A Taxonomy of Software Component Models

Kung-Kiu Lau and Zheng Wang
School of Computer Science, the University of Manchester, Manchester M13 9PL, UK
{kung-kiu,zw}@cs.man.ac.uk

Abstract

CBSE currently lacks a universally accepted terminology. Existing component models adopt different component definitions and composition operators. We believe that for future research it would be crucial to clarify and unify the CBSE terminology, and that the starting point for this endeavour should be a study of current component models. In this paper, we take this first step and present and discuss a taxonomy of these models. The purpose of this taxonomy is to identify the similarities and differences between them with respect to commonly accepted criteria, with a view to clarification and/or potential unification.

1. Introduction

One problem that CBSE currently faces is the lack of a universally accepted terminology. Even the most basic entity, a software component, is defined in many different ways [7]. For example, a widely adopted definition due to Szyperski [34] is the following:

> "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"

Another definition, due to Meyer [23], is:

> "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."

Plainly, these two definitions are not identical.

Another problem with most definitions (including the two above) of software components is that they are not based on a component model. A notable exception is the following definition by Heineman and Council [18]:

> "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."

The importance of the component model, as illustrated by its role in this definition, is that it provides the underlying semantic framework for not only defining what components are, but also 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.

Component models, however, pose another problem for CBSE, in that the standard component models like EJB [14], COM [6] and CCM [27] adopt slightly different component definitions from widely adopted ones like Szyperski's, and from each other.

For future research it would therefore be crucial to clarify and hopefully unify the CBSE terminology and its definition. We believe that the starting point for this endeavour should be a study of current component models. In this paper, we take this first step and present and discuss a taxonomy of these models. The purpose of this taxonomy is to identify the similarities and differences between them with respect to commonly accepted criteria, with a view to clarification and/or potential unification.

2. Software Component Models

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

A *software component model* 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.

In this paper we will consider mainly the following software component models: JavaBeans [33], EJB, COM, CCM, Koala [35], SOFA [29], KobrA [5], ADLs [11], UML2.0 [26], PECOS [24], Pin [19] and Fractal [8].

2.1. The Syntax of Software Components

The syntax of components defines what they are physically, i.e. the rules for constructing and representing them. Ideally the syntax should be embodied in a language that can be used for defining and constructing components.

In current component models, the language for components tends to be a programming language. For example in both JavaBeans and EJB, components are defined as Java classes.

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

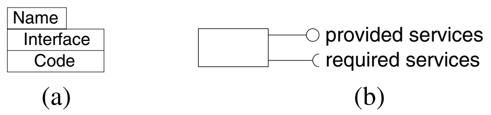


Figure 1. A software component.

the services 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)). 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.

In current component models, components tend to be objects in the sense of object-oriented programming. The methods of these objects are their provided services, and the methods they call in other objects are 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. Java beans interact with one another via adaptor classes generated by the container. Adaptor 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 [31], 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.

2.3. The Composition of Software Components

In CBSE, 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. [22]. 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, e.g. JavaBeans, EJB, COM and CCM, there is no composition language. ADLs of course have composition languages [4].

In order to reason about composition, we need a composition theory. Such a theory allows us to calculate and thus predict the result of applying a composition operator to components (see [13] for a discussion). Current component models tend not to have composition theories, even those with a composition language.

Composition can take place during different stages of the *life cycle* of components [10]. We identify two main stages in this cycle, the *design* phase and the *deployment* phase, where composition takes place:

- *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.

Composition in the Design Phase

In the Design Phase, components can be composed into *composite* components. Composite components are also *stateless*. Composite components form sub-parts of a system, and as such are useful for designing a 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.

For example, in JavaBeans, the container provides a repository for beans, e.g. ToolBox in the Beans Development Kit (BDK) [32, 30], which are stored as JAR files. However, no composition is possible in the design phase, and therefore no composite beans can be formed.

Composition in the Deployment Phase

In the Deployment Phase, component instances can be composed into a complete system that is executable. The system and its constituent component instances are the end result of system design (design phase) and implementation (deployment phase).

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

3. Categories of Component Models

Clearly, we can classify existing software component models according to component syntax, semantics or composition. In this section, we present these three categories.

3.1. 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 2).

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 2. Categories based on syntax.

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

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 [6], CCM uses the OMG IDL [27], 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 a language called CoCo (Component Composition Language) [17], whilst in Pin, components are defined in a language called CCL (Construction and Composition Language) [38]. 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.

3.2. 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) those in which components are architectural units (Figure 3).

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

Figure 3. Categories based on semantics.

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 [25]. 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).

3.3. Categories based on 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. It is depicted in Figure 4. We believe this view of the ideal component life cycle is the commonly accepted one in CBSE, and not just our own subjective view: all existing component models reflect this ideal life cycle to varying degrees, as we will show later.

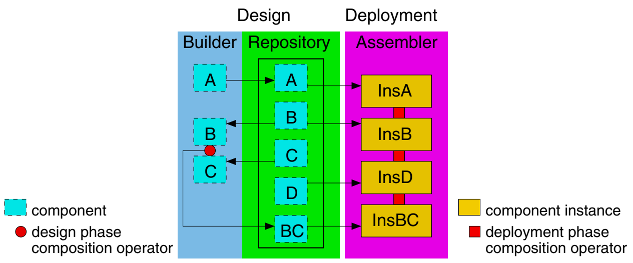


Figure 4. The idealised component life cycle.

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 4; (ii) retrieve components from the repository, compose them and deposit them back in the repository, e.g. in Figure 4, 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 4, 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 some do not. Thus many categories are possible. Figure 5 gives four categories that cover all major existing component models.

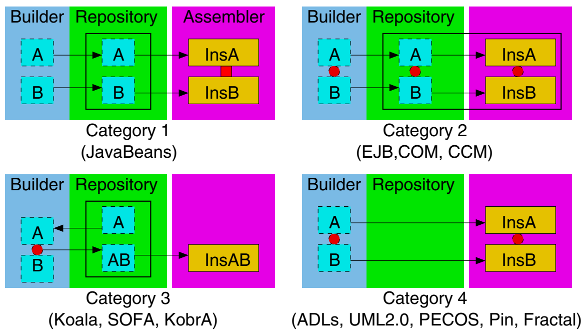


Figure 5. Categories based on composition.

In Category 1, in the design phase, there is a repository but there is no composition: components 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 (see Figure 6). In the design phase, Java beans can be constructed 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. 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.

Figure 6 shows a simple JavaBeans example. MessageBeanA and MessageBeanB display the messages "Hello, I'm Bean A" and "Hello, I'm Bean B" respectively when notified of the 'mousePressed' event by another bean.

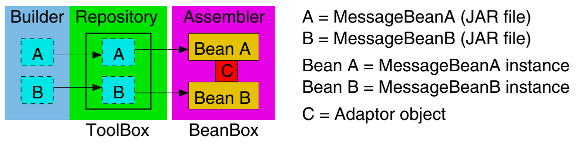

Builder
Repository
Assembler
A
B
A
B
Bean A
C
Bean B
ToolBox
BeanBox
A = MessageBeanA (JAR file)
B = MessageBeanB (JAR file)
Bean A = MessageBeanA instance
Bean B = MessageBeanB instance
C = Adaptor object

Figure 6. Category 1: JavaBeans example.

In the design phase, JAR files for MessageBeanA and MessageBeanB are constructed, containing for each bean the bean implementation class, the event state object and the event listener interface. These files are deposited into the ToolBox of the BDK. No bean composition is possible at this stage.

In the deployment phase, instances Bean A and Bean B of MessageBeanA and MessageBeanB respectively are created by dragging the beans from the ToolBox and dropping them into the BeanBox. In the BeanBox, Beans A and B can be composed via event handling: a source event in one bean can be linked to a target method in the other. For instance, Beans A and B can be linked in such a way that when the mouse is pressed on Bean B, the message "Hello, I'm Bean A" will be displayed by Bean A. The composition mechanism provided by the BeanBox is an (automatically generated and compiled) adaptor object that handles events and passes method calls between the beans.

In Category 2 (Figure 5), 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.

For example, in EJB, enterprise beans can be constructed in a Java programming environment such as Eclipse [15]. The repository for enterprise beans is an EJB container hosted and managed by a J2EE server. In the design phase, enterprise beans are composed by method and event delegation. In the deployment phase, no new composition is possible (see e.g. [9]), and so there is no assembler. The EJB container provides the run-time environment for the bean instances.

Figure 7 shows an EJB example. A book store 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. Suppose this

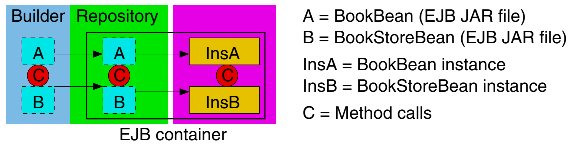

Builder
Repository
A
C
B
A
C
B
InsA
C
InsB
EJB container
A = BookBean (EJB JAR file)
B = BookStoreBean (EJB JAR file)
InsA = BookBean instance
InsB = BookStoreBean instance
C = Method calls

Figure 7. Category 2: EJB example.

book store system can be implemented using an entity bean BookBean that represents a table of books in a database, and a session bean BookStoreBean that adds details of a set of books into the table of books.

In the design phase, the EJB JAR files of BookBean and BookStoreBean are constructed. For each enterprise bean, the JAR file contains the enterprise bean implementation class, the home and remote interfaces, and the deployment descriptor. The EJB JAR files for BookBean and BookStore Bean are deposited into an EJB container on a J2EE server. The two beans are composed at this stage by method calls.

In the deployment phase, BookBean and BookStoreBean are looked up by an application client program, and instances of them are created in the EJB container. These instances are composed within the EJB container in same manner as the composition of BookBean and BookStoreBean in the container when the whole system was designed.

In COM, components are constructed in a programming environment such as Microsoft Visual Studio .NET [37]. The repository of COM components is the COM server, and composition is by method calls through interface pointers in the design phase. In the deployment phase, there is no assembler, and the COM server provides the run-time environment.

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 [25]. The repository of CORBA components is a CCM container hosted and managed by an application server, and composition is also by direct method calls in the design phase. In the deployment phase, the CCM container provides the run-time environment.

Category 3 (Figure 5) is the same as Category 2 except that in Category 3 the builder can retrieve (composite) components from the repository, e.g. in Figure 5, A is retrieved from the repository; and the repository can store composite components, e.g. in Figure 5, 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.

For example, 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. 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.

Figure 8 shows a Koala example of a Stopwatch device

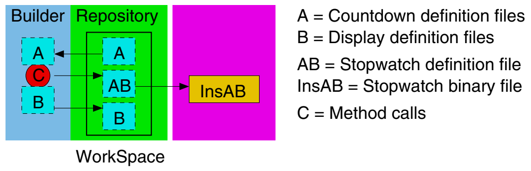

Builder
Repository
A
C
B
A
AB
B
InsAB
WorkSpace
A = Countdown definition files
B = Display definition files
AB = Stopwatch definition file
InsAB = Stopwatch binary file
C = Method calls

Figure 8. Category 3: Koala example.

being implemented by constructing a new Countdown component and composing it with a Display component from the repository.

In the design phase, the definition files for the Display component are retrieved from the repository. (Definition files contain the definitions of interfaces, components and data.) Then the definition files for the Countdown component are constructed. Using their definition files, Countdown and Display are composed by method calls. This yields a definition file for Stopwatch. The definition files for Countdown and Stopwatch are deposited into the KoalaModel WorkSpace.

In the deployment phase, the definition files of Stopwatch, Countdown and Display are compiled by the Koala compiler to C header files. Then the programmer has to write C files and compile these with the header files to binary C code for Stopwatch.

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

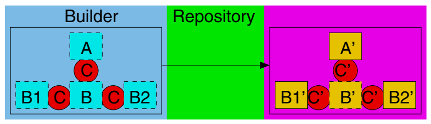
In KobrA, components can be constructed in a visual builder tool such as Visual UML [36]. The repository of

KobrA components is a file system that stores a set of UML diagrams, and KobrA components composition is by direct method calls in the design phase.

In Category 4 (Figure 5), 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.

In ADLs, components and connectors are constructed in the design phase, possibly in a visual builder tool, e.g. AcmeStudio [1]. 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.

Figure 9 shows an example in the Acme ADL [16]. 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.



A = ATM component	A' = Implementation of A
B = BankConsortium component	B' = Implementation of B
B1 = Bank component 1	B1' = Implementation of B1
B2 = Bank component 2	B2' = Implementaion of B2
C = Connector	C' = Connector implementation

Figure 9. Category 4: ADL example.

In the design phase, the architecture for the whole system is designed. This is done by using components and connectors: components are units of computation (and storage), whereas connectors define the interactions between the units. So, in this example, specifications of ATM, BankConsortium, Bank1 and Bank2 are constructed within a builder tool. The specifications contain ADL definitions for each component and connector in the system.

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 [2] that can be compiled to instances of the components [3]. Then 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 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: delegation connectors and assembly connectors. Like some ADLs, UML2.0 only specifies components and connectors, but does not provide support for their implementation in the deployment phase.

In PECOS, components are constructed in a programming environment such as Eclipse. In the design phase, components are composed by linking their ports with connectors. Components and connectors are specified in CoCo. 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++.

In Pin, components and connectors are specified in CCL and their implementations are usually generated by the CCL processor. In the design phase, components are composed by connectors that link source pins of one component to the sink pins of another. In the deployment phase, components are executed in the Pin run-time environment.

In Fractal, components are constructed in a programming environment with Fractal APIs. Fractal components are composed by method calls through connectors. In the deployment phase, the Java Virtual Machine serves as the run-time environment for Fractal components.

4. A Taxonomy of 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 2) and those based on semantics (Figure 3), 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 5), 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 CBSE 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 10, based on component composition in the ideal component life cycle, as discussed in the last section.

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 10. A taxonomy based on composition.

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.

5. Discussion

The basis for the taxonomy in Figure 10 is the ideal component life cycle, discussed in Section 3.3. This can be justified by the commonly accepted desiderata of CBSE: (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 [12]. 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.

6. Conclusion

We have presented a taxonomy that uses a unifying terminology for software component models. We believe this is an important first step for this endeavour. We have deliberately avoided adopting terminology from any one 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 models because the latter do not follow a unified terminology.

The taxonomy also reveals clearly the strengths and weaknesses of existing models. In addition to what we discussed in the previous section, no existing component model supports predictable assembly [28], which is the cornerstone of CBSE. To address this, new component models have to be developed. The on-going model Pin is one such, and we ourselves are working on another [20].

Finally, details of the examples in Section 3.3 of composition in various component models can be found in [21].

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] AcmeStudio 2.1 User Manual. `http://www-2.cs.cmu.edu/~acme/Manual/AcmeStudio-2.1.htm`

[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 *et al.* 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 *et al.* *Component-based Product Line Engineering with UML*. Addison-Wesley, 2001.

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

[7] M. Broy *et al.* What characterizes a software component? *Software – Concepts and Tools*, 19(1):49–56, 1998.

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

[9] 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.

[10] 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.

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

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

[13] I. Crnkovic *et al.* 6th ICSE workshop on CBSE: automated reasoning and prediction. *ACM SIGSOFT Software Engineering Notes*, 29(3):1–7, May 2004.

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

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

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

[17] T. Genssler *et al.* *PECOS in a Nutshell*. `http://www.pecos-project.org/`, September 2002.

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

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

[20] K.-K. Lau, P. Velasco Elizondo, and Z. Wang. Exogenous connectors for software components. In *Proc. 8th Int. SIGSOFT Symp. on CBSE*, pages 90–106. Springer-Verlag, 2005.

[21] K.-K. Lau and Z. Wang. A survey of software component models. Technical Report CSPP-30, School of Computer Science, the University of Manchester, 2005.

[22] 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.

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

[24] O. Nierstrasz *et al.* A component model for field devices. In *Proc. 1st Int. IFIP/ACM Working Conference on Component Deployment*, pages 200–209. ACM Press, 2002.

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

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

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

[28] Predictable Assembly from Certifiable Components. Software Engineering Institute, Carnegie-Mellon University. `http://www.sei.cmu.edu/pacc/`.

[29] 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.

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

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

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

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

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

[35] R. van Ommering *et al.* The Koala component model for consumer electronics software. *IEEE Computer*, pages 78–85, March 2000.

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

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

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