output long msecs;
}
```

This is a passive component with two properties attached, viz. memSize and description, and an output port msecs of type long.

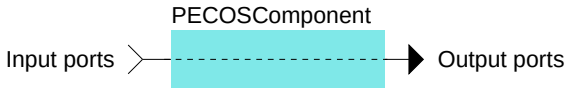


Figure 56: PECOS components.

The PECOS component model is thus data-flow oriented: components exchange data with their environment (and hence other components) through ports. The ports of a component are the only means of interaction with other components. Thus a component's interface is given by its set of ports. The required services of a component are its input ports and the provided services of a component are its output ports (Figure 56). Note, however, that because ports are data channels, there may be no explicit dependency between input and output ports. Note also that components may have no input ports or no output ports. For example, a component like a clock in Example 3.10.1 simply outputs time at regular intervals, without any inputs, whilst a display unit may take inputs from the clock, display the time, but not send any data to another component.

3.10.2 Composition of PECOS Components

In PECOS, there is no component repository. Each system or component is specified in CoCo in a top-down manner, in terms of compositions of sub-components. Since CoCo does not specify the behaviour of components, the CoCo specification of the entire system is a syntactic specification of the composition of the sub-components. Since there is no repository, this composition is design-time composition, rather than repository phase composition. The (sub)components have no implementation at this stage.

[10] And connectors, in the case of composite components.

Components are composed by 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 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, i.e. connectors may not cross component boundaries.

A complete system typically represents a device running in a control loop. It must have a schedule that specifies the order in which its behaviour and the behaviour of its sub-components are run. Therefore, in the deployment phase, to realise an executable system from its CoCo specification, the behaviour of sub-components has to be implemented by the programmer, and a schedule must then be provided for the system, and any subsystem that contains sub-components. No new composition is possible at this stage.

Example 3.10.2 Figure 57 shows a PECOS system, called Device, with the following CoCo specification:

```
active component Device {
   Clock clock;
   Display display;
   DigitalDisplay digitalDisplay;
   EventLoop eventLoop;

   connector time (clock.msecs, display.time, digitalDisplay.time_in_msecs);
   connector eventLoop_started (eventLoop.started, digitalDisplay.can_draw);
}
```

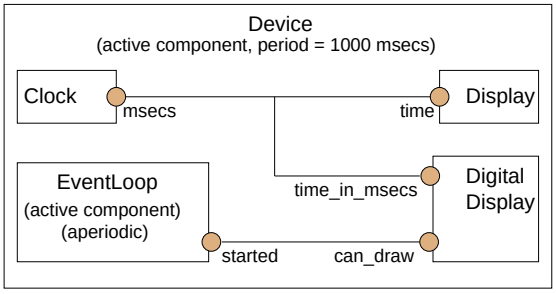


Figure 57: Example of design time composition in PECOS.

There are four sub-components in this device: Clock, Display, EventLoop and DigitalDisplay. The Clock component is used to provide the time. The Display component is used to display the time. The EventLoop component is an active component used to handle graphical events such as mouse click and repaint events. The DigitalDisplay component is used to display the time digitally. The CoCo specification of these components are as follows:

```
component Clock {
    output long msecs;
}

active component EventLoop {
```

```
component Display {
    input long time;
}

component DigitalDisplay {
```

```
    output bool started;                        input long time_in_msecs;
}                                               input bool can_draw;
                                            }
```

The system Device has no ports because it is the application to be run. The application is complete when all the components have been implemented (in Java or C++), and a schedule has been provided for Device.

3.10.3 Summary

In PECOS, a component is a unit of design which has a specification and an implementation. Components are defined in an ADL-like language called CoCo and constructed in a programming environment such as Eclipse. In the design phase, components are composed by linking their ports with connectors. There is no repository in PECOS. Implementation of PECOS components is usually done in Java or C++, and so the run-time environment in the deployment phase is that for Java or C++.

3.11 Pin

3.11.1 The Semantics and Syntax of Pin Components

In Pin component model, a component is an architectural unit that specifies a stimulus-response behaviour by a set of ports (pins). Components are defined in CCL (Construction and Composition Language) [55] that is essentially an architecture description language. In Pin, a component is represented by a set of sink pins and source pins together with the component's behaviour (Figure 58). Like ports of ADLs, pins are the interaction points of components. Sink pins are

Figure 58: A Pin component in Pin notation.

used to receive communication (stimuli) and source pins are used to initiate communication (responses) with its environment. Each pin has a data interface that describes the data type, which has a parameter-passing mode, where mode is one of In, Out, InOut. The behaviour of a component is described by parallel or interleaved composition of its reactions that specify the stimulus-response behaviour of a component on its sink and source pins.

Example 3.11.1 Consider a simple component AComp as shown in Figure 59. The component AComp is specified with both structural and behavioural aspects in CCL. Structurally, it has one asynchronous sink pin, receive, and two synchronous (unicast) source pins, send and publish. Behaviourally, it has a threaded reaction mission (a thread is a unit of concurrency) that takes receive, send and publish as parameters. A threaded reaction represents state transitions: when an event is observed, the corresponding action is taken.

In general, the required services of a component in Pin are sink pins and the provided services are source pins (Figure 60 (a)). Alternatively, sink pins can also be system required

```
component AComp() {
      sink asynch receive();
      source unicast send();
      source publish();
      threaded react mission (
         receive, send, publish) {
        start –> ready { }
        ready –> work {
            trigger ^receive();
            action ^send();
          }
        work –> log {
            trigger ^send();
            action ^publish();
          }
        log –> ready {
            trigger ^publish();
            action ^receive();
          }
       }
}
```

Figure 59: An example of a Pin component.

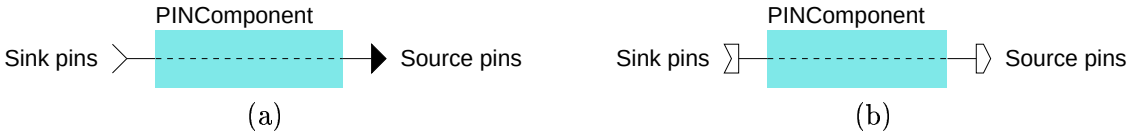


Figure 60: Required and provided services of Pin components.

services, source pins can also be system provided services. A component in Pin component model may behave like a system by itself (Figure 60 (b)).

3.11.2 Composition of Pin Components

In Pin, components can not directly interact with each other. Components always access other components through their pins. In the design phase, components are composed by connectors that link source pins of one component to the sink pins of another (Figure 61). Generally there

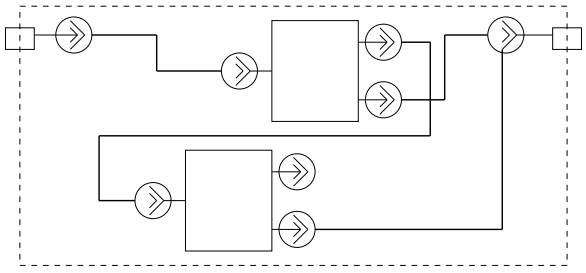


Figure 61: Composition of Pin components.

are two kinds of connectors in Pin:

- **Synchronous connector** is used to connect one source pin of a component to one sink pin of another;

- **Asynchronous connector** is used to connect one source pin of a component to multiple sink pins of another.

An assembly (composite component) can be composed by components and connectors in the design phase. The source and sink pins of an assembly defines the services that are provided to and required from its environment. Only components within the same assembly can interact with each other. The pins of an assembly are connected with the pins of its components by assembly junctions. Generally there are two kinds of assembly junctions:

- **null junctions** are used to connect components having the same environment type;

- **gateway junctions** are used to connect components in different types of environments.

In Pin, components and connectors are specified in CCL and so their implementations are usually generated by the CCL processor. There is no repository in Pin. In the deployment phase, components are executed in the Pin run-time environment.

Example 3.11.2 Consider the simple bank system in Example 3.11.1 again. In the design phase, components comp1 and comp2 of type AComp are composed by connectors that link comp1's source pin, send, to comp2's sink pin, receive, to an assembly AComposite. Both comp1 and comp2 function within the same environment (viz. E) as the assembly AComposite (Figure 62).

```
enviroment E {
    service Receive() {
        source unicast receive();
        threaded react received(receive) {
            start –> ready {};
            ready –> work {;
                action ^receive;
            }
            work –> ready {};
    }
    service Send() {
        sink asynch send();
        threaded react sending(send) {
            start –> ready {};
            ready –> work {;
                action ^send;
            }
            work –> ready {};
    }
}
```

```
assembly AComposite() (E) {
    assume {
        E: Send compositesend();
        E: Receive compositereceive();
    }
    AComp comp1(), comp2();
    compositesend:receive ~> comp1:receive;
    comp1:send ~> comp2:receive;
    comp1:publish ~> compositesend;
    comp2:publish ~> compositesend;
}
```

Figure 62: Example of composition of Pin components.

In the deployment phase, implementations of the components and connectors are generated by the CCL processor and executed in the Pin run-time environment.

3.11.3 Summary

In Pin, a component is an architectural unit that specifies a stimulus-response behaviour by a set of ports (pins). Components are defined in CCL that is essentially an architecture description language. In the design phase, components are composed by connectors that link source pins of one component to the sink pins of another. There is no repository in Pin. In the deployment phase, implementations are usually generated by the CCL processor and components are executed in the Pin run-time environment.

3.12 Fractal

3.12.1 The Semantics and Syntax of Fractal Components

In Fractal, a component is a run-time entity that behaves like an object. Interface Definition Languages (e.g. OMG IDL) are used to define generic interfaces that can be implemented by components in specific programming languages. Current Fractal API is extended and modified from Java API with JavaBeans-like introspection facilities.

A Fractal component comprises a content and a controller (Figure 63). The content of a

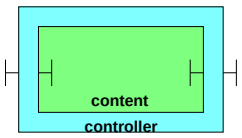


Figure 63: A Fractal component.

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 behaviour associated with this component. In particular, it intercepts incoming and outgoing operation invocations and operation returns targeting or originating from the component's content it controls.

A Fractal component can implement multiple interfaces. A parametric component is a component whose attributes can be set by its clients. A parametric component should implement an AttributeController interface.

Example 3.12.1 For example, consider a Stopwatch device that is used to count down from a specific number, e.g. 100. The Stopwatch device comprises a Countdown component and a Display component (Figure 64). The component Countdown provides an interface to output the number that it is counting. It can be parameterised by one attribute: a "total" attribute to configure the initial number that it starts to count down from. The other component Display uses the Countdown component to print the number that the Countdown component is counting.

In general, in Fractal component model, functions defined in a component's interfaces are services provided by the component, whilst external function calls from other components via interfaces are required services (Figure 65 (a)). Of course, external function calls may be made by client applications and the results returned to the clients, in which case a component behaves like a complete system, with client function calls as system required services and the functions it implements as system provided services (Figure 65 (b)).

```
public interface Count {
    void count ();
}
public interface ControlTotal extends AttributeController {
    int getTotal ();
    void setTotal (int total);
}
public class Countdown implements Count, ControlTotal {
    private int total = 0;
    public void count () {
        for (int i = total; i > 0; i--) {
            System.out.print(i);
        }
    }
    public int getTotal () {
        return total;
    }
    public void setTotal (final int total) {
        this.total = total;
    }
}
```

Countdown component

```
public interface Signal {
    void display ();
}
public class Display implements Signal, BindingController {
    private Count count;
    public void display () {
        count.count();
    }
    public String[] listFc () {
        return new String[] {"c"};
    }
    public Object lookupFc (final string disstr) {
        if (disstr.equals("c")) {
            return count;
        }
        return null;
    }
    public void bindFc (final string disstr, final Object countobj) {
        if (disstr.equals("c")) {
            count = (Count)countobj;
        }
    }
    public void unbindFc (final string disstr) {
        if (disstr.equals("c")) {
            count = null;
        }
    }
}
```

Display component

Figure 64: Examples of Fractal components.

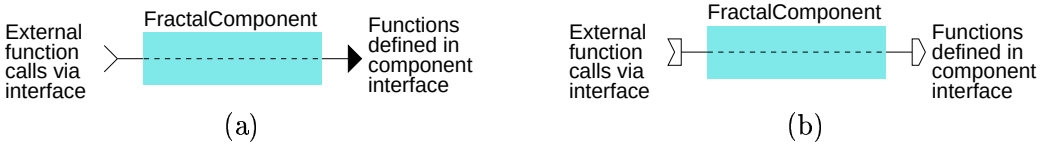


Figure 65: Required and provided interfaces of Fractal components.

3.12.2 Composition of Fractal Components

In Fractal, components are constructed in a programming environment with Fractal APIs. Fractal components are composed by method calls through connectors in the design phase (Figure 66). In the deployment phase, no new composition is possible, so there is no assembler, the Java Virtual Machine serves as the run-time environment for Fractal components.

Example 3.12.2 Consider the Stopwatch device in Example 3.12.1 again. In the design phase, the Stopwatch device (Figure 67) is implemented by constructing and composing Countdown component and Display component. Component instances (objects) are created and instantiated from component templates. The instances of Countdown and Display are composed by method calls (Figure 68).

In the deployment phase, no new composition of component instances is possible, and the Java Virtual Machine serves as the run-time environment for Fractal components.

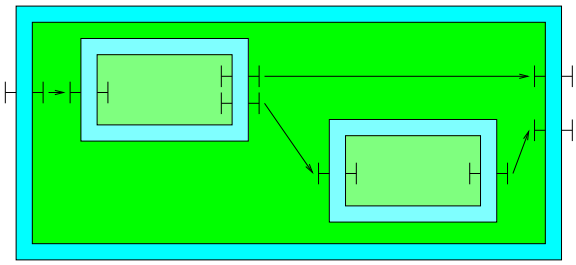


Figure 66: Composition of Fractal components.

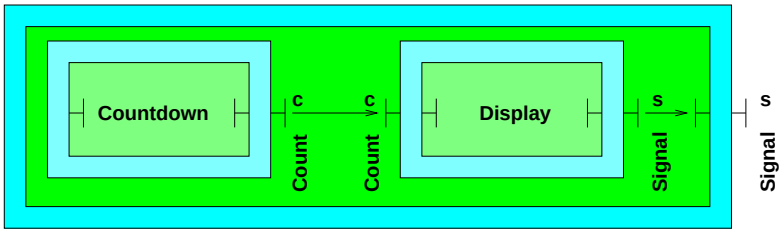


Figure 67: Stopwatch device.

3.12.3 Summary

In Fractal, a component is a run-time entity that behaves like an object. Components are constructed in a programming environment with Fractal APIs. In the design phase, Fractal components are composed by method calls through connectors. In the deployment phase, no new composition is possible, so there is no assembler, and Java Virtual Machine provides the run-time environment.

4 Towards A Taxonomy

Having described current software component models in a uniform way, i.e. with reference to the abstract model (Section 2), we can now attempt a taxonomy. Clearly, we can classify existing models according to component syntax, semantics or composition. This classification should provide the basis for a meaningful taxonomy. In this section, we first present the three categories based on component syntax, component semantics, and component composition, and then argue that a meaningful taxonomy should be based on component composition, because it is central to CBD. We then present such a taxonomy.

4.1 Categories based on Component Syntax

Based on component syntax, current models fall into three categories: (i) models in which components are defined by object-oriented programming languages, (ii) those in which an IDL (interface definition language) is used and in which components can be defined in programming languages with mappings from the IDL; and (iii) those in which components are defined by architecture description languages (Figure 69).

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

```
Component  boot = Fractal.getBootstrapComponent();
TypeFactory  tf = (TypeFactory)boot.getFcInterface("type−factory");
ComponentType  deviceType = tf.createFcType(new InterfaceType[] {
  tf.createFcItfType("s", "Signal", false, false, false);
ComponentType  displayType = tf.createFcType(new InterfaceType[] {
  tf.createFcItfType("s", "Signal", false, false, false),
  tf.createFcItfType("c", "Count", true, false, false)});
ComponentType  countdownType = tf.createFcType(new InterfaceType[] {
  tf.createFcItfType("c", "Count", false, false, false),
  tf.createFcItfType("total−controller", "ControlTotal", false, false, false)});

GenericFactory  cf = (GenericFactory)boot.getFcInterface("generic−factory");

Component deviceTmpl = cf.newFcInstance(deviceType, "deviceTemplate",
                                        new Object[] {"composite", "Device"});

Component displayTmpl = cf.newFcInstance(displayType, "displayTemplate",
                                         new Object[] {"primitive", "Display"});

Component countdownTmpl = cf.newFcInstance(countdownType, "countdownTemplate",
                                   new Object[] {"parametricPrimitive", "Countdown"});

ControlTotal ct = (ControlTotal)countdownTmpl.getFcInterface("total−controller");
ct.setTotal(100);

ContentController cc = (ContentController)deviceTmpl.getFcInterface("content−controller");
cc.addFcSubComponent(displayTmpl);
cc.addFcSubComponent(countdownTmpl);

((BindingController)deviceTmpl.getFcInterface("binding−controller")).bindFc("s", displayTmpl.getFcInterface("s"));
((BindingController)displayTmpl.getFcInterface("binding−controller")).bindFc("c", countdownTmpl.getFcInterface("c"));

Component stopwatchdevice = ((Factory)deviceTmpl.getFcInterface("factory")).newFcInstance();
(LifeCycleController)stopwatchdevice.getFcInterface("lifecycle−controller")).startFc();
((Signal)stopwatchdevice.getFcInterface("s").display();
```

Stopwatch Device

Figure 68: Example of composition of Fractal components.

Component models that belong to (ii) are COM, CCM and Fractal. These models use IDLs to define generic interfaces that can be implemented by components in specific programming languages. COM uses the Microsoft IDL [8], CCM uses the OMG IDL [40], whereas Fractal can use any IDL.

Component models that belong to (iii) are ADLs, UML2.0, KobrA, Koala, SOFA, PECOS and Pin. Obviously in all ADLs, components are defined in architecture description languages. In UML2.0 and KobrA, the UML notation is used as a kind of architecture description language, 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, whilst in Pin, components are defined in CCL. CoCo and CCL are composition languages that are essentially architecture description languages.

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

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

Figure 69: Categories based on component syntax.

4.2 Categories based on Component Semantics

Based on semantics, component models can be grouped into three categories: (i) component models in which components are classes; (ii) models in which components are objects; and (iii)

Component Semantics	Models
Classes	JavaBeans, EJB
Objects	COM, CCM, Fractal
Architectural Units	ADLs, UML2.0, KobrA, Koala, SOFA, PECOS, Pin

Figure 70: Categories based on component semantics.

those in which components are architectural units (Figure 70).

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

Component models that belong to (ii) are COM, CCM and Fractal, since semantically components in these models are run-time entities that behave like objects. In COM, a component is a piece of compiled code that provides some services, that is hosted by a COM server. In CCM, a component is a CORBA meta-type that is an extension and specialisation of a CORBA object, that is hosted by a CCM container on a CCM platform such as OpenCCM. In Fractal, a component is an object-like run-time entity in languages with mappings from the chosen IDL.

Component models that belong to (iii) are ADLs, UML2.0, KobrA, Koala, SOFA, PECOS and Pin. Semantically, components in these models are units of computation and control (and data) connected together 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 that 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. In Pin, a component is an architectural unit that specifies a stimulus-response behaviour by a set of ports (pins).

4.3 Categories based on Component Composition

To define categories based on composition, we first consider composition in an ideal life cycle. This will provide a basis for comparing composition in existing component models. An idealised version of the component life cycle that we described in Section 2 is one where a repository is available in the design phase, and component composition is possible in both the design and the deployment phases.

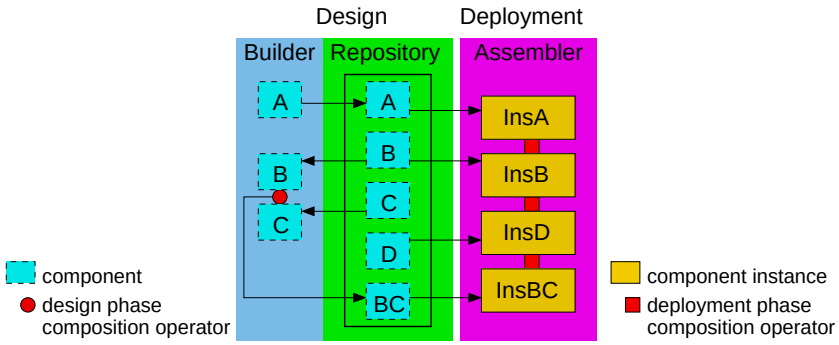


Figure 71: The idealised component life cycle.

It is depicted in Figure 71. We believe this view of the ideal component life cycle is the commonly accepted one in CBD, and not just our own subjective view: all existing component models reflect this ideal life cycle to varying degrees, as we will show later.

In the design phase of the idealised life cycle, a *builder* tool can be used to (i) construct new components, and then deposit them in the *repository*, e.g. A in Figure 71; (ii) retrieve components from the repository, compose them and deposit them back in the repository, e.g. in Figure 71, B and C are composed into a composite BC that is deposited in the repository.

In the deployment phase of the idealised life cycle, instances of components in the repository are created, and an *assembler* tool can be used to compose them into a complete system, e.g. in Figure 71, instances of A, B, D and BC are created and composed into a system. The system is then executable in the run-time environment of the deployment phase.

The idealised component life cycle 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, whilst some models do; some have composition in the deployment, whilst

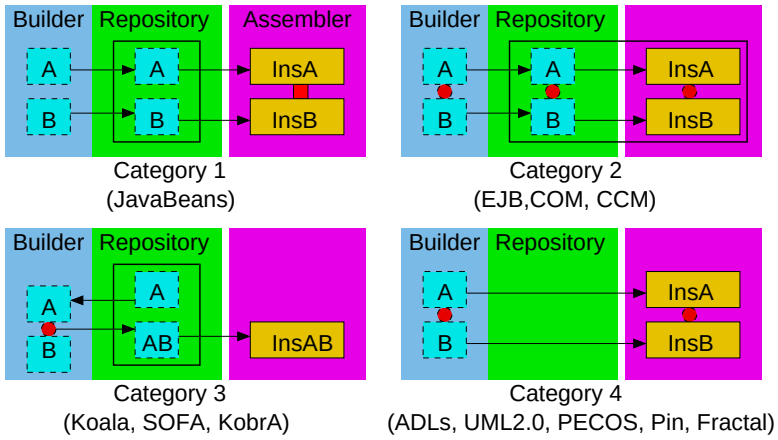


Figure 72: Categories based on component composition.

some do not. Thus many categories are possible. Figure 72 gives four categories that cover all major existing component models.

In Category 1, in the design phase, there is a repository but there is no composition: compo-

nents are constructed individually by the builder and stored separately in the repository. The builder can only construct new components, and cannot retrieve components from the repository. In the deployment phase, components are retrieved from the repository, and instantiated, and the assembler can be used to compose the instances. The sole member of this category is JavaBeans.

In Category 2, in the design phase, there is also a repository, but composition is possible. Like in Category 1, the builder can only construct new components, and cannot retrieve components from the repository. Moreover, no composite component can be stored in the repository. Therefore composition has to be performed by the builder and then has to be stored 'as is' in the repository, i.e. as a set of individual components together with the links between them defined by the composition. As a result, this composition has to be retained even in the deployment phase, since it is only possible to instantiate the individual components (and not their composite). Consequently, composition is not possible, and therefore there is no assembler, in the deployment phase. This category includes EJB, COM and CCM.

Category 3 is the same as Category 2 except that in Category 3 the builder can retrieve (composite) components from the repository, e.g. in Figure 72, A is retrieved from the repository; and the repository can store composite components, e.g. in Figure 72, AB is a composite. No new composition is possible in the deployment phase, and so there is no assembler. Koala, SOFA and KobrA belong to this category.

In Category 4, there is no repository. In the design phase, the builder has to construct a complete system of components and their composition. Unlike the other categories, where component instances are well-defined, in Category 4 component instances and their composition are not always defined, and their implementation is not always specified (with the exception of Fractal). Therefore in the deployment phase, the task of implementing the whole system often remains. All ADLs belong to this category, as well as ADL-like models, viz. UML2.0, PECOS, Pin and Fractal.

4.4 A Taxonomy of Software Component Models

The three groupings of categories in the previous section are based on syntax, semantics and composition. The question is whether it is possible or meaningful to combine them into a single taxonomy. Looking at the categories based on syntax (Figure 69) and those based on semantics (Figure 70), it is obvious that they can be merged straightforwardly into two groups:

- *object*-based: JavaBeans, EJB, COM, CCM and Fractal;
- *architecture*-based: ADLs, UML2.0, KobrA, Koala, SOFA, PECOS and Pin.

However, comparing these two groups with the categories based on composition in the component life cycle (Figure 72), 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 and CCM 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 ADLs, UML2.0, PECOS and Pin 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 ADLs, UML2.0, PECOS and Pin.

In view of this, we believe the only meaningful taxonomy is one based on composition in the component life cycle. Composition is the central issue in CBD after all. Moreover, in the ideal 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 towards one phase or the other. In object-based models like COM, CCM and Fractal, where components are objects that are executable binaries and are therefore more deployment phase entities than design phase entities. On the other hand, in architecture-based models like ADLs and UML2.0, components are expressly design entities by definition, with or without well-defined instances in the deployment phase.

So we propose the taxonomy of software component models shown in Figure 73, based on

Category	Models	Characteristics				
		DR	RR	CS	DC	CP
1	JavaBeans	✓	×	×	×	✓
2	EJB, COM, CCM	✓	×	✓	×	×
3	Koala, SOFA, KobrA	✓	✓	✓	✓	×
4	ADLs, UML2.0, PECOS, Pin, Fractal	×	×	✓	×	×

DR In design phase new components can be deposited in a repository
RR In design phase components can be retrieved from the repository
CS Composition is possible in design phase
DC In design phase composite components can be deposited in the repository
CP Composition is possible in deployment phase

Figure 73: A taxonomy based on composition.

component composition in the ideal component life cycle, as discussed in the last section.

In Category 1, 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, i.e. 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 formed and composed.

In Category 2, in the design phase, new components can be deposited in a repository, but cannot be retrieved from it. Composition is possible, i.e. 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; the composition of the component instances is the same as in that of the components in the design phase.

In Category 3, 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.

In Category 4, 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 is the same as in that of the components in the design phase.

4.5 Discussion

The basis for the taxonomy in Figure 73 is the ideal component life cycle, discussed in Section 4.3. This can be justified by the commonly accepted desiderata of CBD: (i) components

are pre-existing reusable software units – this necessitates the use of a repository; (ii) components can be produced and used by independent parties – this requires builder and assembler tools that can interact with a repository; (iii) components can be copied and instantiated – this means components should be distinguished from their instances, and hence the distinction between the design phase and the deployment phase; (iv) components can be composed into composite components which in turn can be composed with (composite) components into even larger composites (or subsystems), and so on – this requires that composites can be deposited in and retrieved from a repository, just like any components. All the models in the taxonomy reflect these criteria, to greater or lesser degrees.

Given this, it is interesting to note that models in Category 3 meet the requirements of the ideal 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 [19]. The main reason for its success is precisely its use of repositories of families of pre-existing components, i.e. product lines.

At the other end of the scale, models in Category 4 do not 'perform' so well mainly because they are ADL-based and are therefore focused on designing (systems and) components from scratch, rather than reusing existing components.

Models in Categories 1 and 2 are 'middle of the road'. They also use repositories, but they behave differently from those in Category 3 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.

Finally, the taxonomy reveals that no existing model has composition in both the design and the deployment phase. No model can retrieve composites for further composition in the deployment phase, not even those in Category 3. So there is room for improvement, and better component models are possible. To define such models would require new composition operators, or connectors. For example, we are experimenting with *exogenous* connectors [31] that encapsulate control and thus minimise coupling between components.

5 Conclusion

In this report, we have presented a survey of software component models in which all the models are described in a uniform manner, i.e. all with reference to the same abstract model. Undoubtedly this is an important starting point for clarification and unification of the CBD terminology. In the survey, 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' specific to some component models because the latter do not follow a unified terminology.

The survey provides a clear exposition of each model, with examples to illustrate syntax, semantics and composition. It also provides the basis for a taxonomy. The taxonomy reveals clearly the strengths and weaknesses of existing component models. In addition to what we discussed in the previous section, no existing component model supports predictable assembly [54, 55], which is the ultimate goal of CBD. To address this, new component models have to be developed. The on-going model Pin [21] is one such, and we ourselves, and no doubt others too, are working on other models.

6 Acknowledgements

We wish to thank Ivica Crnkovic, David Garlan, Dirk Muthig, Oscar Nierstrasz, Bastiaan Schonhage and Kurt Wallnau for information and helpful discussions.

References

[1] J. Aldrich, C. Chambers, and D. Notkin. Architectural reasoning in ArchJava. In *Proc. 16th European Conference on Object-Oriented Programming*, pages 334–367. Springer-Verlag, 2002.

[2] J. Aldrich, C. Chambers, and D. Notkin. ArchJava: Connecting software architecture to implimentation. In *Proc. ICSE 2002*, pages 187–197. IEEE, 2002.

[3] J. Aldrich, D. Garlan, B.R. Schmerl, and T. Tseng. Modeling and implementing software architecture with acme and archjava. In *Proc. OOPSLA Companion 2004*, pages 156–157, 2004.

[4] R. Allen and D. Garlan. A formal basis for architectural connection. *ACM Transactions on Software Engineering and Methodology*, 6(3):213–249, 1997.

[5] 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.

[6] BEA Systems *et al.* CORBA Components. Technical Report orbos/99-02-05, Object Management Group, 1999.

[7] C. Blexrud, M. Bortniker, J. Crossland, D. Esposito, J. Hales, W. Hankison, V. Honnaya, T. Huankison, S. Kristich, E. Lee, R. Lhotka, B. Loesgen, S. Mohr, S. Robinson, A. Rofail, B. Sherrell, S. Short, and D. Wahlin. *Professional Windows DNA: Building Distributed Web Applications with VB, COM+, MSMQ, SOAP, and ASP.* Wrox Press Inc, September 2000.

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

[9] 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*, 19(1):49–56, 1998.

[10] E. Bruneton, T. Coupaye, and M. Leclercq. An open component model and its support in Java. In *Proceedings of 7th CBSE*, pages 7–22. Springer -Verlag, 2004.

[11] E. Bruneton, T. Coupaye, and J.B. Stefani. Recursive and Dynamic Software Composition with Sharing. In *Proceedings of 7th International Workshop on Component-Oriented Programming.* ECOOP02, 2002.

[12] E. Bruneton, T. Coupaye, and J.B. Stefani. The Fractal component model. Technical Report Specification V2, ObjectWeb Consortium, 2003.

[13] Carnegie Mellon University. *AcmeStudio 2.1 User Manual.* `http://www-2.cs.cmu.edu/~acme/Manual/AcmeStudio-2.1.htm`.

[14] D. Chappell. *Understanding ActiveX and Ole.* Microsoft Press, January 1996.

[15] D. Chappell. How Microsoft Transaction Server changes the COM programming model. *Microsoft Systems Journal*, January 1998.

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

[17] Y. Choi, O. Kwon, and G. Shin. An approach to composition of EJB components using C2 style. In *Proc. 28th Euromicro Conference*, pages 40–46. IEEE, 2002.

[18] B. Christiansson, L. Jakobsson, and I. Crnkovic. CBD process. In I. Crnkovic and M. Larsson, editors, *Building Reliable Component-Based Software Systems*, pages 89–113. Artech House, 2002.

[19] P. Clements and L.M. Northrop. *Software Product Lines: Practices and Patterns.* Addison Wesley, 2001.

[20] P.C. Clements. A survey of architecture description languages. In *8th Int. Workshop on Software Specification and Design*, pages 16–25. ACM, 1996.

[21] CMU SEI. A Snapshot of the Pin Component Model. `http://www.sei.cmu.edu/pacc/`.

[22] COM web page. `http://www.microsoft.com/com/`.

[23] Microsoft Corporation. *Microsoft Internet Information Server Resource Kit.* Microsoft Press, January 1998.

[24] I. Crnkovic, H. Schmidt, J. Stafford, and K. Wallnau. 6th ICSE workshop on component-based software engineering: automated reasoning and prediction. *ACM SIGSOFT Software Engineering Notes*, 29(3):1–7, May 2004.

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

[26] Eclipse web page. `http://www.eclipse.org/`.

[27] D. Garlan, R.T. Monroe, and D. Wile. Acme: Architectural description of component-based systems. In M. Sitaraman G.T. Leavens, editor, *Foundations of Component-Based Systems*, pages 47–68. Cambridge University Press, 2000.

[28] T. Genssler, A. Christoph, B. Schulz, M. Winter, C.M. 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/`, September 2002.

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

[30] J. Ivers, N. Sinha, and K.C Wallnau. A Basis for Composition Language CL. Technical Report CMU/SEI-2002-TN-026, CMU SEI, 2002.

[31] K.-K. Lau, P. Velasco Elizondo, and Z. Wang. Exogenous connectors for software components. In *Proc. 8th Int. SIGSOFT Symp. on Component-based Software Engineering, LNCS 3489*, pages 90–106, 2005.

[32] M. Lumpe, F. Achermann, and O. Nierstrasz. A formal language for composition. In G. Leavens and M. Sitaraman, editors, *Foundations of Component Based Systems*, pages 69–90. Cambridge University Press, 2000.

[33] A. Major. *COM IDL and Interface Design.* John Wiley & Sons, February 1999.

[34] N. Medvidovic and R. N. Taylor. A classification and comparison framework for software architecture description languages. *IEEE Transactions on Software Engineering*, 26(1):70–93, January 2000.

[35] B. Meyer. The grand challenge of trusted components. In *Proc. ICSE 2003*, pages 660–667. IEEE, 2003.

[36] R. Monson-Haefel. *Enterprise JavaBeans.* O'Reilly & Associates, 4th edition, 2004.

[37] 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. In *Proc. 1st Int. IFIP/ACM Working Conference on Component Deployment*, pages 200–209. ACM Press, 2002.

[38] ObjectWeb – Open Source Middleware. *OpenCCM User's Guide.* `http://openccm.objectweb.org/doc/0.8.1/user_guide.html`.

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

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

[41] T. Pattison. *Programming Distributed Applications with COM+ and Microsoft Visual Basic 6.0.* Microsoft Press, June 2000.

[42] F. Plasil, D. Balek, and R. Janecek. SOFA/DCUP: Architecture for component trading and dynamic updating. In *Proc. ICCDS98*, pages 43–52. IEEE Press, 1998.

[43] F. Plasil, M. Besta, and S. Visnovsky. Bounding Component Behavior via Protocols. In *Proc. Technology of Object-Oriented Languages and Systems 99*, pages 387–398. IEEE, 1999.

[44] F.E. Redmond. *DCOM: Microsoft Distributed Component Object Model.* John Wiley & Sons Inc, September 1997.

[45] Sun Microsystems. *The Bean Builder.* `https://bean-builder.dev.java.net/`.

[46] Sun Microsystems. *Java 2 Platform, Enterprise Edition.* `http://java.sun.com/j2ee/`.

[47] Sun Microsystems. *JavaBeans Architecture: BDK Download.* `http://java.sun.com/products/javabeans/software/bdk_download.html`.

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

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

[50] R. van Ommering. The Koala component model. In I. Crnkovic and M. Larsson, editors, *Building Reliable Component-Based Software Systems*, pages 223–236. Artech House, 2002.

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

[52] Visual Object Modelers. *Visual UML.* `http://www.visualobject.com/default.htm`.

[53] Microsoft Visual Studio Developer Center. `http://msdn.microsoft.com/vstudio/`.

[54] K.C. Wallnau. Volume III: A Technology for Predictable Assembly from Certifiable Components. Technical Report CMU/SEI-2003-TR-009, CMU SEI, 2003.

[55] K.C. Wallnau and J. Ivers. Snapshot of CCL: A Language for Predictable Assembly. Technical Report DRAFT–CMU/SEI-2003-TN-025, CMU SEI, 2003.

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