Software Component Models:
Past, Present and Future

Kung-Kiu Lau, Zheng Wang,
Simone Di Cola, Cuong Tran, Vasilis Christou

School of Computer Science
The University of Manchester
United Kingdom

kung-kiu@cs.man.ac.uk

Tutorial, CompArch 2014, 30 June 2014, Lille, France
# Schedule

<table>
<thead>
<tr>
<th>Part I</th>
<th>09:00–09:45</th>
<th>Introduction</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>Traditional CBSE desiderata</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Idealised component and system life cycles</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Overview of current component models</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Current life cycles</td>
</tr>
<tr>
<td>Part II</td>
<td>09:45–10:30</td>
<td>Taxonomy: overview (5 categories)</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Taxonomy: categories 1,2</td>
</tr>
<tr>
<td><strong>Break</strong></td>
<td>10:30–11:00</td>
<td><strong>Coffee</strong></td>
</tr>
<tr>
<td>Part III</td>
<td>11:00–11:45</td>
<td>Taxonomy: categories 3,4,5</td>
</tr>
<tr>
<td>Part IV</td>
<td>11:45–12:30</td>
<td>Future challenges and new CBSE desiderata</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Future component models</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Future life cycles</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Conclusion</td>
</tr>
</tbody>
</table>

**Disclaimer:** *In this tutorial, we only provide overviews of component models, not user manuals for them!*

We accept responsibility for any factual errors or inaccuracies, and we welcome your feedback.
Part I

- Introduction
- Traditional CBSE desiderata
- Idealised component and system life cycles
- Overview of current component models
- Current life cycles
Initially, CBSE research focused on:
- identifying desiderata [18]
- developing different approaches

Later, the notion of component models [37, 47, 48, 29] was introduced:
- a common framework for defining and analysing CBSE approaches wrt CBSE desiderata
- every CBSE approach is underpinned by a component model

Studies of component models [47, 48]:
- yield taxonomy of component models based on CBSE desiderata
- show early approaches/models do not fully meet the CBSE desiderata
A software component model defines:

- what components are:
  - syntax of components
  - semantics of components

- how to compose components:
  - syntax of composition operators
  - semantics of composition

Introduction

‘Standard’ Component Definitions

Szyperski [62]
“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.”

Meyer [50]
“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.”

Heineman and Councill [37]
“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.”

<table>
<thead>
<tr>
<th>Component Definition</th>
<th>Based on Component Model?</th>
<th>Defines Component Model?</th>
</tr>
</thead>
<tbody>
<tr>
<td>Szyperski</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>Meyer</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>Heineman &amp; Councill</td>
<td>Yes</td>
<td>No</td>
</tr>
</tbody>
</table>
## Models versus Frameworks

### Component Models versus Component(-oriented Programming) Frameworks

#### Component Frameworks

- provide **programming environments**
- objected-oriented examples: COM, .NET, OSGi, EJB, Fractal (?)

#### Component Models

- provide **semantics**: components and their composition

- A component framework contains a component model
- COM, .NET, OSGi, EJB, Fractal all contain a model with objects as components and method call as composition
### Present

- Taxonomy of component models shows:
  - Current component models also do not fully meet the CBSE desiderata
- New component models proposed
- Taxonomy expanded

### Future

- CBSE faces new challenges:
  - increased **scale**
  - increased **complexity**
  - increased **safety**
- Future component models have to meet **new desiderata**
Traditional CBSE Desiderata

- Components should *pre-exist*
- Components should be *produced independently*
- Component should be *deployed independently*
- It should be possible to *copy* and *instantiate* components
- It should be possible to *build* composites
- It should be possible to *store* composites

Idealised Component Life Cycle
Composition in Component Design Phase and Component Deployment Phase

CBSE Desiderata

<table>
<thead>
<tr>
<th>Desideratum</th>
<th>Design Phase</th>
<th>Deployment Phase</th>
</tr>
</thead>
<tbody>
<tr>
<td>Components should pre-exist</td>
<td>Deposit components in repository</td>
<td>Retrieve components from repository</td>
</tr>
<tr>
<td>Components should be produced independently</td>
<td>Use builder</td>
<td></td>
</tr>
<tr>
<td>Components should be deployed independently</td>
<td>Use assembler</td>
<td></td>
</tr>
<tr>
<td>It should be possible to copy and instantiate components</td>
<td>Copies possible</td>
<td>Copies and instances possible</td>
</tr>
<tr>
<td>It should be possible to build composites</td>
<td>Composition possible</td>
<td>Composition possible</td>
</tr>
<tr>
<td>It should be possible to store composites</td>
<td>Use repository</td>
<td></td>
</tr>
</tbody>
</table>


Idealised Component and System Life Cycles

- Idealised component life cycle entails an idealised system life cycle
- Component life cycle should be separate from system life cycle

Current Component Models

Components

A Generic Component

Required Service ——— Provided Service

An Object

Provided method

An Architectural Unit

in1 in2 out1 out2

An Encapsulated Component

Components

<table>
<thead>
<tr>
<th>Components</th>
<th>Provided services</th>
<th>Required services</th>
<th>Composition mechanism</th>
</tr>
</thead>
<tbody>
<tr>
<td>Objects</td>
<td>Methods</td>
<td>———</td>
<td>Method call</td>
</tr>
<tr>
<td>Architectural units</td>
<td>Out-ports</td>
<td>In-ports</td>
<td>Port connection</td>
</tr>
<tr>
<td>Encapsulated components</td>
<td>Methods</td>
<td>None</td>
<td>Exogenous composition</td>
</tr>
</tbody>
</table>
Current Component Models

Composition Mechanisms

Connection: Method Call & Port Connection

(a) Direct message passing
(b) Indirect message passing

Coordination: Exogenous Composition

Coordinator

U1 U2

communication channel
Current Component and System Life Cycles

Waterfall-like component and system life cycles [26, 41, 60, 24, 28, 40]

The V model [63] for modular system development adapted for CBSE (e.g. [31, 34])
## Current Component Models

Support for Idealised Component and System Life Cycles

<table>
<thead>
<tr>
<th>Category</th>
<th>Component Models</th>
<th>Design</th>
<th>Deploy</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>Deposit-N</td>
<td>Retrieve</td>
</tr>
<tr>
<td>Design without Repository</td>
<td>Acme-like ADLs UML2.0, PECOS</td>
<td>×</td>
<td>×</td>
</tr>
<tr>
<td>Design with Deposit-only Repository</td>
<td>EJB, OSGi, Fractal COM, .NET, CCM</td>
<td>✓</td>
<td>×</td>
</tr>
<tr>
<td>Deployment with Repository</td>
<td>JavaBeans, Web Services</td>
<td>✓</td>
<td>×</td>
</tr>
<tr>
<td>Design with Repository</td>
<td>Koala, SOFA, Kobra SCA, Palladio, ProCom</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Design &amp; Deployment with Repository</td>
<td>X-MAN</td>
<td>✓</td>
<td>✓</td>
</tr>
</tbody>
</table>

Deposit-N=Deposit components constructed from scratch  
Deposit-C=Deposit composite components constructed from existing components
Taxonomy of Component Models

Category 1: Design without Repository
(Acme–like ADLs, UML2.0, PECOS)

Category 2: Design with Deposit–only Repository
(EJB, OSGi, Fractal, COM, .NET, CCM)

Category 3: Deployment with Repository
(JavaBeans, Web Services)

Category 4: Design with Repository
(Koala, SOFA, KobrA, SCA, Palladio, ProCom)

Category 5: Design and Deploy with Repository
(X-MAN)

Lau et al. (University of Manchester)
Part II

- Taxonomy of component models: Overview (5 categories)
- Taxonomy of component models: Categories 1 and 2
## Taxonomy of Component Models

### Overview

<table>
<thead>
<tr>
<th>Category</th>
<th>Component Models</th>
<th>Design</th>
<th>Deploy</th>
</tr>
</thead>
<tbody>
<tr>
<td>Design without Repository</td>
<td>Acme-like ADLs UML2.0, PECOS</td>
<td>✗</td>
<td>✗</td>
</tr>
<tr>
<td>Design with Deposit-only Repository</td>
<td>EJB, OSGi, Fractal COM, .NET, CCM</td>
<td>✓</td>
<td>✗</td>
</tr>
<tr>
<td>Deployment with Repository</td>
<td>JavaBeans, Web Services</td>
<td>✓</td>
<td>✗</td>
</tr>
<tr>
<td>Design with Repository</td>
<td>Koala, SOFA, Kobra SCA, Palladio, ProCom</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Design &amp; Deployment with Repository</td>
<td>X-MAN</td>
<td>✓</td>
<td>✓</td>
</tr>
</tbody>
</table>

Deposit-N=Deposit components constructed from scratch
Deposit-C=Deposit composite components constructed from existing components
Category 1: Design without Repository
(Acme-like ADLs, UML2.0, PECOS)
Acme-like ADLs

Acme

Acme [33] is a prototype Architecture Description Language (ADL).

It typifies first-generation ADLs, e.g. Darwin [1], UniCon [3], Wright [4], ArchJava [7, 8].

Acme-like ADLs: Components

In Acme-like ADLs, a component is an architectural unit that represents a primary computational element and data store of a system.

- **Interfaces** are defined by a set of **ports**

- Each **port** identifies a point of interaction between the **component** and its environment (including other components)

- A component may have **multiple interfaces** by using different types of ports
In Acme-like ADLs, components are composed by connectors.

Connectors connect components via their ports.
In ACME-like ADLs, the components and the system are designed together in an ADL tool.

- The builder is the ADL tool if any
- There is no repository
- There is no assembler

**Diagram:***

```
A  = component A  A'  = implementation of A
B  = component B  B'  = implementation of B
B1 = component B1 B1' = implementation of B1
B2 = component B2 B2' = implementation of B2
c  = connector   c'  = implementation of c
```
**Acme-like ADLs**

**Component and System Life Cycles**

- **Component life cycle** coincides with **system life cycle**
- During **component/system design phase**, components are
  - identified and defined
  - composed by connectors into a system design
- The **design** for both components and the system has to be implemented (somehow) in a chosen programming language.
- At **run-time**, the implemented system is executed in the run-time environment of that programming language.
Consider a simple bank system consisting of an **ATM** component, a **BankConsortium** component, and 2 **Bank** components **Bank1** and **Bank2**.

```
Component ATM = {
  Port send;
}

Component BankConsortium = {
  Port receive;
  Port send;
}

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

Component Bank2 = {
  Port receive;
  Property bankid : String = "Bank2";
}
```
In design phase, the architecture for the whole system is designed using the above components and the following connectors:

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

Connector BankContoB1 = {
    Role request;
    Role produce;
};

Connector BankContoB2 = {
    Role request;
    Role produce;
};
```
System BankSys = {
    Component ATM = {
        Port send;
    };

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

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

    Connector BankContoB1 = {
        Role request;
        Role produce;
    };

    Attachments {
        ATM.send to ATMtoBankCon.request;
        ATMtoBankCon.produce to BankConsortium.receive;
        BankConsortium.send to BankContoB1.request;
        BankContoB1.produce to Bank1.receive;
        BankConsortium.send to BankContoB2.request;
        BankContoB2.produce to Bank2.receive;
    }
};

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

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

Component BankConsortium = {
    Role request;
    Role produce;
};

Connector BankContoB2 = {
    Role request;
    Role produce;
};
In UML2.0 [53], 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 required and provided interfaces (ports);
- Services of components are encapsulated through their required and provided interfaces.
UML 2.0: Composition

UML 2.0 components are composed by UML connectors:
- delegation connectors
- assembly connectors

Composites are assembled by assembly connectors
Systems are assembled by delegation and assembly connectors
In UML2.0, the components and the system are designed together in a visual builder tool such as Visual UML.

- The **visual builder tool** is the builder
- There is **no repository**
- There is **no assembler**

![Diagram showing UML2.0 support for idealised component and system life cycles](image)

- Builder
- RTE
- Implementation Language RTE
- Visual Builder Tool
- InsA = UMLA instance
- InsB = UMLB instance
- \( A = \text{UMLA} \)
- \( B = \text{UMLB} \)
- \( \text{InsA} = \text{UMLA instance} \)
- \( \text{InsB} = \text{UMLB instance} \)
- \( \bullet = \text{connector} \)
Component life cycle coincides with system life cycle.

During component/system design phase, components are:
- identified and defined
- composed by connectors into a system design

The design for both components and the system has to be implemented (somehow) in a chosen programming language.

At run-time, the implemented system is executed in the run-time environment of that programming language.
Consider a simple bank system that is implemented by ATM, BankConsortium, Bank1 and Bank2 components.

- **ATM**
  - Provided interfaces:
    - GetCardNo
  - Required interfaces:
    - CheckBankID

- **BankConsortium**
  - Provided interfaces:
    - CheckBankID
    - GetCardNo
  - Required interfaces:
    - Withdraw
    - Deposit
    - CheckBalance

- **Bank1**
  - Provided interfaces:
    - Withdraw
    - Deposit
    - CheckBalance

- **Bank2**
  - Provided interfaces:
    - Withdraw
    - Deposit
    - CheckBalance
In design phase, the architecture for the whole system is designed.
In PECOS\textsuperscript{1} [35], a component is a unit of design which has a specification and an implementation.

- Every component has a name, a number of property bundles, a set of ports, and behaviour.
- Ports are interfaces of components.

PECOS components are specified in the CoCo (Component Composition) language.

\textsuperscript{1}PErvasive COmponent Systems
In PECOS, components are composed by **connectors**.

**Connectors** connect components via their **ports**.

---

**Software Component Models**

Lau et al. (University of Manchester)
In PECOS, the components and the system are designed and constructed together in a programming environment such as **Eclipse**.

- The **programming environment** is the builder
- There is **no repository**
- There is **no assembler**
Component life cycle coincides with system life cycle
During component/system design phase, components are
▶ identified and defined
▶ composed by connectors into a system design in the CoCo (Component Composition) language
The design has to be implemented in a chosen programming language, usually Java or C++.
At run-time, the implemented system is executed in the run-time environment of Java or C++.
PECOS: Example

Consider a device that is assembled from Clock, Display, EventLoop and DigitalDisplay components.

```plaintext
class Clock {
  output long msecs;
}
class Display {
  input long time;
}
class EventLoop {
  output bool started;
  input long time_in_msecs;
}
class DigitalDisplay {
  input bool can_draw;
}
```
In the design phase, the architecture for the device is designed:

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);
}
Category 2: Design with Deposit-only Repository
(EJB, OSGi, Fractal, COM, .NET, CCM)
In **EJB** [30, 51] a component is an **enterprise Java bean** with a **Java interface**:

- **an enterprise Java bean** is a **Java class** in an **EJB container** on a **J2EE server**
- **an EJB container** uses the **interface** to manage and execute the **Java class** and its instances.
For an EJB:

- its **Java class** defines the **methods** of the bean
- its **interface** exposes the capabilities of the bean and provides all the methods needed for (remote) client applications to access the bean (over a network)

There are 3 kinds of EJBs:

- **Entity beans**
  Entity beans model business **data**; they are Java objects that cache **database information**.

- **Session beans**
  Session beans model business **processes**; they are Java objects that act as **agents performing tasks**.

- **Message-driven beans**
  Message-driven beans model **message-related** business **processes**; they are Java objects that act as **message listeners**.
Enterprise beans are composed (in the EJB container) by method and event delegation.
EJBs are constructed and composed in a J2EE-compliant IDE, and deposited and executed in an EJB container.

- A J2EE-compliant IDE (e.g. **NetBeans**) is the **builder** for EJB (composition of beans)
- An **EJB container** is the **repository**
- There is **no assembler**

![Diagram showing EJB life cycle](image-url)
In EJB, **components** are EJBs, and a **system** is the composition of EJBs in the EJB container (with a **remote interface**)

**Component life cycle** coincides with **system life cycle**

In **component/system design phase**, enterprise beans
- are designed, implemented and **composed** into a complete **system**
- and **deposited** in the EJB container

**Client applications** make calls to enterprise beans in the system via the system’s remote interface

At **run-time**, client applications are executed, invoking enterprise beans in the system.
Consider a bank which wishes to provide basic services (check balance, withdrawal and deposit) on its customer accounts.

The table of accounts in the database can be represented as an entity bean Account that consists of a Java class and a helper class.

- The Account Java class is defined with methods to access and change account details.
- Each instance of Account represents a row of the table of accounts in the database.
- AccountFacade is the helper class that behaves like the (EJB2) home interface of the Account bean.
EJB: Example (cont’d)

```java
@Entity  
@Table(name = "ACCOUNT")  
@XmlRootElement  
@NamedQueries(
    @NamedQuery(name = "Account.findAll", query = "SELECT a FROM Account a"),
    @NamedQuery(name = "Account.findByAccno", query = "SELECT a FROM Account a WHERE a.accno = :accno"),
    @NamedQuery(name = "Account.findByBalance", query = "SELECT a FROM Account a WHERE a.balance = :balance"))
public class Account implements Serializable {
    private static final long serialVersionUID = 1L;

    @Id  
    @Basic(optional = false)  
    @NotNull  
    @Size(min = 1, max = 4)  
    @Column(name = "ACCNO")  
    private String accno;

    @Basic(optional = false)  
    @NotNull  
    @Column(name = "BALANCE")  
    private int balance;

    public Account() { }

    public Account(String accno) {
        this.accno = accno;
    }

    public Account(String accno, int balance) {
        this.accno = accno;
        this.balance = balance;
    }

    public String getAccno() {
        return accno;
    }

    public void setAccno(String accno) {
        this.accno = accno;
    }

    public int getBalance() {
        return balance;
    }

    public void setBalance(int balance) {
        this.balance = balance;
    }

    ...
```

Lau et al. (University of Manchester)  Software Component Models  CompArch 2014  46 / 177
To construct the system we also need a session bean Bank that consists of a Java class and interface:

- **Bank** is the Java class that defines the business methods (services on accounts)
- **BankRemote** is the remote interface
@Stateless
public class Bank implements BankRemote {
    @EJB
    private AccountFacade accountFacade;

    @Override
    public Integer balance(final String accno) throws Exception {
        Account acc = accountFacade.find(accno);
        if (acc != null) return acc.getBalance();
        else throw new Exception("Account not found.");
    }

    @Override
    public void deposit(final String accno, final Integer amount) throws Exception {
        if (amount <= 0) throw new Exception("Invalid amount.");
        Account acc = accountFacade.find(accno);
        if (acc != null) acc.setBalance(acc.getBalance() + amount);
        else throw new Exception("Account not found.");
    }

    // Other methods...
}
The system is assembled from the **Account** entity bean and the **Bank** session bean:
OSGi

A component framework that brings modularity to JAVA platform

http://www.osgi.org/Technology/WhatIsOSGi

Lau et al (University of Manchester)
OSGi: Bundles

OSGi consists of bundles:

- **Bundle-ManifestVersion:** 2
- **Bundle-Name:** Greeting API
- **Bundle-SymbolicName:** org.foo.hello
- **Bundle-Version:** 1.0
- **Bundle-Activator:** org.foo.HelloWorld
- **Export-Package:** org.foo.hello;version="1.0"
- **Import-Package:** org.foo.hello;version="[1.0,2.0)"

Life-Cycle:

- INSTALLED
- RESOLVED
- ACTIVE
- STOPPING
- STARTING
- UNINSTALLED
OSGi Component Model
Components and Composition

- OSGi bundles do not compose, but POJOs within them do via direct method invocation.
- So components in OSGi component model are Java objects; and composition is by direct method call.

![Diagram showing service registry and interactions between Bundle A and Bundle B]
POJOs in OSGi bundles are constructed in any editor, e.g. Eclipse. They are composed inside a bundle to provide a service (exposed by the bundle)

(POJOs inside) Bundles are installed in an OSGi-compliant framework, e.g. Equinox, which is therefore the repository

There is no assembler

A = POJOA
B = POJOB
InsA = POJOA instance
InsB = POJOB instance
= method call
In OSGi component models, components are POJOs, and a system is the service provided by their composition (with an interface published by the bundle).

Component life cycle coincides with system life cycle.

In component/system design phase, POJOs
  ▶ are designed, implemented and composed into a system
  ▶ and deposited in the an OSGi-compliant framework, e.g. Equinox

Client applications make calls to POJOs inside bundles via the published service interface.

At run-time, client applications are executed, invoking POJO instances in the system.
**OSGi: Example - HelloWorld Producer**

```java
@Override
public void start(BundleContext bundleContext){
    registration = bundleContext.registerService(
        HelloWorldProvider.class.getName(),
        new HelloWorldProviderImpl(),null);
}
```

```java
@Override
public void start(BundleContext bundleContext){
    new HelloWorldConsumer((bundleContext.getServiceReference(HelloWorldProvider.class.getName())));
}
```

**Service Registry**

1. **HelloWorldProvider**
2. **Publish**
3. **Find**
4. **Interact**
5. **HelloWorldConsumer**

**Public Class**

```java
public class HelloWorldProviderImpl
    implements HelloWorldProvider{
    @Override
    public void hello(){
        System.out.println("Hello World!");
    }
}
```

```java
public class HelloWorldConsumer{
    public HelloWorldConsumer(HelloWorldProvider
    helloWorldProvider) {
        helloWorldProvider.hello();
    }
}
```
In *Fractal* [19, 20, 32], a component:

- is a unit of encapsulation and behaviour;
- consists of two parts:
  - **content**
    - a finite set of sub-components;
  - **membrane**
    - typically composed of several controllers, each in charge of a specific function;
    - supports interfaces to introspect and reconfigure its internal features;
    - maintains a causally connected representation of the component’s content.
Fractal: Components (cont’d)
Fractal: Composition

- Composition via **port bindings**
- A binding can be either:
  - **primitive**: if the bound interfaces are in the same address space (e.g. B-C in picture); or
  - **composite**: if the bound interfaces span different address spaces; it is embodied in a binding object which itself takes the form of a component (e.g. A-E in picture)
Fractal components are **constructed** in the Fractal for Eclipse (F4E) programming environment

- The **programming environment** is the **builder**
- The **programming environment** is the **repository**
- There is **no assembler**
- The **run-time environment** is the **JVM**

A = FractalA (JAR file)
B = FractalB (JAR file)
InsA = FractalA instance
InsB = FractalB instance

= method call
Component life cycle coincides with system life cycle

During component/system design phase, components in a chosen programming language (Java or C/C++) are
  - identified and defined
  - composed by port bindings into a system design using Fractal APIs

At run-time, the system is executed in the run-time environment of the chosen programming language (Java or C/C++).
Fractal: Example

```
<definition name="hw.HelloWorld">

  <interface name="r" role="server" signature="java.langRunnable"

  <component name="caller" definition="hw.CallerImpl" />
  <component name="callee" definition="hw.CallerImpl" />

  <binding client="this.r" server="caller.r" />
  <binding client="callee.s" server="callee.s" />

</definition>
```

In COM (Component Object Model) [17, 49, 54, 27], a component is a unit of compiled code on Windows Registry.

- **Services** in a component are **invoked via pointers** to the functions that implement them.
- For each service provided there is an **interface** (a COM component can implement **multiple interfaces**).
- **COM interfaces** are specified in Microsoft IDL.
- Every component must implement an **IUnknown** interface.
COM components are composed by method calls via interface pointers.

Diagram:

- Component1
  - IUnknown
  - Client Reference
- Component2
  - IUnknown

References:
- Lau et al. (University of Manchester)
- Software Component Models
- CompArch 2014
COM components are constructed in a programming environment such as Microsoft Visual Studio

- The **programming environment** is the **builder**
- The **Windows Registry** is the **repository**
- There is **no assembler**

![Diagram]

**Builder**
A
B

**Repository**
A
A
B
B

**RTE**
InsA
InsB

**Programming environment**
A = COMA
B = COMB

**Windows Registry**
InsA = COMA instance
InsB = COMB instance

**Windows OS**

**Legend**

- = method call
Component life cycle coincides with system life cycle:

- In component/system design phase, COM components are designed and implemented.
- assembled into a complete system.
- deposited in Windows Registry.

Client applications make calls to COM components in the system via interface pointers.

At run-time, client applications are executed, invoking COM components in the system.
Consider a spell checker system that comprises a checker component and a dictionary component.

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)]
oclass CSpellCheck {
    interface ISpellCheck;
}
};

A “library” is an interface glued with a coclass, e.g. the “library” of ISpellCheck and CSpellCheck makes the whole component...
import "unknwn.idl";

interface IUseCustomDictionary : IUnknown {
    HRESULT UseCustomDictionary([out, retval] vector <string>* dict);
}

library CustomdictionaryLib {
    coclass CCustomDictionary {
        interface IUseCustomDictionary;
    }
};

Dictionary component interface -- IUseCustomDictionary
In design phase, the spell checker system is assembled through method calls via interface pointers.

```c
#include <string.h>
CSpellCheckImpl :: CSpellCheckImpl() { }
CSpellCheckImpl :: ~CSpellCheckImpl() { }
STDMETHODIMP_(ULONG) CSpellCheckImpl :: AddRef(void) { ... }
STDMETHODIMP_(ULONG) CSpellCheckImpl :: Release(void) { ... }
STDMETHODIMP CSpellCheckImpl :: QueryInterface(...){ ... }
STDMETHODIMP CSpellCheckImpl :: CheckSpelling(...){ ... }
...

Checker component implementation

```
In Microsoft .NET [55, 66, 2], a component is an assembly that is a binary unit supported by Common Language Runtime (CLR).

- A .NET component is made up of metadata and code in Intermediate Language (IL).
- The metadata contains the description of assembly, types and attributes.
- The IL code can be executed in CLR.
.NET components are composed by method calls through references via metadata
.NET components are constructed in a **programming environment** such as **Microsoft Visual Studio .NET**

- The **programming environment** is the **builder**
- The **Microsoft Enterprise Library (MEL)** is the **repository**
- There is **no assembler**

![Diagram](image-url)
Component life cycle coincides with system life cycle

In component/system design phase, .NET components are
  ▶ designed and implemented
  ▶ assembled into a complete system
  ▶ deposited in a Windows server

Client applications make calls to .NET components in the system

At run-time, client applications are executed, invoking .NET components in the system.
Consider a banking system with an **ATM** component, which serves two instances **Bank1** and **Bank2** of a **Bank** component.

**ATM Component**

### Class
- Name: ATM;
- Visibility: Public;
- Type: Class

### Method
- Name: LocateBank;
- Visibility: Public;
- Virtual;
- Interop;
- IL;
- Managed;
- **Signature:**
  - void LocateBank(CardNo ACardNo, Password CusPass);
- **Invoke:** Bank.Deposit(...);

### Parameter
- **Name:** ACardNo;
- **Order:** 1;
- **Attributes:** In;

### Parameter
- **Name:** CusPass;
- **Order:** 2;
- **Attributes:** In;

**Bank Component**

### Class
- Name: Bank;
- Visibility: Public;
- Type: Class

### Method
- Name: Deposit;
- Visibility: Public;
- Virtual;
- Interop;
- IL;
- Managed;
- **Signature:**
  - void Deposit(CardNo ACardNo, Password CusPass);

### Parameter
- **Name:** ACardNo;
- **Order:** 1;
- **Attributes:** In;

### Parameter
- **Name:** CusPass;
- **Order:** 2;
- **Attributes:** In;

**IL Code**
The banking system is assembled from the ATM component and two instances of Bank component.
In CCM (CORBA Component Model) [14, 13, 6], a component is a CORBA meta-type hosted by a CCM container on a CCM platform such as OpenCCM.

- A CORBA meta-type is an extension and specialisation of a CORBA Object [52, 16]
- Component interfaces are made up of ports: Facets (provided services), Receptacles (required services), Event Sources and Event Sinks.
- Component types are specific, named collections of features that can be described in OMG IDL 3
- CCM components have homes that are component factories to manage a component instance life cycle
CCM components are assembled by method and event delegations in such a way that:

- facets match receptacles
- event sources match event sinks
CCM components are constructed in a programming environment such as Open Production Tool Chain and deposited into a CCM container hosted and managed by a CCM platform such as OpenCCM.

- The programming environment is the builder
- The CCM container is the repository
- There is no assembler

![Diagram](image_url)
Component life cycle coincides with system life cycle

In Component/system design phase, CCM components are
- designed and implemented
- composed into a complete system
- deposited in the CCM server

Client applications make calls to CCM components in the system via the system’s interface

At run-time, client applications are executed, invoking CCM components in the system.
Consider a simple bank system implemented by ATM, BankConsortium, Bank1 and Bank2 components (in OMG IDL 3):

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

```csharp
enum BankState {
    IsCustomer, NotCustomer
};

eventtype AccountInfo {
    public string cardno;
    public BankState customerinfo;
}
```

```csharp
component ATM {
    attribute string atmid;
    uses Bank getBankID;
    consumes AccountInfo customer;
}
```

```csharp
receptacle
event sink

manages instances
```

```csharp
home ATMhome manages ATM {
    factory new(in string atmid);
}
```
CCM: Example (cont’d)

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

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);
};
```
The bank system is assembled from the ATM, BankConsortium, Bank1 and Bank2 components.

The composition of CCM components is specified in a Component Assembly Descriptor (an XML file)
CCM: Example (cont’d)

```xml
<?xml version="1.0"?>
<DOCTYPE component assembly BANKSYSTEM "componentassembly.dtd">

<component assembly id = "banksys">
  <description> bank assembly descriptor</description>
  <components>
    <componentfile id = "ATM component">
      <filearchive name = "ATM.csd"/>
    </componentfile>
    <componentfile id = "BankConsortium component">
      <filearchive name = "BankConsortium.csd"/>
    </componentfile>
    <componentfile id = "Bank component">
      <filearchive name = "Bank.csd"/>
    </componentfile>
  </components>

  <partitioning>
    <homereplacement id = "ATMHome">
      <component refer id = "ATM Component"/>
      <component instantiation id = "atm">
        <register with naming name = "ATMHome"/>
      </homereplacement>

    <homereplacement id = "BankConsortiumHome">
      <component refer id = "BankConsortium Component"/>
      <component instantiation id = "bankconsortium">
        <register with naming name = "BankConsortiumHome"/>
      </homereplacement>

    <homereplacement id = "BankHome">
      <component refer id = "Bank Component"/>
      <component instantiation id = "bank1">
        <register with naming name = "BankHome"/>
      </homereplacement>
      <component instantiation id = "bank2">
        <register with naming name = "BankHome"/>
      </homereplacement>
    </partitioning>

  <connections>
  ...
  </connections>
</component assembly>
```
CCM: Example (cont’d)

<connections>
  <connectinterface>
    <usesport>
      <usesidentifier>getBankID</usesidentifier>
      <componentinstantiationref idref = "atm"/>
      <usesidentifier>deposit</usesidentifier>
      <usesidentifier>withdraw</usesidentifier>
      <usesidentifier>checkBalance</usesidentifier>
      <componentinstantiationref idref = "bankcon"/>
    </usesport>
  </connectinterface>

  <providesport>
    <providesidentifier>getBankID</providesidentifier>
    <componentinstantiationref idref = "bankcon"/>
    <providesidentifier>deposit</providesidentifier>
    <providesidentifier>withdraw</providesidentifier>
    <providesidentifier>checkBalance</providesidentifier>
    <componentinstantiationref idref = "bank"/>
  </providesport>

  <connectevent>
    <publishesport>
      <publishesidentifier>customer</publishesidentifier>
      <componentinstantiationref idref = "bankcon"/>
    </publishesport>

    <consumesport>
      <consumesidentifier>customer</consumesidentifier>
      <componentinstantiationref idref = "atm"/>
    </consumesport>
  </connectevent>
</connections>
Part III

- Taxonomy of component models: Categories 3,4 and 5
Category 3: Deployment with Repository (JavaBeans, Web Services)
JavaBeans: Components

In JavaBeans [61, 39], 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 bean builder tool like NetBeans.
In deployment phase, bean instances are composed via event delegation.

- A bean ‘composes’ with another bean by sending a message through delegation of events.
- The bean builder tool automatically generates, compiles, and loads event adaptor classes for logistics of events.
In NetBeans, individual beans are constructed as Java classes, and deposited in the Palette. Bean instances are retrieved from the Palette into the Design Form and composed into a system.

- NetBeans is the builder for Java beans
- the Palette of NetBeans is the repository (no composition)
- The Design Form of NetBeans is the assembler (composition of bean instances)
- JVM is the run-time environment
JavaBeans: NetBeans visual builder

Picture taken from [39].
Component life cycle is separate from system life cycle

In component design phase, beans are designed, implemented and deposited in the repository (e.g. NetBeans Palette)

In system design/component deployment phase, beans are retrieved from the repository and composed into a system in the assembler (e.g. NetBeans Design Form).

In system run-time, the system is executed in the assembler in JVM.
**JavaBeans: Example**

- **jButton1** has a method to generate an event (mouse press) when it is pressed.
- **jLabel1** has a method that outputs the message “You pressed the button.”
- The two beans are composed by an adaptor that when notified of an event (mouse press) calls **jLabel1**’s method, to produce the GUI shown.

Pictures taken from [39].
Web Services: Components

- **Web services** [9, 12, 5] are web application components that can be published, found, and used on the Web.

- A web service contains:
  - an interface in **WSDL** (Web Service Description Language)
    - describes the functionalities the web service provides
  - a **binary implementation** (the **service code**)

- **Service clients** communicate directly with **service providers** [12].
Web Services: Composition

- Web services are composed by method calls through SOAP (Simple Object Access Protocol) or JSON (JavaScript Object Notation) messages.
- SOAP uses XML tags while JSON uses name/value pairs [12].
**Orchestration**

1. Receive
2. Invoke
3. Invoke
4. Invoke
5. Reply

**Choreography**

1. Invoke
2. Invoke
3. Reply
4. Invoke
5. Invoke


Lau et al (University of Manchester)
Web services are constructed in a programming environment, e.g. Eclipse for Java, and deposited on a web server. Web services are composed (by orchestration) in a BPEL editor and the orchestration is executed on a BPEL engine.

- The programming environment is the builder
- The web server is the repository
- A BPEL editor is the assembler
- A BPEL engine is the run-time environment
Component life cycle is separate from system life cycle

In component design phase, services are

- designed and implemented
- deposited on a web server

In system design/component deployment phase, services are orchestrated in a BPEL editor

At run-time, the orchestration is executed on a BPEL engine
Web Services: Example
Composition by Orchestration

BPEL Process for Business Travels

Client

1: Request
6: Reply

portType

<<invoke (sync)>>
Retrieve employee travel status

Employee Travel Status Web Service

2: Request
3: Reply

<<invoke (async)>>
Get plane ticket offer from American Airlines

<<invoke (async)>>
Get plane ticket offer from Delta Airlines

[American.price<=Delta.price]

[American.price>Delta.price]

<<assign>>
Select American Airlines ticket

<<assign>>
Select Delta Airlines ticket

<<reply>>
Return the best offer

American Airlines Web Service

4.1: Invoke
4.2: Call-back

Delta Airlines Web Service

5.1: Invoke
5.2: Call-back

Picture from: http://www.oracle.com/technetwork/articles/matjaz-bpel1-090575.html

Lau et al. (University of Manchester)
Web Services: Example (cont’d)
Composition by Orchestration

BPEL Process

Employee Travel Status Web Service

American Airlines Web Service

Delta Airlines Web Service

Lau et al. (University of Manchester) Software Component Models CompArch 2014 98 / 177
Employee Travel Status Web Service

<message name="EmployeeTravelStatusRequestMessage">
  <part name="employee" type="tns:EmployeeType" />
</message>

<message name="EmployeeTravelStatusResponseMessage">
  <part name="travelClass" type="tns:TravelClassType" />
</message>

<portType name="EmployeeTravelStatusPT">
  <operation name="EmployeeTravelStatus">
    <input message="tns:EmployeeTravelStatusRequestMessage" />
    <output message="tns:EmployeeTravelStatusResponseMessage" />
  </operation>
</portType>
American Airlines and Delta Airlines Web Service

<Message name="FlightTicketRequestMessage">
  <Part name="flightData" type="tns:FlightRequestType" />
  <Part name="travelClass" type="emp:TravelClassType" />
</Message>

<Message name="TravelResponseMessage">
  <Part name="confirmationData" type="tns:FlightConfirmationType" />
</Message>

<PortType name="FlightAvailabilityPT">
  <Operation name="FlightAvailability">
    <Input message="tns:FlightTicketRequestMessage" />
  </Operation>
</PortType>

<PortType name="FlightCallbackPT">
  <Operation name="FlightTicketCallback">
    <Input message="tns:TravelResponseMessage" />
  </Operation>
</PortType>
BPEL Process for Business Travels

```xml
<message name="TravelRequestMessage">
  <part name="employee" type="emp:EmployeeType" />
  <part name="flightData" type="aln:FlightRequestType" />
</message>

<portType name="TravelApprovalPT">
  <operation name="TravelApproval">
    <input message="tns:TravelRequestMessage" />
  </operation>
</portType>

<portType name="ClientCallbackPT">
  <operation name="ClientCallback">
    <input message="aln:TravelResponseMessage" />
  </operation>
</portType>
```
Taxonomy of Component Models: Category 4

Category 4: Design with Repository
(Koala, SOFA, KobrA, SCA, Palladio, ProCom)
In Koala\(^2\) [65, 64], a component is an architectural unit which has a specification and an implementation.

- **Semantically**, components are units of computation and control (and data) connected together in an architecture.
- **Syntactically**, components are defined in an ADL-like language (Koala).

Components are definition files only (no implementation).
Koala: Composition

Koala components are composed by method calls through connectors.
In Koala, components (definition files) are constructed in the Koala programming environment and deposited in WorkSpace. They are retrieved from Workpace and composed into a system, also deposited in WorkSpace. The implementation of the component and system definition files (in C) is executed in the run-time environment of C.

- The **builder** is a Koala programming environment
- **KoalaModel Workspace** (a file system) provides the repository (composition of definition files)
- There is **no assembler**

![Diagram](image)

- **A** = Component A's definition files
- **B** = Component B's definition files
- **AB** = Component AB's definition file
- **InsAB** = Component AB's binary file
- `= method call`
Component life cycle is separate from system life cycle

In component design phase, Koala components are defined (in definition files) and deposited in the repository.

In system design/component deployment phase, Koala components are retrieved from the repository and composed into a system (a definition file), also deposited in the repository.

The definition files for the system and the components are compiled (by the Koala compiler) into C header files. C files are written to implement the components and the system, and compiled into binary C code.

At run-time, the binary code of the system is executed in the run-time environment of C.
Consider a **Stopwatch** device that comprises a **Countdown** component and a **Display** component.

- The **interfaces** are specified in **Koala IDL**
- The **component definitions** are in **Koala CDL**
In design phase, the Stopwatch device is constructed by composing a Countdown component (new) with a Display component (from the repository).

The definition file for Stopwatch is assembled from Countdown and Display.

```java
component Stopwatch {
  contains component Countdown c;
  contains component Display d;
  connects d.dr = c.cp;
}
```
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 (to implement the components)
- compile these with the header files to binary C code for Stopwatch.
In SOFA\textsuperscript{3} [56, 22, 21, 59], a component is an **architectural unit** which has a specification and an implementation, and is specified by its **frame** and **architecture**.

- The **frame** defines **provided** and **required interfaces**, and properties of the component.
- The **architecture** describes the **structure** of the component.

---

\textsuperscript{3}SOFTware Appliances
SOFA 2: Components
Including Run-time Control Interface and Microcomponents

Diagram:
- Frame
- Control Interface
- Standalone Microcomponents
- Delegation chain
- Business Provided Interface
- Component Content
- Business Required Interface
- Micro Component
SOFA: Composition

SOFA components are composed via connectors by using the following communication styles:

- **procedure call**: classic client server call.
- **messaging**: asynchronous message delivery from a producer to subscribed listeners.
- **streaming**: uni- or bidirectional stream of data between a sender and (multiple) recipients.
- **blackboard**: communication via shared memory.
SOFA components are constructed in SOFA IDE tool and deposited into the Repository of the tool.

- SOFA IDE tool is the **builder**.
- The Repository in SOFA IDE is the **repository**.
- There is **no** assembler.

Builder: A → AB

Repository: A → AB

RTE: A = SOFAA
     B = SOFAB
     AB = SOFAAB
     InsAB = SOFAAB instance

SOFA IDE: SOFA
           Repository: SOFA
           SOFA Anode: RTE

Connection: AB = connector
Component life cycle is separate from system life cycle

In component design phase, SOFA components are defined and deposited in the repository of the SOFA IDE

In system design/component deployment phase, SOFA components are retrieved from the repository and composed into a system

At run-time, the binary code of the system is executed in the run-time environment SOFANode
SOFA: Example

- The **Logger** component provides a log method.
- The **Tester** component calls the log method via Logger’s provided interface.
- Both components are composed in the **LogApplication** composite component.

Example taken from [http://sofa.ow2.org/docs/howto.html](http://sofa.ow2.org/docs/howto.html).
SOFA: Example (cont’d)

FRAME
<frame name="foo.FLogDemo"/>

ARCHITECTURE
<architecture name="foo ALOGDemo" frame="sofatype://foo.FLogDemo">
  <sub-comp name="tester" frame="sofatype://foo.FTester" arch="sofatype://foo.ATESTer"/>
  <sub-comp name="logger" frame="sofatype://foo.FLogger" arch="://foo.ALogger"/>
  <connection>
    <endpoint sub-comp="tester" itf="log"/>
    <endpoint sub-comp="logger" itf="log"/>
  </connection>
</architecture>

Tester

Logger

Atester Required Interface

Alogger Provided Interface

LogApplication
SOFA: Example (cont’d)

FRAME
<frame name="foo.FTester">
  <requires name="log" if- type="sofatype://foo.ILog"/>
</frame>

ARCHITECTURE
<architecture name="foo.ATester"
frame="sofatype://foo.FTester" impl="foo.ATester"/>

REQUIRED INTERFACE IMPLEMENTATION
public class ATester implements SOFAClient {
  public void setRequired(String name, Object iface) {
    if (name.equals("log")) {
      if (iface instanceof ILog) {
        logger = (ILog) iface;
        logger.log("Hello World")
      }
    }
  }
}

FRAME
<frame name="foo.FLogger">
  <provides name="log" if- type="sofatype://foo.ILog"/>
</frame>

ARCHITECTURE
<architecture name="foo.Alogger"
frame="sofatype://foo.FLogger" impl="foo.Alogger"/>

PROVIDED INTERFACE IMPLEMENTATION
public class ALogger implements ILog {
  public void log(String message) {
    System.out.println("LOG: " + message);
  }
}
KobrA: Components

In KobrA⁴ [11], a component is a UML component [25]. Every KobrA component has a specification and an implementation.

- The specification describes what a component does and thus it is the interface of the component.
- The implementation describes how it does it.

KobrA: Composition

KobrA components are composed by direct method calls.

⁴Komponenten-basierte Anwendungsentwicklung (component-based application development)
KobrA components can be constructed in a **visual builder tool** such as Visual UML and deposited into a **file system**.

- The **visual builder tool** is the builder
- The **file system** is the repository
- There is **no assembler**

![Diagram showing KobrA components](image-url)

- Builder
- Repository
- RTE
- A = KobrAA
- B = KobrAB
- AB = KobrAAB
- InsAB = KobrAAB instance
- = method call
Component life cycle is separate from system life cycle

In component design phase, KobrA components are defined in UML and deposited in the repository

In system design/component deployment phase, KobrA components are retrieved from the repository and composed into a system in UML, also deposited in the repository

All the components and the system have to be implemented in an object-oriented programming language

At run-time, an instance of the system is executed in the run-time environment of the chosen programming language
Consider a **book store** that maintains a database of its book stock and sells its books by an **Automatic Teller Machine (ATM)**.

<table>
<thead>
<tr>
<th>&lt;&lt;subject&gt;&gt;</th>
</tr>
</thead>
<tbody>
<tr>
<td>BookStore</td>
</tr>
<tr>
<td>noOfBooks: Integer:=0</td>
</tr>
<tr>
<td>addBook(Book b)</td>
</tr>
<tr>
<td>addBooks(Book[ ] blist)</td>
</tr>
<tr>
<td>viewBooks()</td>
</tr>
<tr>
<td>deleteBook(Book b)</td>
</tr>
<tr>
<td>findBook(Book b)</td>
</tr>
</tbody>
</table>

The specification of the BookStore component is a **UML class diagram** that specifies what the BookStore component does.
In design phase, the book store system is implemented by constructing a new **ATM** component and composing it with **BookStore** and **Book** components from the repository.

The book store system is **assembled** from the ATM, BookStore and Book components by **direct method calls**.
In SCA\(^5\) [10, 38], a component has services, references and properties.
SCA: Composition

Services Binding
- SOAP
- JMS
- JCS
- ...

Reference Binding
- SOAP
- JMS
- JCS
- ...

C₀

C₁

C₂

Wire
Promote
In SCA, components are constructed (in various programming languages) in the SCA IDE and stored in the SCA Repository. At run-time, SCA components are executed in various programming language RTEs.

- The SCA IDE is the **builder**
- The SCA Repository is the **repository**
- The RTE is that provided by the programming languages used

\[
\begin{align*}
A &= \text{SCAA} \\
B &= \text{SCAB} \\
AB &= \text{SCAAB} \\
\text{InsAB} &= \text{SCAAB instance} \\
\end{align*}
\]
Component life cycle and system life cycle are separated

In component/system design phase, SCA components are

- designed and implemented
- deposited into a repository (vendor specific)
- composed into a complete system

At run-time, client applications are executed, invoking services exposed by SCA components
SCA: Example

BigBank

Account
(Java)

AccData
(Java)

Stock

StockQuote
(C++)

Calculator

Add
(C)

Subtract
(Ruby)

Multiply
(Phyton)

Divide
(Groovy)

Picture taken from: https://cwiki.apache.org/confluence/display/TUSCANYWIKI/Building+SOA+With+Apache+Tuscany+Incubator

Lau et al (University of Manchester) Software Component Models CompArch 2014
In Palladio [15, 57], a component consists of:

- an interface
  - service signatures and (optional) protocols
  - and (optional) behavioural specifications
    - specified by using Service Effect Specification (SEFF)

Three (basic) component types: provided type → complete type → implementation type, in ascending order of concreteness of specifications

A basic component is an atomic component

A composite component or a system is an assembly of basic and other composite components
Composition is *port connection via connectors*

Connectors can be *assembly or delegation*

Picture taken from [57].
In **design phase**, (basic and composite) components are abstractly or concretely defined, assembled, and stored in repository. The **builder** is the **PCM tool**.

Also in **design phase**, components are chosen and assembled into systems.

System code skeleton is generated and then implemented using an implementation language such as Java.
Repository is not necessarily derived from domain requirements i.e. components can be identified during system design.

There is no clear separation between component design and deployment phase.

Components can be just abstract specifications.

Components do not necessarily have implementations.
Consider a simple ATM system that can read customers’ bank cards and provide basic services:

▶ withdraw
▶ deposit
▶ check balance

We identify three **atomic** components:

▶ CardReader
▶ Bank
▶ GUI
In design phase, we design the 3 identified components. We also build a composite component BankComposite from the atomic ones.
The composite component **BankComposite** is built by assembling **CardReader** and **Bank**.
To construct the system, we assemble BankComposite and GUI.
ProCom [58] is a two-layered component model.

**ProSys - upper layer**
- “Subsystem” components
- Active, distributed
- Asynchronous message passing

**ProSave - lower layer**
- “Function” components
- Passive, non-distributed
- Separation of data and control flow

**Connection between the two layers**
A subsystem component can internally be modelled by ProSave components
An atomic subsystem:

A composite subsystem:

Message channel
ProSys: Composition

- Message ports not directly connected
- Composition via explicit message channels
An Electronic Stability Control (ESC) system:

Picture taken from [58].
A ProSave component:

- is a unit of functionality, designed to encapsulate low-level tasks
- exposes its functionality via **services**, each consisting of:
  - an **input group** of ports: it contains the activation trigger and required data
  - an **output group** of ports: it makes available the data produced

A primitive component and its relative header file:

```c
typedef struct {
    int *speed;
    float *dist;
} in_S1;

typedef struct {
    int *control;
} out_S1;

void init();
void entry_S1(in_S1 *in, out_S1 *out);
```
ProSave: Composition

- Separated data and control flow
- Connectors for more elaborate control: Control fork, Control join, Control selection, Control or, Data fork, and Data or

A typical usage of selection and or-connectors:

![Diagram Image]

Picture taken from [23].
The Electronic Stability Control System:

Picture taken from [58].
ProCom components are constructed in the PRIDE tool and deposited into the repository of the tool.

- PRIDE tool is the **builder**.
- The repository in PRIDE is the **repository**.
- There is **no** assembler.
- The run-time environment is that of C/C++.
Component life cycle is separate from system life cycle

In component design phase, ProSys/ProSave components are defined and deposited in the repository of the PRIDE tool.

In system design/component deployment phase, ProSys/ProSave components are retrieved from the repository and composed into a system.

At run-time, the binary code of the system is executed in the run-time environment of C/C++. 
Taxonomy of Component Models: Category 5

Category 5: Design and Deploy with Repository (X-MAN)

Builder
A
B

Repository
A
AB

Assembler
A
AB

RTE
InsA
InsAB

Lau et al. (University of Manchester)
In X-MAN [46, 45, 36, 42], components \textit{encapsulated} units of computation, with \textit{only provided services}.

- an \textbf{atomic component} contains an \textit{invocation connector} (IU) and a \textit{computation unit} (U); the invocation connector, when activated by control coming from a composition connector, invokes methods provided by the computation unit.

- a \textbf{composite component} contains sub-components composed by composition connectors; composite components are \textit{self-similar}.
Components are composed by **composition connectors**, which

- encapsulate control
- coordinate control flow between components.

---

**Lau et al. (University of Manchester)**

Software Component Models

CompArch 2014  148 / 177
X-MAN is supported by the X-MAN tool. In this tool, components (both atomic and composite) are **built** in the **builder** and **deposited** in the **repository**. Components are retrieved from the repository and composed into a system in the **assembler**. The system is executed in the **simulator** of the X-MAN tool.
Component life cycle is separate from system life cycle

In component design phase, X-MAN components (both atomic and composite) are defined and constructed and deposited in the repository of the X-MAN tool.

In system design/component deployment phase, X-MAN components are retrieved from the repository and composed into a system in the assembler of the X-MAN tool.

At run-time, the binary code of the system is executed in the simulator of the X-MAN tool.
Consider a simple **passenger door management system** on a aircraft. The system determines to engage or disengage the door locks or issue warnings based on air speed, pressure, door handle position, door latch and emergency status.
In the design phase, three atomic components `CLLVoter`, `PswController` and `LockingController` are designed and deposited in a repository. All atomic components in X-MAN are fully implemented with source code (e.g. written in C/C++):
Also in the design phase, a composite component **DoorController** is designed by composing the formerly designed atomic components. **DoorController** is then deposited in a repository:
In the deployment phase, two instances (one for each aircraft door) of `DoorController` are deployed and composed into the system:
Part IV

- Future challenges and new CBSE desiderata
- Future component models
- Future life cycles
- Conclusion
### Well-known benefits of CBD

- reduced production cost
- reduced time-to-market
- increased software reuse

Even greater benefits of CBD?
- increased scale
- increased complexity
- increased safety

What would be the key?
- composition and compositionality
- compositional construction
- compositional V&V
- compositional product line engineering?
## Future Challenges and New Desiderata

### Well-known benefits of CBD
- reduced production cost
- reduced time-to-market
- increased software reuse

### Even greater benefits of CBD?
- increased scale
- increased complexity
- increased safety

---

Lau et al (University of Manchester)  
Software Component Models  
CompArch 2014  
156 / 177
Future Challenges and New Desiderata

Well-known benefits of CBD
- reduced production cost
- reduced time-to-market
- increased software reuse

Even greater benefits of CBD?
- increased scale
- increased complexity
- increased safety

What would be the key?
- composition and compositionality
## Future Challenges and New Desiderata

### Well-known benefits of CBD
- reduced production cost
- reduced time-to-market
- increased software reuse

### Even greater benefits of CBD?
- increased scale
- increased complexity
- increased safety

### What would be the key?
- composition and compositionality
  - compositional construction
  - compositional V&V
  - compositional product line engineering?
Compositional Construction
Towards Increased Scale, Complexity and Safety

Additional Desiderata for Composition
- hierarchical (algebraic) composition mechanisms
- (algebraic) composition operators

Existing Software Composition Mechanisms

**Containment**
- U3
- U1
- U2

**Extension**
- U1
- U2
- U3
- extension connection

**Coordination**
- Coordinator
- U1
- U2
- communication channel

**Connection**
- (a) Direct message passing
- (b) Indirect message passing

Lau et al (University of Manchester) Software Component Models CompArch 2014 157 / 177
A Taxonomy of Software Composition Mechanisms

<table>
<thead>
<tr>
<th>Unit of Composition</th>
<th>Composition Mechanism</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Containment</td>
</tr>
<tr>
<td>Function</td>
<td>Function nesting</td>
</tr>
<tr>
<td>Procedure</td>
<td>Procedure nesting</td>
</tr>
<tr>
<td>Class</td>
<td>Class nesting</td>
</tr>
<tr>
<td>Mixin</td>
<td>Mixin inheritance</td>
</tr>
<tr>
<td>Mixin/Class</td>
<td>Mixin-class inheritance</td>
</tr>
<tr>
<td>Trait</td>
<td>Trait composition</td>
</tr>
<tr>
<td>Trait/Class</td>
<td>Trait-class composition</td>
</tr>
<tr>
<td>Subject</td>
<td>Subject composition</td>
</tr>
<tr>
<td>Feature</td>
<td>Feature composition</td>
</tr>
<tr>
<td>Aspect/Class</td>
<td>Weaving</td>
</tr>
<tr>
<td>Module</td>
<td>Module nesting</td>
</tr>
<tr>
<td>Architectural unit</td>
<td>Port connection</td>
</tr>
<tr>
<td>Fragment box</td>
<td>Invasive composition</td>
</tr>
<tr>
<td>Process</td>
<td>Channels</td>
</tr>
<tr>
<td>Web service</td>
<td>Orchestration (Control coordination)</td>
</tr>
<tr>
<td>Encapsulated component</td>
<td>Exogenous composition (Control coordination)</td>
</tr>
</tbody>
</table>

## Algebraic Software Composition Mechanisms

<table>
<thead>
<tr>
<th>Composition Mechanism</th>
<th>Algebraic ?</th>
</tr>
</thead>
<tbody>
<tr>
<td>Containment</td>
<td>Extension</td>
</tr>
<tr>
<td></td>
<td>Mixin-class inheritance</td>
</tr>
<tr>
<td></td>
<td>Trait-class composition</td>
</tr>
<tr>
<td>Weaving</td>
<td>Port connection</td>
</tr>
<tr>
<td></td>
<td>Trait-class composition</td>
</tr>
<tr>
<td></td>
<td>Multiple inheritance</td>
</tr>
<tr>
<td>Procedure nesting</td>
<td>Mixin inheritance</td>
</tr>
<tr>
<td>Module nesting</td>
<td>Trait composition</td>
</tr>
<tr>
<td>Class nesting</td>
<td>Subject composition</td>
</tr>
<tr>
<td>Object composition</td>
<td>Feature composition</td>
</tr>
<tr>
<td>Object aggregation</td>
<td>Invasive composition</td>
</tr>
</tbody>
</table>

### Algebraic Composition Mechanism

<table>
<thead>
<tr>
<th>Containment</th>
<th>Extension</th>
<th>Connection</th>
<th>Coordination</th>
<th>Composition operator ?</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Mixin inheritance</td>
<td>Higher-order function</td>
<td>Exogenous composition</td>
<td>Yes</td>
</tr>
<tr>
<td></td>
<td>Subject composition</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Function nesting</td>
<td>Multiple inheritance</td>
<td>Trait composition</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Procedure nesting</td>
<td>Trait composition</td>
<td>Port connection</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Module nesting</td>
<td>Feature composition</td>
<td>Invasive composition</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Class nesting</td>
<td>Invasive composition</td>
<td>Channels</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Object composition</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Object aggregation</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Compositional Construction
The X-MAN Component Model

X-MAN: Encapsulated Components + Exogenous Composition

Atomic component

Composition connector

Composite component

Projects (European Artemis JU)

**CESAR**: Cost Efficient Methods and Processes for Safety Relevant Embedded Systems (57 partners; budget: €58M)

**EMC2**: Embedded Multi-Core Systems for Mixed Criticality Applications in Dynamic and Changeable Real-Time Environments (96 partners; budget: €98M)

Compositional Construction in X-MAN

CESAR Project: Aircraft Fuel System
Compositional Construction in X-MAN

Aircraft Fuel System: Component-based Design

Diagram:

- ManualControlDeployment
  - ManualTankControlComposite x 3
    - ManualPumpControlComposite x 2
      - SwitchTimer
      - PumpFaultLogic
      - ManualPumpLogic
    - Constants
      - ManualValveControl
- Atomic
  - Composite
  - Deployment
Compositional V&V must be based on:
- compositional construction with
- separate component and system life cycles
Compositional V & V

Need to adapt the V model accordingly.

The V Model: Modular System Development

- System requirements
- System specification
- Architectural design
- Module design
- Coding
- Acceptance test plan
- System test plan
- Integration test plan
- Unit test plan
- Acceptance testing
- System testing
- Integration testing
- Unit testing

V&V
Compositional V & V

The straightforward adaptation does not work.

The V Model: Component-based System Development?

- System requirements
  - System specification
    - Architectural design
      - Component design
        - Coding
  - Component testing
    - Integration testing
  - System testing
    - Acceptance testing
  - Acceptance test plan

V&V
Compositional V & V

Need one V for each life cycle.

The W Model
Compositional V & V
Aircraft Fuel System: X-MAN Model Checker

SwitchTimer.TimeSwitch <= SwitchTimer_spec.TimeSwitch ...
component-verification-harness893959022652986669.cpp
false

Counterexample:

State 2 file <built-in> line 27 thread 0
______________________________________________
__CPROVER_deallocated=NULL (000000000000000000000000000000000)

State 3 file <built-in> line 28 thread 0
______________________________________________
__CPROVER_malloc_object=NULL (000000000000000000000000000000000)

State 4 file <built-in> line 9 thread 0
______________________________________________
__CPROVER_malloc_is_new_array=FALSE (0)

State 6 file c:\program files\microsoft visual studio 10 0\vc\include\co...

Top-level property:

Compositional V & V
Aircraft Fuel System: X-MAN Theorem Prover

```
TankSeq {
  seq-op: {(TrimTank_iPumpMSwitch,Nat), (TrimTank_iPumpMLp,Nat), (Trim
}

Computing strongest post-conditions ...
seq-op: {[(oTrOn + -1 * TrimTank_oVal)]}
```

Lau et al. (University of Manchester) Software Component Models CompArch 2014 170 / 177
Compositional V & V
Aircraft Fuel System: Proving at Multiple Levels

Composite (theorem proving)

Deployment (theorem proving)

Atomic (model checking)
Compositional V & V
Aircraft Fuel System: Proving at Atomic and Composite Levels

Atomic level:

```plaintext
// Pump always on if switch is on and pump is not faulty
\exists n \in \mathbb{N} \forall t \in [n, \infty) \exists t' \in [t, t+1) \forall t'' \in [t', t' + 1) \exists t''' \in [t'', t'' + 1)
(FALSE = 0) \&
( ((iSwitchOn != FALSE) \& (iPumpFaulty = FALSE)) \&
(imass > 0)
) \rightarrow
(oOn != FALSE) )
```

Composite level:

```plaintext
// Pump switch timer greater than zero if-and-only-if switch is on
((oSwitchOnTicks > 0) \leftrightarrow (iSwitchOn != 0))
\&
// Pump always off if switches off
((oSwitchOnTicks = 0) \rightarrow (oPumpOn = 0))
\&
// Pump always off if zero mass in tank
((imass = 0) \rightarrow (oPumpOn = 0))
\&
// Pump always off if on for 5 ticks or more and still LP detected
(((oSwitchOnTicks >= 5) \& (iPumpLp = 0)) \rightarrow (oPumpOn = 0))
\&
// Pump always on if mass in tank and switch on, but less than 5 ticks
(((imass > 0) \& (oSwitchOnTicks > 0) \& (oSwitchOnTicks < 5)) \rightarrow (oPumpOn != 0))
\&
// If switch on for more than 5 ticks and mass in tank, pump on iff pump pressure
(((oSwitchOnTicks >= 5) \& (imass > 0)) \rightarrow ((oPumpOn != 0) \leftrightarrow (iPumpLp = 0)))
```
Compositional V & V
Aircraft Fuel System: Top-level Proof

Contracts of components not checked again.

Top level property: 

\((\text{SwitchOnTicks} >= 5) \& \& (\text{Mass} > 0) \rightarrow (\text{PumpOn} = 0) \leftrightarrow (\text{PumpLP} = 0))\)
Current PLE practice

- focuses on variability management (using feature model only)
- lacks product architectures (product line ≠ architecture)
- lacks reference architecture (feature model + functional model)
- lacks scalability

<table>
<thead>
<tr>
<th>Product line engineering</th>
<th>Domain engineering</th>
<th>Product engineering</th>
</tr>
</thead>
<tbody>
<tr>
<td>Feature model + Functional model</td>
<td>Reference architecture</td>
<td>Reference architecture</td>
</tr>
<tr>
<td>Reference architecture</td>
<td>All product variants</td>
<td></td>
</tr>
</tbody>
</table>
Compositional Product Line Engineering?

For scalability

Use tree-like product architectures and hence reference architecture?

Feature Model Tree

Legend:

- Requires
- XOR
- Mandatory

Automobile

Brake Control Software

Brake ECU

ABS Controller
Non-ABS Controller
1 Mbit/s CAN Bus
250kbit/s CAN Bus

'Spaghetti' Products

'Tree' Products
PLE with the W Model

Domain Engineering

- Feature Model + Functional Model
- Reference Architecture

Domain knowledge

- Component design
- Component V&V & certification
- Coding

Component Life Cycle

- Product requirements
- Product specification
- Component selection & adaptation
- Product assembly
- Compositional V&V
- Coding

Product Life Cycle

- Acceptance testing
- Product V&V
- All Products

Product Engineering

- Reference Architecture

Lau et al. (University of Manchester)
Conclusion

Past
CBD identified desiderata

Present
CBD delivering following benefits:
- reduced production cost
- reduced time-to-market
- increased software reuse

Future
CBD to deliver even greater benefits:
- increased scale
- increased complexity
- with safety?


   http://www.cs.cmu.edu/~able/wright/.

   http://www.w3schools.com/Webservices/default.asp.


   Architectural reasoning in ArchJava.

   ArchJava: Connecting software architecture to implementation.


    Component-based Product Line Engineering with UML.


CORBA Components.

The Palladio component model for model-driven performance prediction.

Pure Corba.

Essential COM.
Addison-Wesley, 1998.

What characterizes a software component?

An open component model and its support in Java.

The fractal component model and its support in java.

SOFA 2.0: Balancing Advanced Features in a Hierarchical Component Model.

Communication style driven connector configurations.

[23] T. Bures et al.
Y: A new component-based software lifecycle model.

*UML Components: A Simple Process for Specifying Component-Based Software.*

CBD process.

[27] COM web page.

Component-based development process and component lifecycle.

A classification framework for software component models.


AMBERS: Improving Requirements Specification Through Assertive Models and SCADE/DOORS Integration.

http://fractal.ow2.org/.

Acme: Architectural description of component-based systems.
[34] A.P. Gaufillet and B.S. Gabel.  
Avionic software development with TOPCASED SAM.  

*PECOS in a Nutshell.*  

Component-based design and verification in X-MAN.  

*Component-Based Software Engineering: Putting the Pieces Together.*  

[38] IBM.  

http://docs.oracle.com/javase/tutorial/javabeans/.

Candidate process models for component based software development.  

Towards a classification model for component-based software engineering research.  

Component-based development.  
A taxonomy of software composition mechanisms.

The W Model for component-based software development.

X-MAN: An MDE tool for component-based system development.

Exogenous connectors for software components.

A taxonomy of software component models.

Software component models.

COM IDL and Interface Design.

The grand challenge of trusted components.

Enterprise JavaBeans.


http://www.v-modell.iabg.de.

[64] R. van Ommering.  
The Koala component model.  

The Koala component model for consumer electronics software.  

Microsoft .NET Compact Framework (Core Reference).  