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

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

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

Past

- 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

Introduction

Software Component Models

Definition

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

[48] K.-K. Lau and Z. Wang. Software Component Models. *IEEE Transactions on Software Engineering* 33(10):709-724, 2007.

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.”

Component Definition	Based on Component Model?	Defines Component Model?
Szyperski	No	No
Meyer	No	No
Heineman & Councill	Yes	No

Models versus Frameworks

Component Models versus Component(-oriented Programming) Frameworks

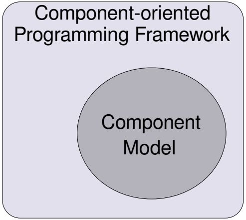
Component Frameworks

- 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



Introduction

CBSE: Present & Future

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

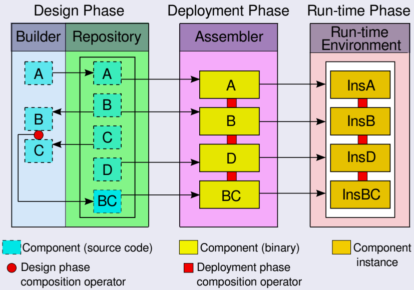
- Components should pre-exist
- Components should be produced independently
- Component should be deployed independently
- It should be possible to copy and instantiate components
- It should be possible to build composites
- It should be possible to store composites

[18] M. Broy, A. Deimel, J. Henn, K. Koskimies, F. Plasil, G. Pomberger, W. Pree, M. Stal and C. Szyperski. What characterizes a software component? *Software — Concepts and Tools* 19:49-56, 1998.

Idealised Component Life cycle

Composition in Component Design Phase and Component Deployment Phase

Idealised Component Life Cycle



[48] K.-K. Lau and Z. Wang. Software Component Models. *IEEE Transactions on Software Engineering* 33(10):709-724, 2007.

CBSE Desiderata

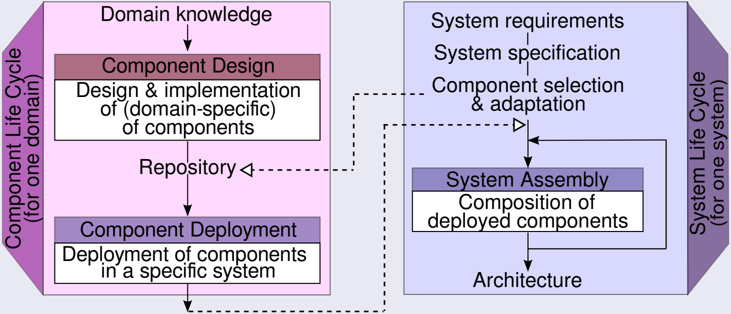
Desideratum	Design Phase	Deployment Phase
Components should pre-exist	Deposit components in repository	Retrieve components from repository
Components should be produced independently	Use builder	——
Components should be deployed independently	——	Use assembler
It should be possible to copy and instantiate components	Copies possible	Copies and instances possible
It should be possible to build composites	Composition possible	Composition possible
It should be possible to store composites	Use repository	——

[18] M. Broy, A. Deimel, J. Henn, K. Koskimies, F. Plasil, G. Pomberger, W. Pree, M. Stal and C. Szyperski. What characterizes a software component? *Software — Concepts and Tools* 19:49-56, 1998.

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

Idealised Component and System Life Cycles

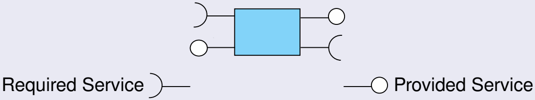


[44] K.-K. Lau, F. Taweel and C. Tran. The W Model for Component-based Software Development. In *Proc. 37th EUROMICRO Conference on Software Engineering and Advanced Applications*, pages 47–50, IEEE, 2011.

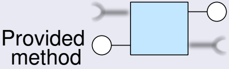
Current Component Models

Components

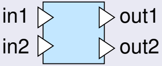
A Generic Component



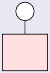
An Object



An Architectural Unit



An Encapsulated Component

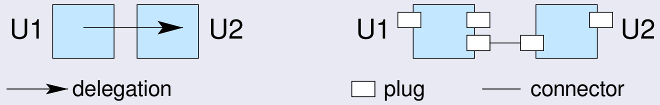


Components	Provided services	Required services	Composition mechanism
Objects	Methods	——	Method call
Architectural units	Out-ports	In-ports	Port connection
Encapsulated components	Methods	*None*	Exogenous composition

Current Component Models

Composition Mechanisms

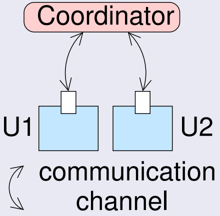
Connection: Method Call & Port Connection



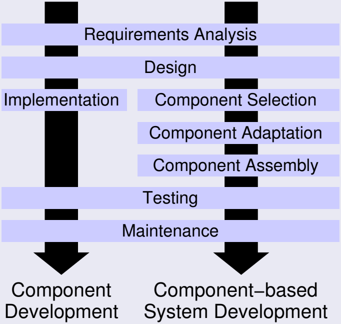
(a) Direct message passing

(b) Indirect message passing

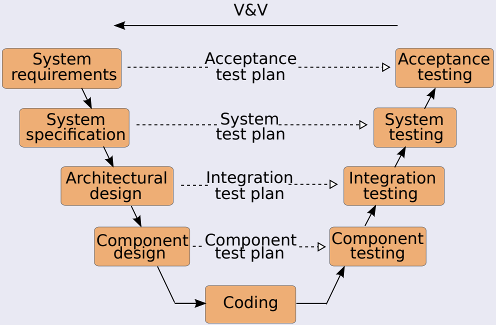
Coordination: Exogenous Composition



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])

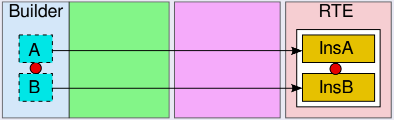
Current Component Models

Support for Idealised Component and System Life Cycles

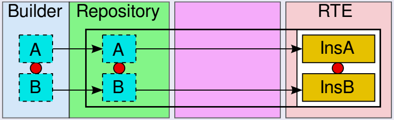
Category	Component Models	Design				Deploy
		Deposit-N	Retrieve	Compose	Deposit-C	Compose
Design without Repository	Acme-like ADLs UML2.0, PECOS	×	×	✓	×	×
Design with Deposit-only Repository	EJB, OSGi, Fractal COM, .NET, CCM	✓	×	✓	×	×
Deployment with Repository	JavaBeans, Web Services	✓	×	×	×	✓
Design with Repository	Koala, SOFA, Kobra SCA, Palladio, ProCom	✓	✓	✓	✓	×
Design & Deployment with Repository	X-MAN	✓	✓	✓	✓	✓

Deposit-N=Deposit components constructed from scratch
Deposit-C=Deposit composite components constructed from existing components

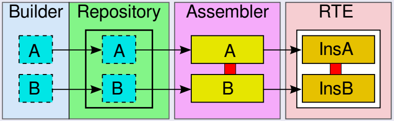
Taxonomy of Component Models



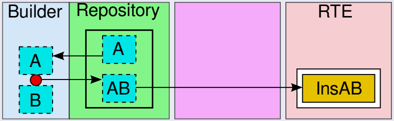
Category 1: Design without Repository
(Acme–like ADLs, UML2.0, PECOS)



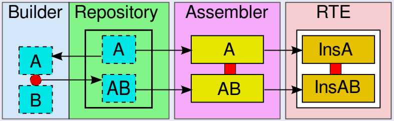
Category 2: Design with Deposit–only Repository
(EJB, OSGi, Fractal, COM, .NET, CCM)



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)

Part II

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

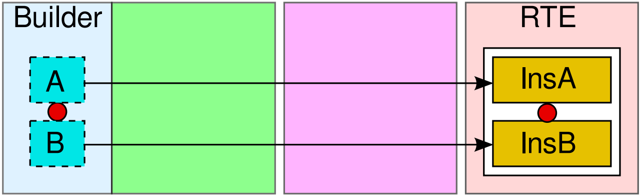
Taxonomy of Component Models

Overview

Category	Component Models	Design				Deploy
		Deposit-N	Retrieve	Compose	Deposit-C	Compose
Design without Repository	Acme-like ADLs UML2.0, PECOS	×	×	✓	×	×
Design with Deposit-only Repository	EJB, OSGi, Fractal COM, .NET, CCM	✓	×	✓	×	×
Deployment with Repository	JavaBeans, Web Services	✓	×	×	×	✓
Design with Repository	Koala, SOFA, Kobra SCA, Palladio, ProCom	✓	✓	✓	✓	×
Design & Deployment with Repository	X-MAN	✓	✓	✓	✓	✓

Deposit-N=Deposit components constructed from scratch
Deposit-C=Deposit composite components constructed from existing components

Taxonomy of Component Models: Category 1



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

Taxonomy of Component Models: Category 1

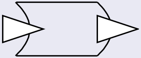
Acme-like ADLs

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

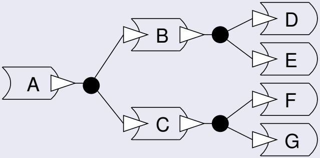
In Acme-like ADLs , a component is an architectural unit that represents a primary computational element and data store of a system.



- 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

Acme-like ADLs: Composition

- In Acme-like ADLs, components are composed by connectors
- Connectors connect components via their ports

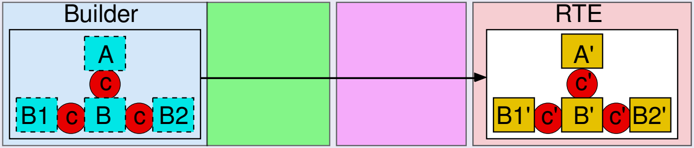


Acme-like ADLs

Support for Idealised Component and System Life Cycles

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



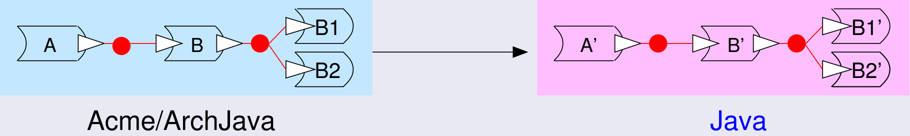
A = component A
B = component B
B1 = component B1
B2 = component B2
c = connector

A' = implementation of A
B' = implementation of B
B1' = implementation of B1
B2' = implementation of B2
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.



Acme: Example

Consider a simple bank system consisting of an ATM component, a BankConsortium component, and 2 Bank components Bank1 and Bank2.

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

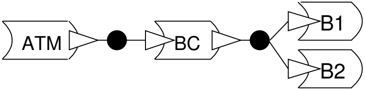
```
Component BankConsortium = {
        Port  receive;
        Port  send;
 }
```

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

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

Acme: Example (cont'd)

In design phase, the architecture for the whole system is designed



using the above components and the following connectors:

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

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

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

Acme: Example (cont'd)

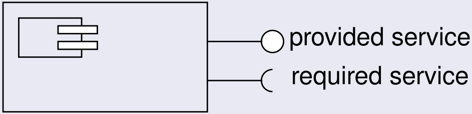
```
System BankSys = {
    Component ATM = {                                   Component BankConsortium = {
          Port  send;                                         Port  receive;
    };                                                        Port  send;
                                                        };

    Component Bank1 = {                                 Component Bank2 = {
          Port  receive;                                      Port  receive;
          Property  bankid : String = "Bank1";                Property  bankid : String = "Bank2";
    };                                                  };

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

    Connector BankContoB1 = {                            Connector BankContoB2 = {
          Role  request;                                       Role  request;
          Role  produce;                                       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;
     }
}
```

Taxonomy of Component Models: Category 1

UML2.0

UML2.0 Component Model: Components

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.

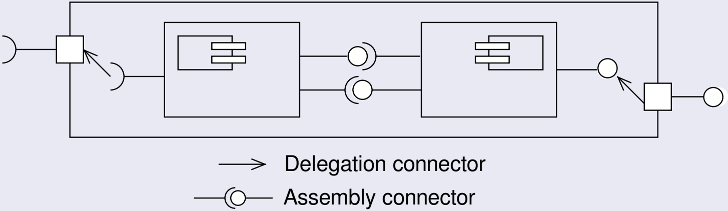
UML 2.0: Composition

UML2.0 components are composed by UML connectors:

- delegation connectors
- assembly connectors

Composites are assembled by assembly connectors

Systems are assembled by delegation and assembly connectors

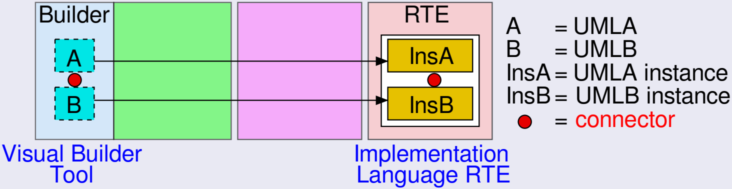


UML2.0

Support for Idealised Component and System Life Cycles

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



A = UMLA
B = UMLB
InsA = UMLA instance
InsB = UMLB instance
● = connector

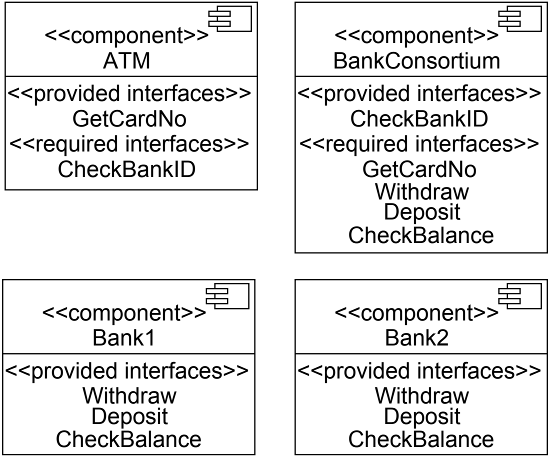
UML2.0

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.

UML 2.0: Example

Consider a simple bank system that is implemented by ATM, BankConsortium, Bank1 and Bank2 components.



<<component>>
ATM

<<provided interfaces>>
GetCardNo
<<required interfaces>>
CheckBankID

<<component>>
BankConsortium

<<provided interfaces>>
CheckBankID
<<required interfaces>>
GetCardNo
Withdraw
Deposit
CheckBalance

<<component>>
Bank1

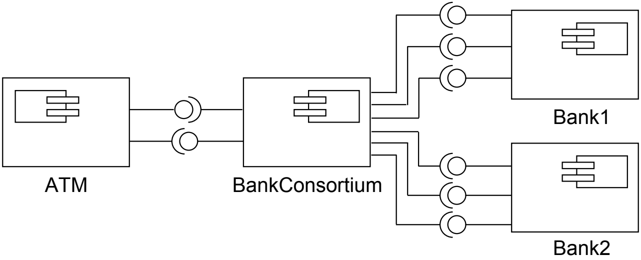
<<provided interfaces>>
Withdraw
Deposit
CheckBalance

<<component>>
Bank2

<<provided interfaces>>
Withdraw
Deposit
CheckBalance

UML2.0: Example (cont'd)

In design phase, the architecture for the whole system is designed.

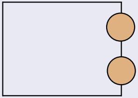


Taxonomy of Component Models: Category 1

PECOS

PECOS: Components

In PECOS[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

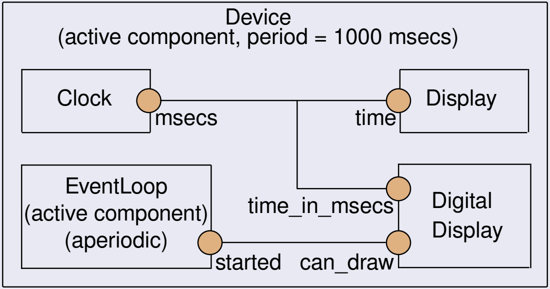
PECOS components are specified in the CoCo (Component Composition) language.

---

[1] PErvasive COmponent Systems

PECOS: Composition

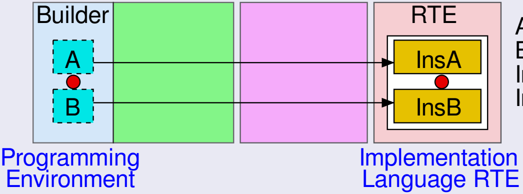
- In PECOS, components are composed by connectors
- Connectors connect components via their ports



PECOS

Support for Idealised Component and System Life Cycles

In PECOS, the components and the system are designed and constructed together in a programming environment such as Eclipse.



A = PECOSA
B = PECOSB
InsA = PECOSA instance
InsB = PECOSB instance
● = connector

- 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

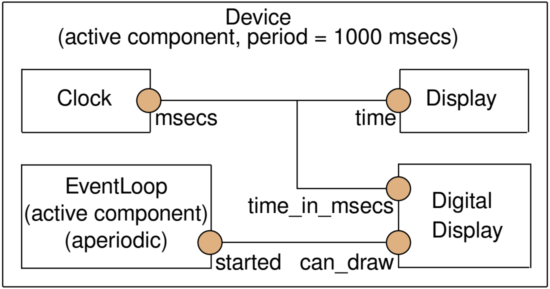
Consider a device that is assembled from Clock, Display, EventLoop and DigitalDisplay components.

```
component Clock {
output long msecs;
}

active component EventLoop {
output bool started;
}
```

```
component Display {
input long time;
}

component DigitalDisplay {
input long time_in_msecs;
input bool can_draw;
}
```

PECOS: Example (cont'd)

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);
}
```

Taxonomy of Component Models: Category 2

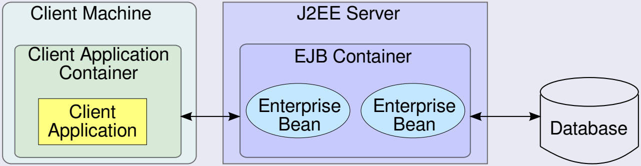
Category 2: Design with Deposit–only Repository
(EJB, OSGi, Fractal, COM, .NET, CCM)

Taxonomy of Component Models: Category 2

Enterprise JavaBeans (EJB)

EJB: Components

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.

EJB: Components (cont'd)

For an EJB:

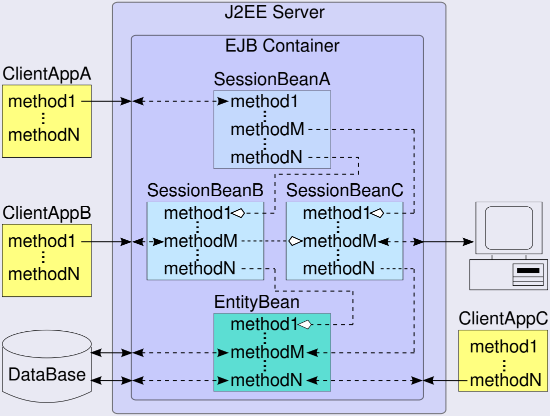
- its Java class defines the methods of the bean
- its interface exposes the capabilities of the bean and provides all the methods needed for (remote) client applications to access the bean (over a network)

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.

EJB: Composition

Enterprise beans are composed (in the EJB container) by method and event delegation

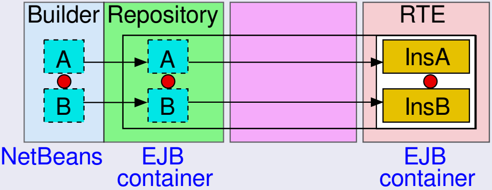


EJB

Support for Idealised Component and System Life Cycles

EJBs are constructed and composed in a J2EE-compliant IDE, and deposited and executed in an EJB contanier.

- 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



A = EJBA (JAR file)
B = EJBB (JAR file)
InsA = EJBA instance
InsB = EJBB instance
● = method call

EJB

Component and System Life Cycles

- 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.

EJB: Example

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)

```
@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;   }

   ...
```

EJB: Example (cont'd)

To construct the system we also need a session bean Bank that consists of a Java class and interface:

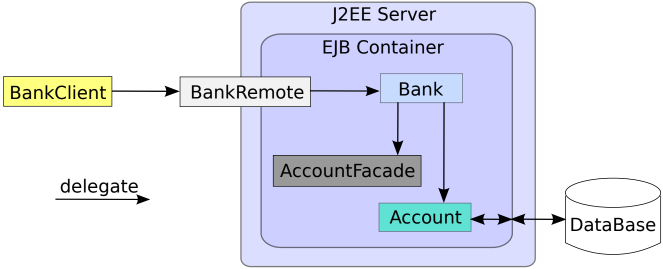
- Bank is the Java class that defines the business methods (services on accounts)
- BankRemote is the remote interface

EJB: Example (cont'd)

```
@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." );
  ...
```

EJB: Example (cont'd)

The system is assembled from the Account entity bean and the Bank session bean:

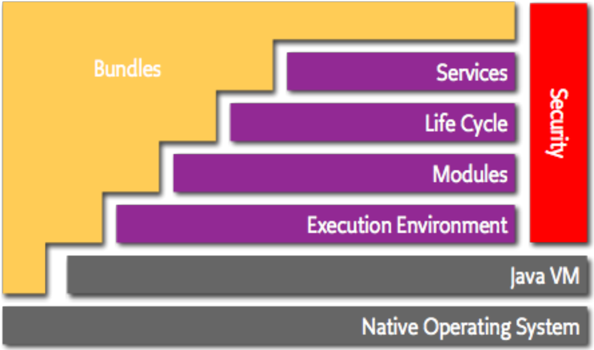


Taxonomy of Component Models: Category 2

OSGi Component Model

OSGi

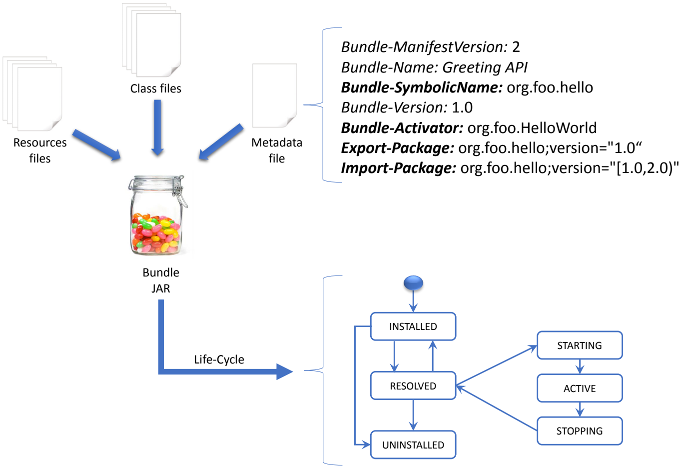
A component framework that brings modularity to JAVA platform



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

OSGi: Bundles

OSGi consists of bundles:

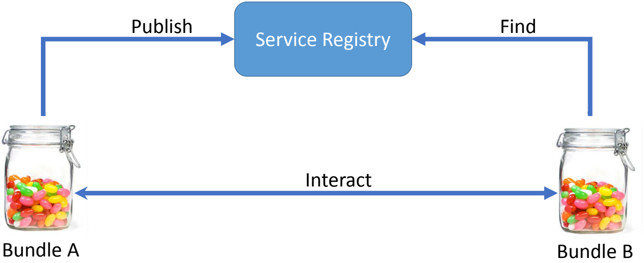


Bundle-ManifestVersion: 2
Bundle-Name: Greeting API
Bundle-SymbolicName: org.foo.hello
Bundle-Version: 1.0
Bundle-Activator: org.foo.HelloWorld
Export-Package: org.foo.hello;version="1.0“
Import-Package: org.foo.hello;version="[1.0,2.0)"

OSGi Component Model

Components and Composition

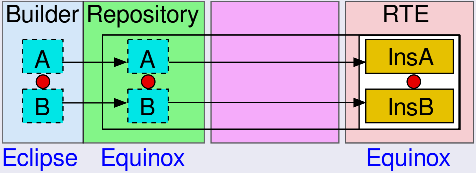
- OSGi bundles do not compose, but POJOs within them do via direct method invocation.
- So components in OSGi component model are Java objects; and composition is by direct method call.



OSGi Component Model

Support for Idealised Component and System Life Cycles

- 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

OSGi Component Model

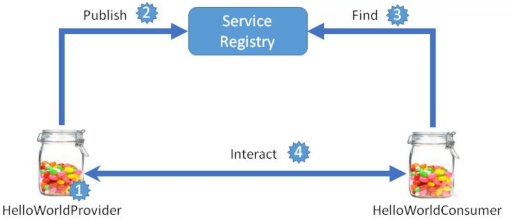
Component and System Life Cycles

- 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

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

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



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

```
public class HelloWorldConsumer{
   public HelloWorldConsumer(HelloWorldProvider
                                 helloWorldProvider) {
        helloWorldProvider.hello();
    }}
```

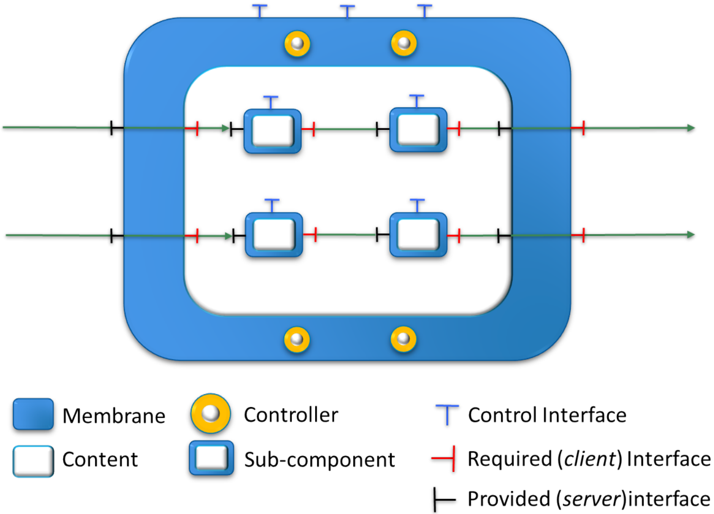
Taxonomy of Component Models: Category 2

Fractal: Components

In Fractal [19, 20, 32], a component:

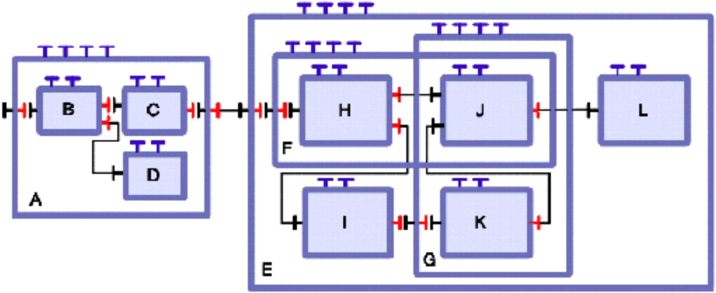
- is a unit of encapsulation and behaviour
- consists of two parts:
  - content
    - a finite set of sub-components
  - membrane
    - typically composed of several controllers, each in charge of a specific function
    - supports interfaces to introspect and reconfigure its internal features
    - 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)

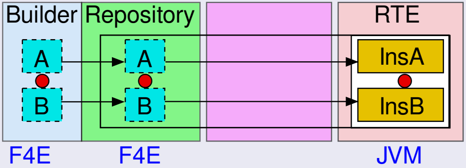


Fractal

Support for Idealised Component and System Life Cycles

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



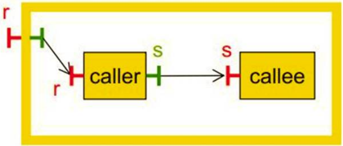
A = FractalA (JAR file)
B = FractalB (JAR file)
InsA = FractalA instance
InsB = FractalB instance
● = method call

Fractal

Component and System Life Cycles

- 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.lang.Runnable”

   <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>
```

External definition in file CallerImpl.fractal

External definition in file CalleeImpl.fractal

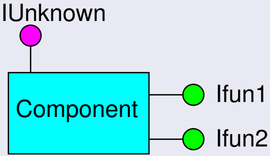
http://fractal.ow2.org/doc/ow2-webinars09/Fractal-Java-Lionel.pdf

Taxonomy of Component Models: Category 2

COM

COM: Components

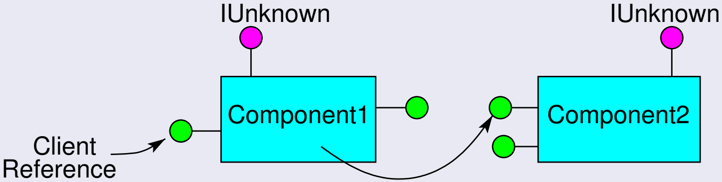
In COM (Component Object Model) [17, 49, 54, 27], a component is a unit of compiled code on Windows Registry.



- 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: Composition

COM components are composed by method calls via interface pointers

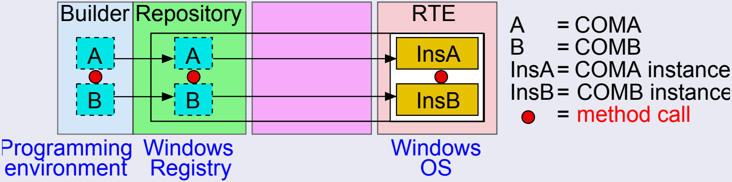


COM

Support for Idealised Component and System Life Cycles

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



A = COMA
B = COMB
InsA = COMA instance
InsB = COMB instance
● = method call

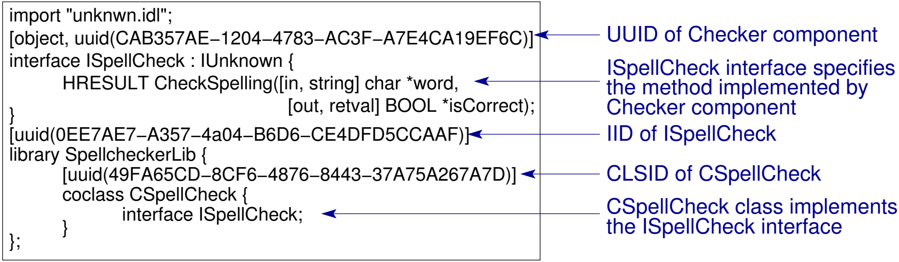
COM

COM Component and System Life Cycles

- 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.

COM: Example

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)]
        coclass CSpellCheck {
                interface ISpellCheck;
        }
};
```

UUID of Checker component

ISpellCheck interface specifies the method implemented by Checker component

IID of ISpellCheck

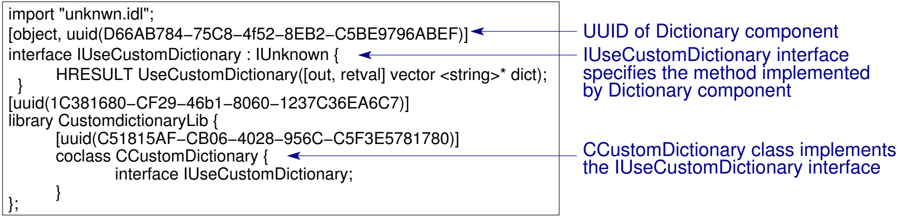
CLSID of CSpellCheck

CSpellCheck class implements the ISpellCheck interface

Checker component interface –– ISpellCheck

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

COM: Example (cont’d)



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

UUID of Dictionary component

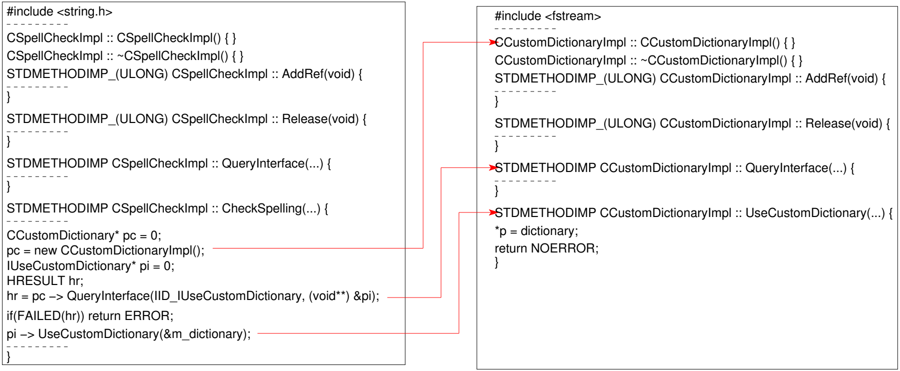
IUseCustomDictionary interface specifies the method implemented by Dictionary component

CCustomDictionary class implements the IUseCustomDictionary interface

Dictionary component interface –– IUseCustomDictionary

COM: Example (cont’d)

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



```
#include <string.h>
- - - - - - - - -
CSpellCheckImpl :: CSpellCheckImpl() { }
CSpellCheckImpl :: ~CSpellCheckImpl() { }
STDMETHODIMP_(ULONG) CSpellCheckImpl :: AddRef(void) {
- - - - - - - - -
}
STDMETHODIMP_(ULONG) CSpellCheckImpl :: Release(void) {
- - - - - - - - -
}
STDMETHODIMP CSpellCheckImpl :: QueryInterface(...) {
- - - - - - - - -
}
STDMETHODIMP CSpellCheckImpl :: CheckSpelling(...) {
- - - - - - - - -
CCustomDictionary* pc = 0;
pc = new CCustomDictionaryImpl();
IUseCustomDictionary* pi = 0;
HRESULT hr;
hr = pc –> QueryInterface(IID_IUseCustomDictionary, (void**) &pi);
if(FAILED(hr)) return ERROR;
pi –> UseCustomDictionary(&m_dictionary);
- - - - - - - - -
}
```

Checker component implementation

```
#include <fstream>
- - - - - - - - -
CCustomDictionaryImpl :: CCustomDictionaryImpl() { }
CCustomDictionaryImpl :: ~CCustomDictionaryImpl() { }
STDMETHODIMP_(ULONG) CCustomDictionaryImpl :: AddRef(void) {
- - - - - - - - -
}
STDMETHODIMP_(ULONG) CCustomDictionaryImpl :: Release(void) {
- - - - - - - - -
}
STDMETHODIMP CCustomDictionaryImpl :: QueryInterface(...) {
- - - - - - - - -
}
STDMETHODIMP CCustomDictionaryImpl :: UseCustomDictionary(...) {
*p = dictionary;
return NOERROR;
}
```

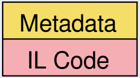
Dictionary component implementation

Taxonomy of Component Models: Category 2

.NET Component Model

.NET Component Model: Components

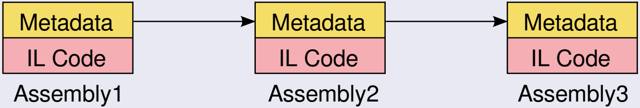
In Microsoft .NET [55, 66, 2], a component is an assembly that is a binary unit supported by Common Language Runtime (CLR)



- 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 Component Model: Composition

.NET components are composed by method calls through references via metadata

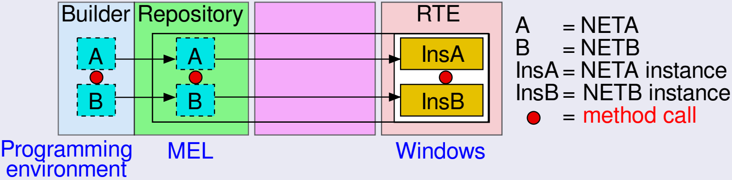


.NET Component Model

Support for Idealised Component and System Life Cycles

.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



A = NETA
B = NETB
InsA = NETA instance
InsB = NETB instance
● = method call

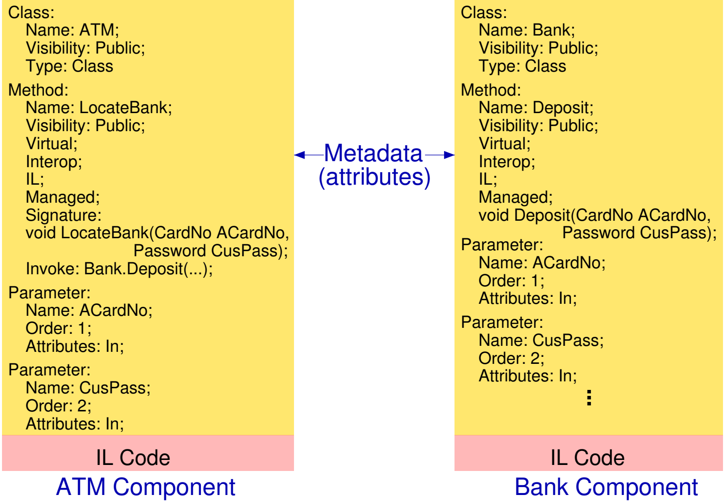
.NET

Component and System Life Cycles

- 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.

.NET: Example

Consider a banking system with an ATM component, which serves two instances Bank1 and Bank2 of a Bank 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;
```

IL Code

ATM Component

Metadata (attributes)

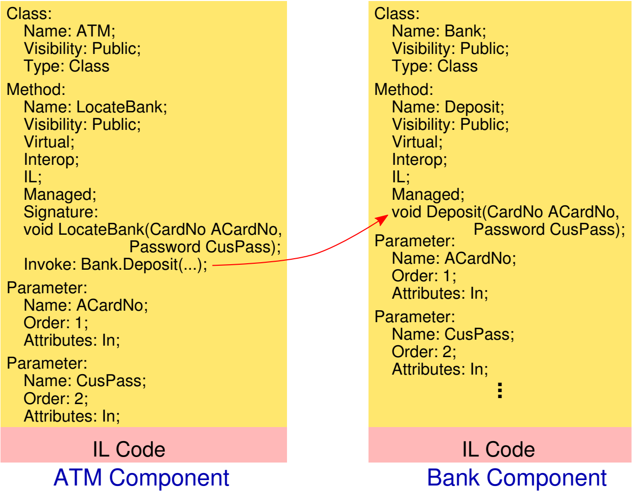
```
Class:
   Name: Bank;
   Visibility: Public;
   Type: Class
Method:
   Name: Deposit;
   Visibility: Public;
   Virtual;
   Interop;
   IL;
   Managed;
   void Deposit(CardNo ACardNo,
                Password CusPass);
Parameter:
   Name: ACardNo;
   Order: 1;
   Attributes: In;
Parameter:
   Name: CusPass;
   Order: 2;
   Attributes: In;
          ⋮
```

IL Code

Bank Component

.NET: Example (cont'd)

The banking system is assembled from the ATM component and two instances of Bank 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;

IL Code

ATM Component

- Class:
  - Name: Bank;
  - Visibility: Public;
  - Type: Class
- Method:
  - Name: Deposit;
  - Visibility: Public;
  - Virtual;
  - Interop;
  - IL;
  - Managed;
  - void Deposit(CardNo ACardNo, Password CusPass);
- Parameter:
  - Name: ACardNo;
  - Order: 1;
  - Attributes: In;
- Parameter:
  - Name: CusPass;
  - Order: 2;
  - Attributes: In;

⋮

IL Code

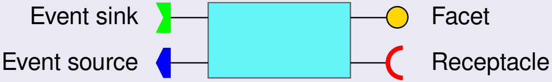
Bank Component

Taxonomy of Component Models: Category 2

CCM

CCM: Components

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.

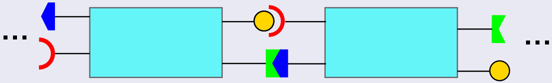


- A CORBA meta-type is an extension and specialisation of a CORBA Object [52, 16]
- Component interfaces are made up of ports: Facets (provided services), Receptacles (required services), Event Sources and Event Sinks.
- Component types are specific, named collections of features that can be described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

CCM: Composition

CCM components are assembled by method and event delegations in such a way that

- facets match receptacles
- event sources match event sinks

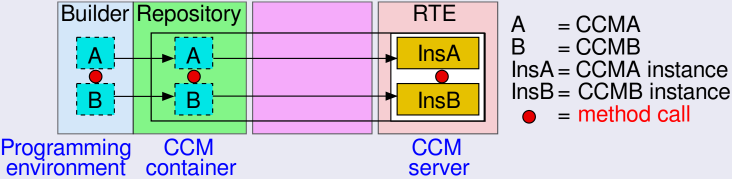


CCM

Support for Idealised Component and System Life Cycles

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



A = CCMA
B = CCMB
InsA = CCMA instance
InsB = CCMB instance
● = method call

CCM

Component and System Life Cycles

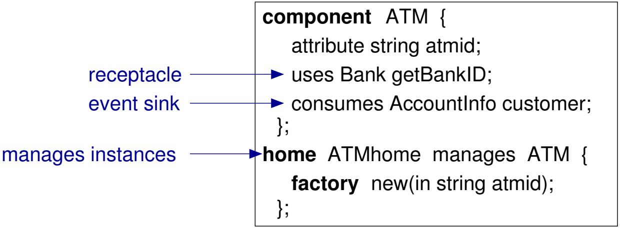
- Component life cycle coincides with system life cycle
- In Component/system design phase, CCM components are
  - designed and implemented
  - composed into a complete system
  - deposited in the CCM server
- Client applications make calls to CCM components in the system via the system's interface
- At run-time, client applications are executed, invoking CCM components in the system.

CCM: Example

Consider a simple bank system implemented by ATM, BankConsortium, Bank1 and Bank2 components (in OMG IDL 3):

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

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



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

CCM: Example (cont'd)

facet

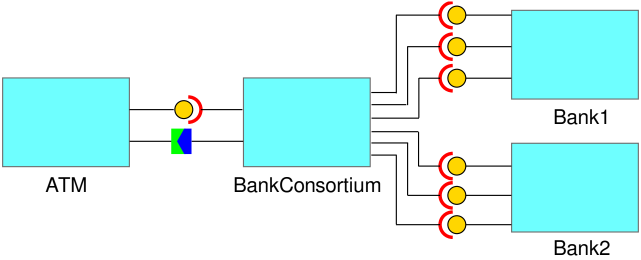
```
component  Bank  {
    attribute string bankid;
    provides Bank deposit;
    provides Bank withdraw;
    provides Bank checkBalance;
  };
home  Bankhome manages Bank {
    factory  new(in string bankid);
  };
```

event source

```
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);
  };
```

CCM: Example (cont'd)

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 version = "1.0"?>
<!DOCTYPE component assembly BANKSYSTEM "componentassembly.dtd">
<component assembly id = "banksys">
  <description> bank assembly descriptor</description>
  <componentfiles>
    <componnetfile id = "ATM component">
    <filearchive name = "ATM.csd">
    </componentfile>
    <componnetfile id = "BankConsortium component">
    <filearchive name = "BankConsortium.csd">
    </componentfile>
    <componnetfile id = "Bank component">
    <filearchive name = "Bank.csd">
    </componentfile>
  </componentfiles>
  <partitioning>
    <homereplacement id = "ATMHome">
      <componentfileref idref = "ATM Component"/>
      <componentinstantiation id = "atm">
      <registerwithnaming name = "ATMHome"/>
    </homereplacement>
    <homereplacement id = "BankConsortiumHome">
      <componentfileref idref = "BankConsortium Component"/>
      <componentinstantiation id = "bankconsortium">
      <registerwithnaming name = "BankConsortiumHome"/>
    </homereplacement>
    <homereplacement id = "BankHome">
      <componentfileref idref = "Bank Component"/>
      <componentinstantiation id = "bank1">
      <componentinstantiation id = "bank2">
      <registerwithnaming name = "BankHome"/>
    </homereplacement>
  </partitioning>
  <connections>
       ⋮
  </connections>
</component assembly>
```

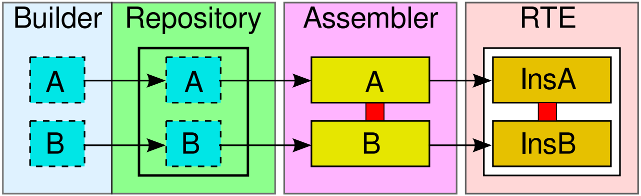
CCM: Example (cont'd)

```
<connections>
  <connectinterface>
    <usesport>
      <usesidentifier>getBankID</usesidentifier>
      <componentinstantiationref idref = "atm"/>
      <usesidentifier>deposit</usesidentifier>
      <usesidentifier>withdraw</usesidentifier>
      <usesidentifier>checkBalance</usesidentifier>
      <componentinstantiationref idref = "bankcon"/>
    </usesport>
    <providesport>
      <providesidentifier>getBankID</providesidentifier>
      <componentinstantiationref idref = "bankcon"/>
      <providesidentifier>deposit</providesidentifier>
      <providesidentifier>withdraw</providesidentifier>
      <providesidentifier>checkBalance</providesidentifier>
      <componentinstantiationref idref = "bank"/>
    </providesport>
  </connectinterface>
  <connectevent>
    <publishesport>
      <publishesidentifier>customer</publishesidentifier>
      <componentinstantiationref idref = "bankcon"/>
    </publishesport>
    <consumesport>
      <consumesidentifier>customer</consumesidentifier>
      <componentinstantiationref idref = "atm"/>
    </consumesport>
  </connectevent>
</connections>
```

Part III

- Taxonomy of component models: Categories 3,4 and 5

Taxonomy of Component Models: Category 3



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

Taxonomy of Component Models: Category 3

JavaBeans

JavaBeans: Components

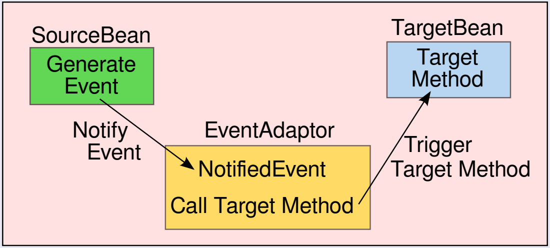
In JavaBeans [61, 39], a component is a bean, which is just any Java class that has:

- methods
- events
- properties

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

JavaBeans: Composition

In deployment phase, bean instances are composed via event delegation



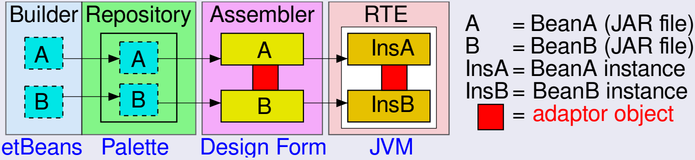
- a bean ‘composes’ with another bean by sending a message through delegation of events
- the bean builder tool automatically generates, compiles, and loads event adaptor classes for logistics of events

JavaBeans (NetBeans)

Support for Idealised Component and System Life Cycles

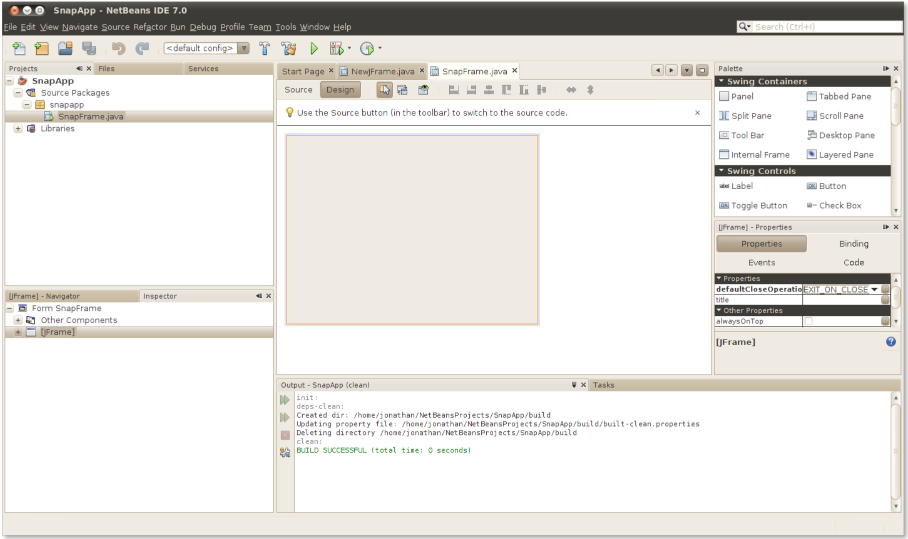
In NetBeans, individual beans are constructed as Java classes, and deposited in the Palette.
Bean instances are retrieved from the Palette into the Design Form and composed into a system.

- 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



A = BeanA (JAR file)
B = BeanB (JAR file)
InsA = BeanA instance
InsB = BeanB instance
🟥 = adaptor object

JavaBeans: NetBeans visual builder



Use the Source button (in the toolbar) to switch to the source code.

defaultCloseOperatio EXIT_ON_CLOSE

title

alwaysOnTop ☐

```
init:
deps-clean:
Created dir: /home/jonathan/NetBeansProjects/SnapApp/build
Updating property file: /home/jonathan/NetBeansProjects/SnapApp/build/built-clean.properties
Deleting directory /home/jonathan/NetBeansProjects/SnapApp/build
clean:
BUILD SUCCESSFUL (total time: 0 seconds)
```

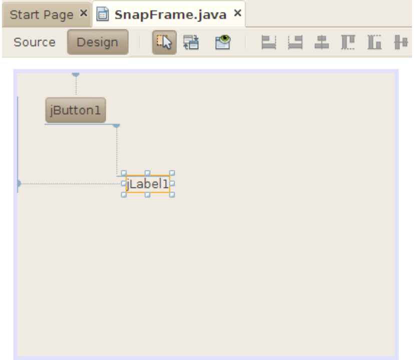
Picture taken from [39].

Javabeans

Component and System Life Cycles

- 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



You pressed the button!

- 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

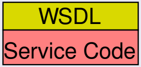
Pictures taken from [39].

Taxonomy of Component Models: Category 3

Web Services

Web Services: Components

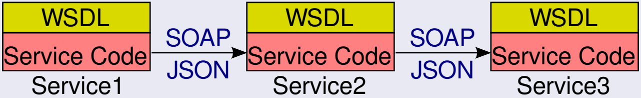
- Web services [9, 12, 5] are web application components that can be published, found, and used on the Web
- A web service contains:
  - an interface in WSDL (Web Service Description Language)
    - ⋆ describes the functionalities the web service provides
  - a binary implementation (the service code)



- Service clients communicate directly with service providers [12].

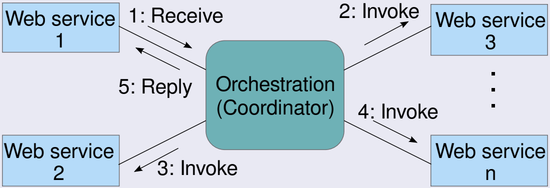
Web Services: Composition

- 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]

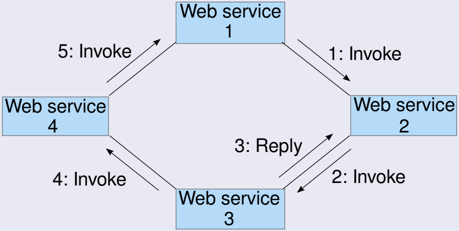


Web Services: Composition

Orchestration



Choreography



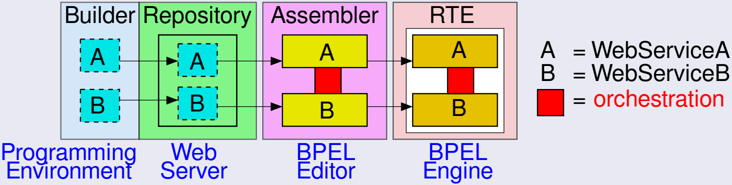
Pictures from: http://www.oracle.com/technetwork/articles/matjaz-bpel1-090575.html

Web Services

Support for Idealised Component and System Life Cycles

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



A = WebServiceA
B = WebServiceB
🟥 = orchestration

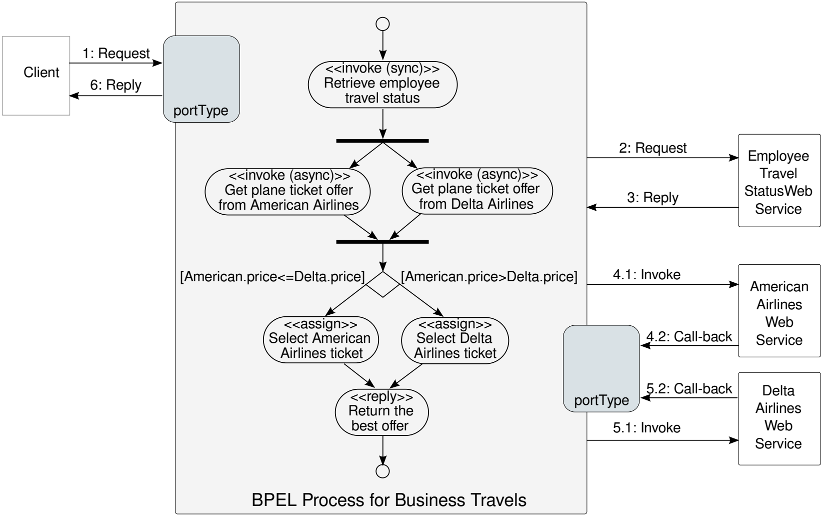
Web Services

Component and System Life Cycles

- 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

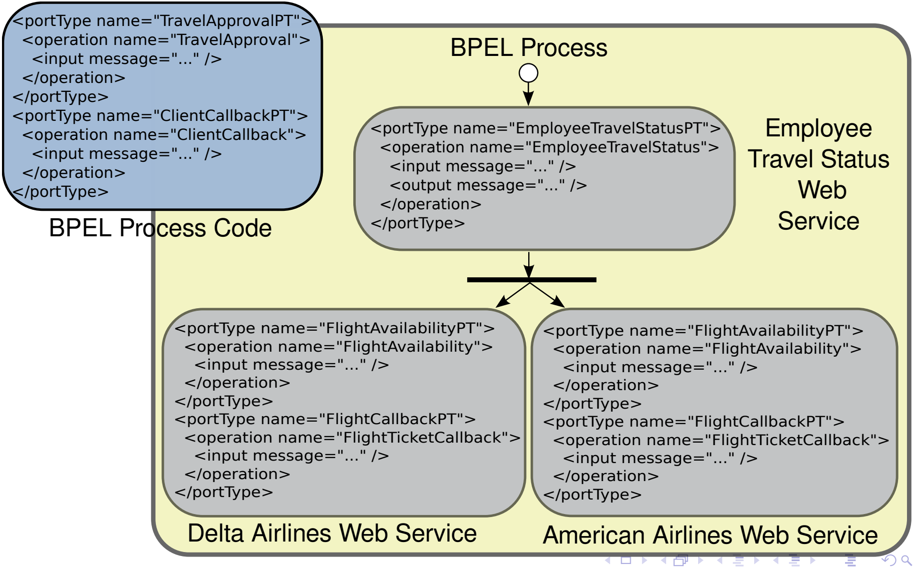
Composition by Orchestration



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

Web Services: Example (cont’d)

Composition by Orchestration



```
<portType name="TravelApprovalPT">
  <operation name="TravelApproval">
    <input message="..." />
  </operation>
</portType>
<portType name="ClientCallbackPT">
  <operation name="ClientCallback">
    <input message="..." />
  </operation>
</portType>
```

BPEL Process Code

BPEL Process

```
<portType name="EmployeeTravelStatusPT">
  <operation name="EmployeeTravelStatus">
    <input message="..." />
    <output message="..." />
  </operation>
</portType>
```

Employee Travel Status Web Service

```
<portType name="FlightAvailabilityPT">
  <operation name="FlightAvailability">
    <input message="..." />
  </operation>
</portType>
<portType name="FlightCallbackPT">
  <operation name="FlightTicketCallback">
    <input message="..." />
  </operation>
</portType>
```

Delta Airlines Web Service

```
<portType name="FlightAvailabilityPT">
  <operation name="FlightAvailability">
    <input message="..." />
  </operation>
</portType>
<portType name="FlightCallbackPT">
  <operation name="FlightTicketCallback">
    <input message="..." />
  </operation>
</portType>
```

American Airlines Web Service

Web Services: Example (cont'd)

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>
```

Web Services: Example (cont'd)

American Airlines and Delta Airlines Web Service

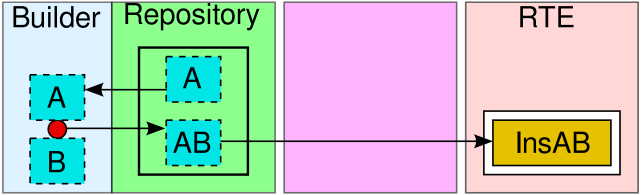
```
<message name="FlightTicketRequestMessage">
  <part name="flightData" type="tns:FlightRequestType" />
  <part name="travelClass" type="emp:TravelClassType" />
</message>
<message name="TravelResponseMessage">
  <part name="confirmationData" type="tns:FlightConfirmationType" />
</message>
<portType name="FlightAvailabilityPT">
  <operation name="FlightAvailability">
    <input message="tns:FlightTicketRequestMessage" />
  </operation>
</portType>
<portType name="FlightCallbackPT">
  <operation name="FlightTicketCallback">
    <input message="tns:TravelResponseMessage" />
  </operation>
</portType>
```

Web Services: Example (cont’d)

BPEL Process for Business Travels

```
<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



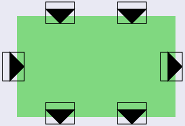
Category 4: Design with Repository
(Koala, SOFA, KobrA, SCA, Palladio, ProCom)

Taxonomy of Component Models: Category 4

Koala

Koala: Components

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).

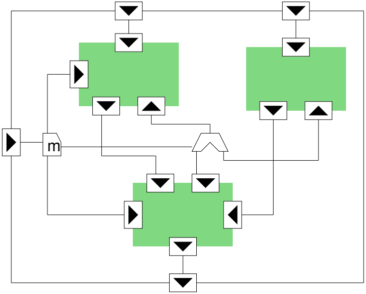
Components are definition files only (no implementation).

---

[2] C[K]omponent Organizer And Linking Assistant

Koala: Composition

Koala components are composed by method calls through connectors.

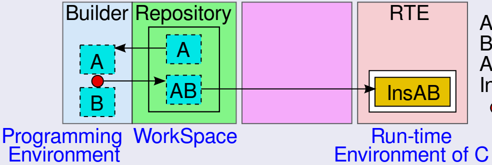


Koala

Support for Idealised Component and System Life Cycles

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



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

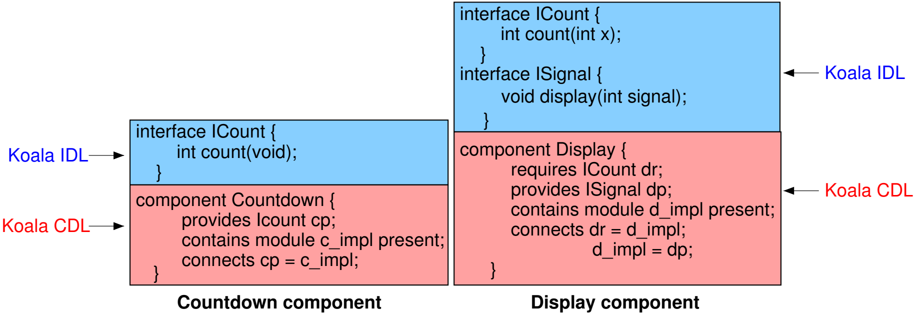
Koala

Component and System Life Cycles

- 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

Koala: Example

Consider a Stopwatch device that comprises a Countdown component and a Display component.



Koala IDL

```
interface ICount {
        int count(void);
    }
```

Koala CDL

```
component Countdown {
        provides Icount cp;
        contains module c_impl present;
        connects cp = c_impl;
  }
```

Countdown component

Koala IDL

```
interface ICount {
        int count(int x);
   }
interface ISignal {
        void display(int signal);
    }
```

Koala CDL

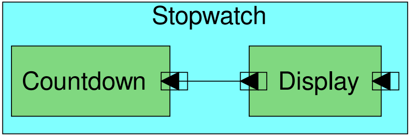
```
component Display {
        requires ICount dr;
        provides ISignal dp;
        contains module d_impl present;
        connects dr = d_impl;
                 d_impl = dp;
    }
```

Display component

- The interfaces are specified in Koala IDL
- The component definitions are in Koala CDL

Koala: Example (cont'd)

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

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

Koala: Example (cont'd)

The definition files of Stopwatch, Countdown and Display are compiled by the Koala compiler to C header files.

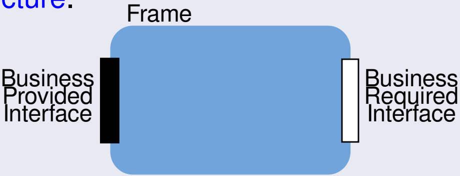
Then the programmer has to

- write C files (to implement the components)
- compile these witTaxonomy of Component Models: Category 4h the header files to binary C code for Stopwatch.

Taxonomy of Component Models: Category 4

SOFA

In SOFA[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.



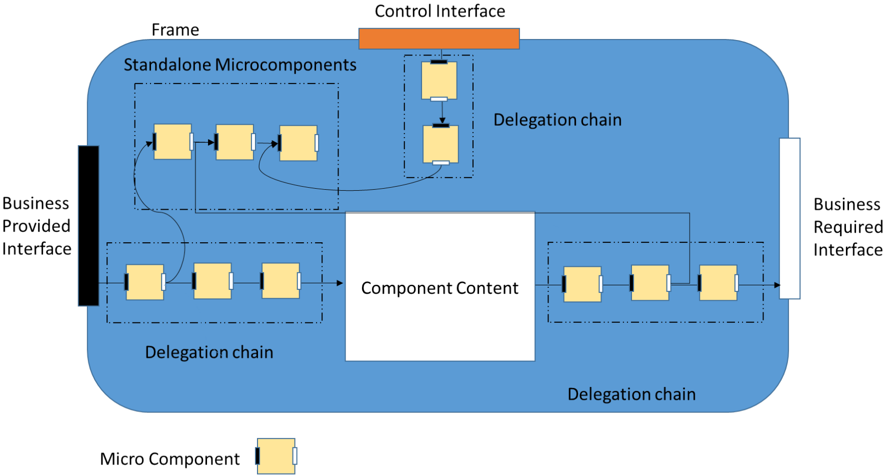
- The frame defines provided and required interfaces, and properties of the component
- The architecture describes the structure of the component

---

[3] SOFtware Appliances

SOFA 2: Components

Including Run-time Control Interface and Microcomponents



SOFA: Composition

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

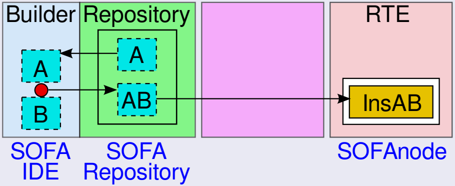
- procedure call: classic client server call.
- messaging: asynchronous message delivery from a producer to subscribed listeners.
- streaming: uni- or bidirectional stream of data between a sender and (multiple) recipients.
- blackboard: communication via shared memory.

SOFA

Support for Idealised Component and System Life Cycles

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.



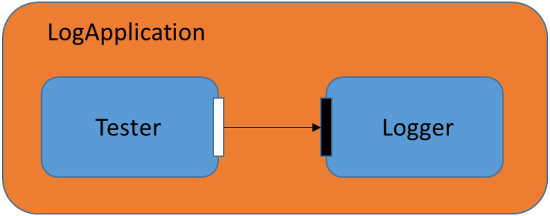
A = SOFAA
B = SOFAB
AB = SOFAAB
InsAB = SOFAAB instance
● = connector

SOFA

Component and System Life Cycles

- 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`.

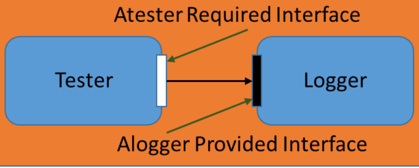
SOFA: Example (cont'd)

FRAME

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

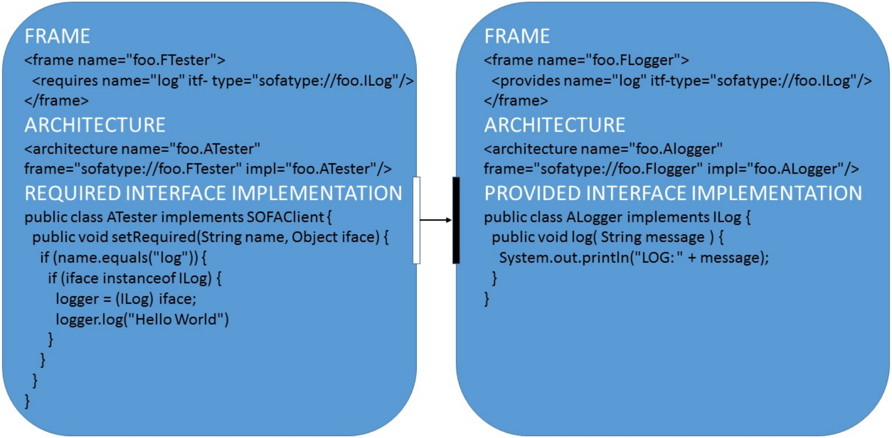
ARCHITECTURE

```
<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>
```



LogApplication

SOFA: Example (cont'd)



FRAME

```
<frame name="foo.FTester">
 <requires name="log" itf- 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")
   }
  }
 }
}
```

Tester

FRAME

```
<frame name="foo.FLogger">
 <provides name="log" itf-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);
 }
}
```

Logger

Taxonomy of Component Models: Category 4

KobrA

KobrA: Components

In KobrA[4] [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.

---

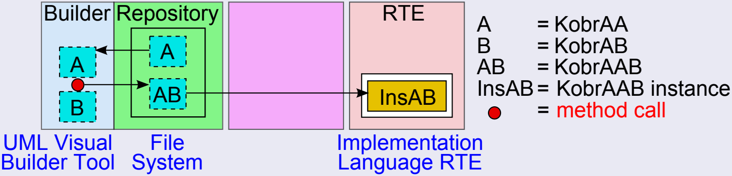
[4] Komponenten-basierte Anwendungsentwicklung (component-based application development)

KobrA

Support for Idealised Component and System Life Cycles

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



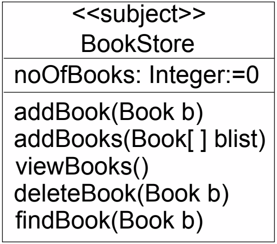
KobrA

Component and System Life Cycles

- Component life cycle is separate form 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

KobrA: Example

Consider a book store that maintains a database of its book stock and sells its books by an Automatic Teller Machine (ATM).



<<subject>>
BookStore

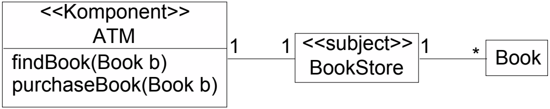
noOfBooks: Integer:=0

addBook(Book b)
addBooks(Book[ ] blist)
viewBooks()
deleteBook(Book b)
findBook(Book b)

The specification of the BookStore component is a UML class diagram that specifies what the BookStore component does.

KobrA: Example (cont'd)

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.



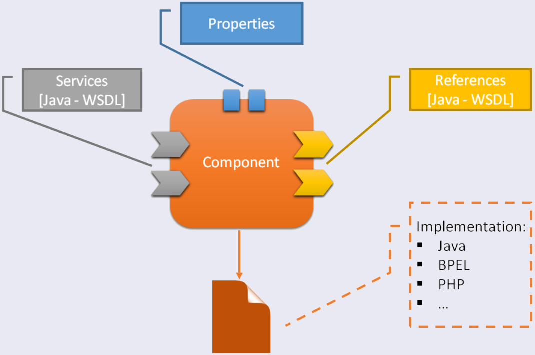
The book store system is assembled from the ATM, BookStore and Book components by direct method calls.

Taxonomy of Component Models: Category 4

SCA

SCA: Components

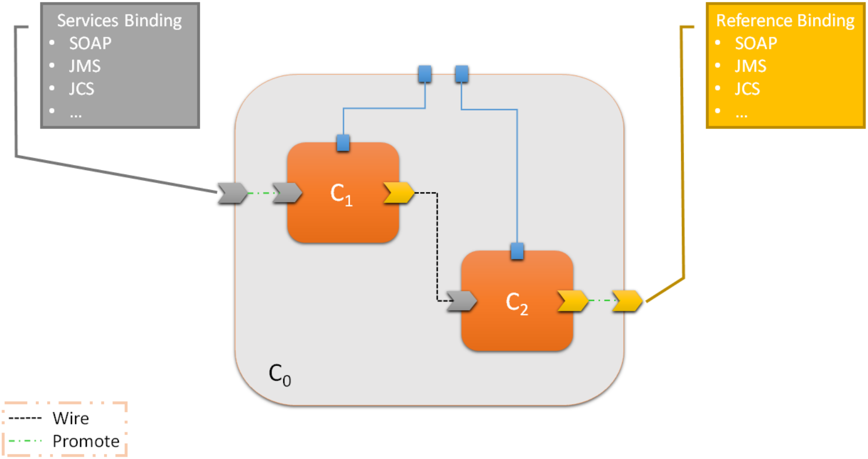
In SCA[5] [10, 38], a component has services, references and properties.



---

[5] Service Component Architecture

SCA: Composition



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

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

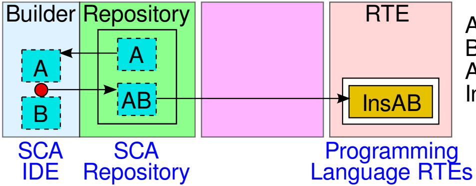
------ Wire
-·-·-· Promote

SCA

Support for Idealised Component and System Life Cycles

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



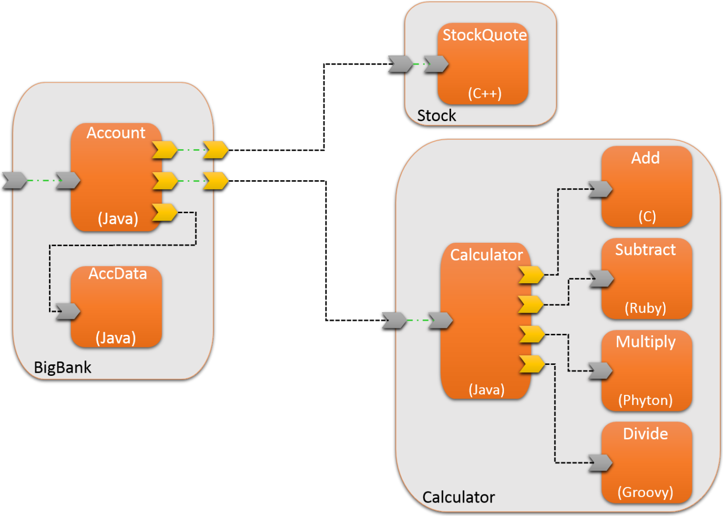
A = SCAA
B = SCAB
AB = SCAAB
InsAB = SCAAB instance
● = connector

SCA

Component and System Life Cycles

- 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



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

Taxonomy of Component Models: Category 4

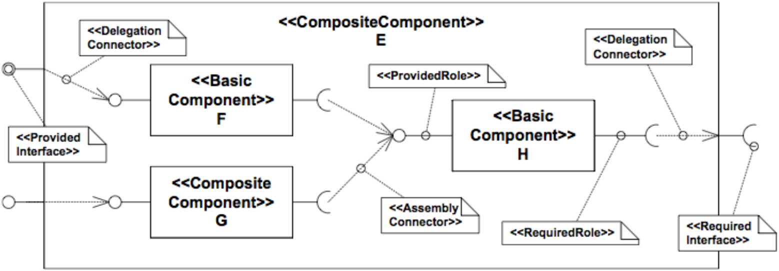
Palladio

Palladio: Components

- 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

Palladio: Composition

- Composition is port connection via connectors
- Connectors can be assembly or delegation

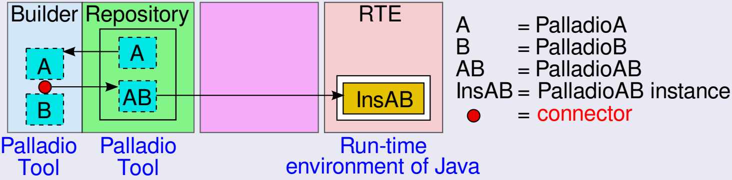


Picture taken from [57].

Palladio

Support for Idealised Component and System Life Cycles

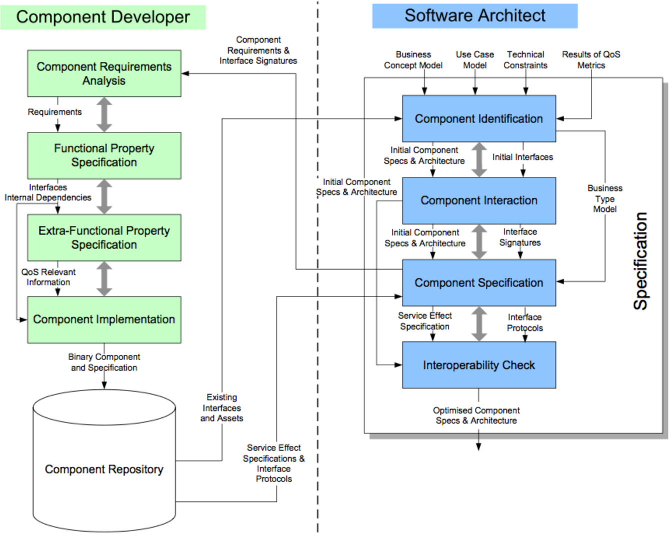
- In design phase, (basic and composite) components are abstractly or concretely defined, assembled, and stored in repository. The builder is the PCM tool.
- Also in design phase, components are chosen and assembled into systems.
- System code skeleton is generated and then implemented using an implementation language such as Java.



A = PalladioA
B = PalladioB
AB = PalladioAB
InsAB = PalladioAB instance
● = connector

Palladio

Component and System Life Cycles



Picture taken from [57].

Palladio

Component and System Life Cycles (cont'd)

- 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.

Palladio: Example

Consider a simple ATM system that can read customers' bank cards and provide basic services:

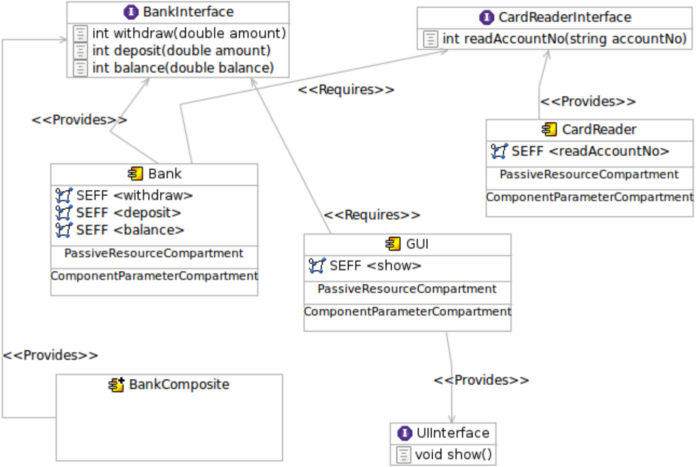
- withdraw
- deposit
- check balance

We identify three atomic components:

- CardReader
- Bank
- GUI

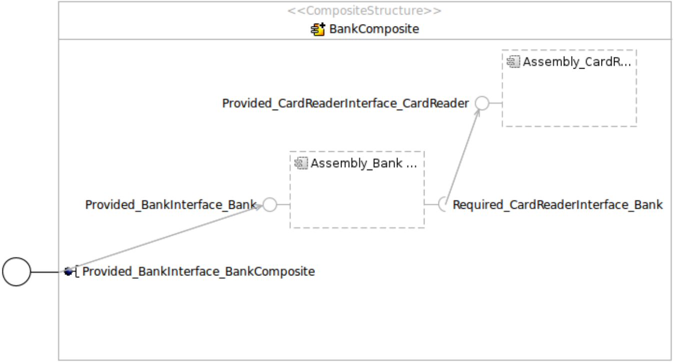
Palladio: Example (cont'd)

- In design phase, we design the 3 identified components.
- We also build a composite component BankComposite from the atomic ones.



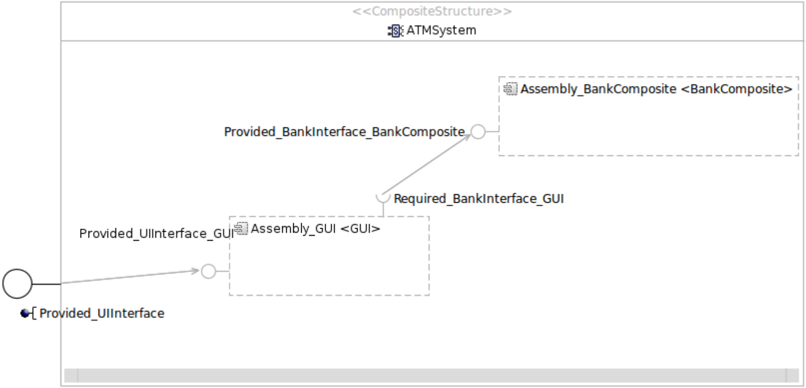
Palladio: Example (cont'd)

- The composite component BankComposite is built by assembling CardReader and Bank.



Palladio: Example (cont'd)

- To construct the system, we assemble BankComposite and GUI.



Taxonomy of Component Models: Category 4

ProCom

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

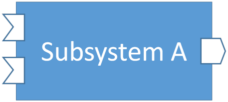
Connection between the two layers

A subsystem component can internally be modelled by ProSave components

ProSys

“Subsystem” Components

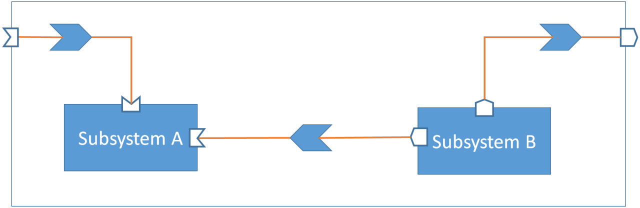
An atomic subsystem:



Typed Input message port

Typed Output message port

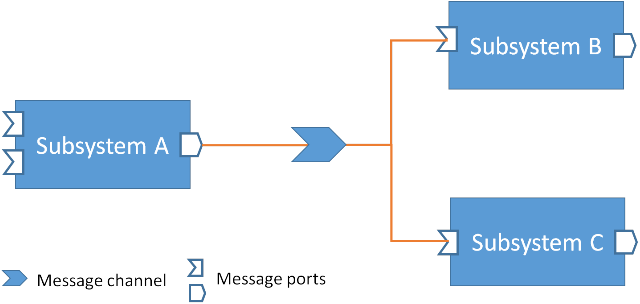
A composite subsystem:



Message channel

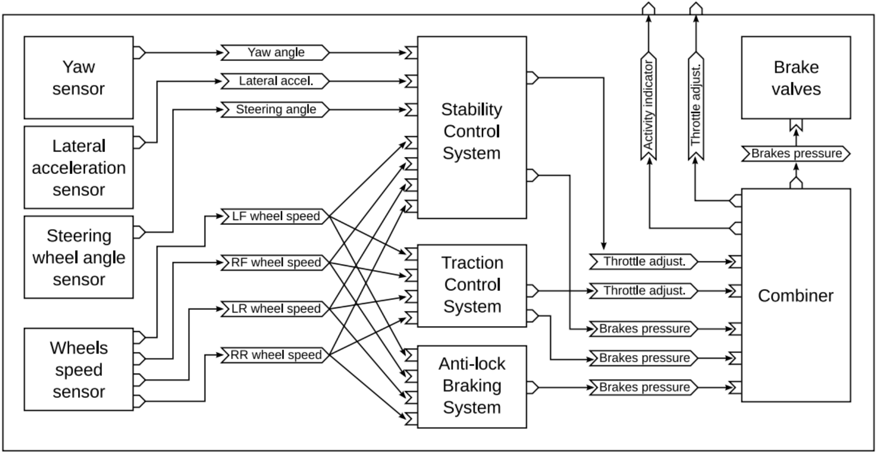
ProSys: Composition

- Message ports not directly connected
- Composition via explicit message channels



ProSys: Example

An Electronic Stability Control (ESC) system:



Picture taken from [58].

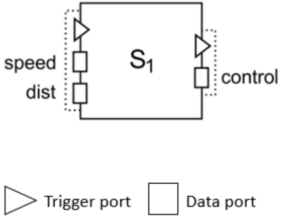
ProSave

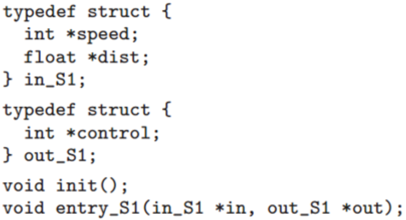
“Function” Components

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:



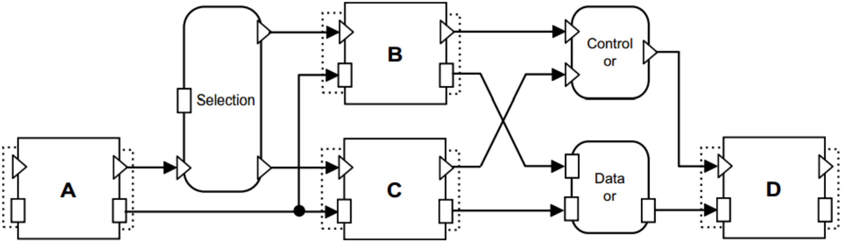


```
typedef struct {
  int *speed;
  float *dist;
} in_S1;
typedef struct {
  int *control;
} out_S1;
void init();
void entry_S1(in_S1 *in, out_S1 *out);
```

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*

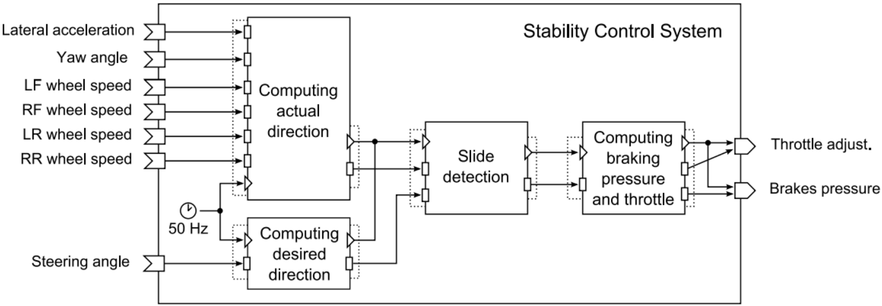
A typical usage of selection and or-connectors:



Picture taken from [23].

ProSave: Example

The Electronic Stability Control System:



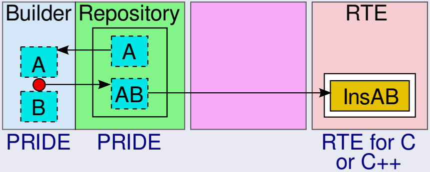
Picture taken from [58].

ProCoM

Support for Idealised Component and System Life Cycles

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++.



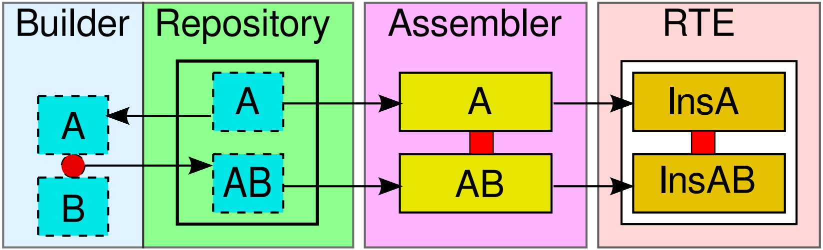
A = PROCOMA
B = PROCOMB
AB = PROCOMAB
InsAB = PROCOMAB instance
● = connector

ProCoM

Component and System Life Cycles

- 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



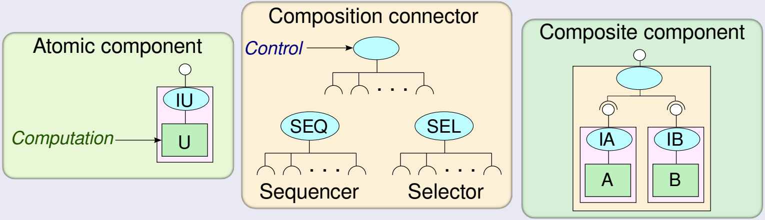
Category 5: Design and Deploy with Repository
(X-MAN)

Taxonomy of Component Models: Category 5

X-MAN

X-MAN: Components

In X-MAN [46, 45, 36, 42], components encapsulated units of computation, with only provided services.

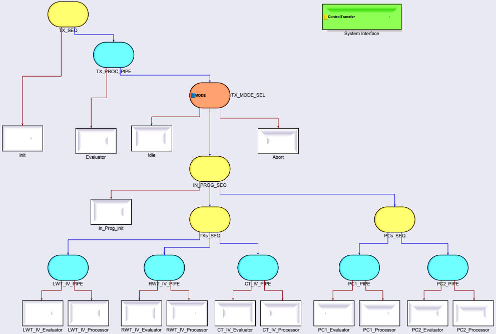


- an atomic component contains an invocation connector (IU) and a computation unit (U); the invocation connector, when activated by control coming from a composition connector, invokes methods provided by the computation unit
- a composite component contains sub-components composed by composition connectors; composite components are self-similar

X-MAN: Composition

Components are composed by composition connectors, which

- encapsulate control
- coordinate control flow between components.



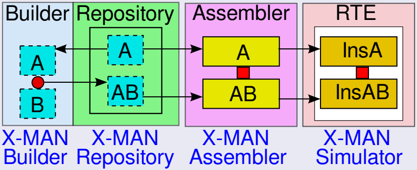
X-MAN

Support for Idealised Component and System Life Cycles

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.



A = XMANA
B = XMANB
AB = XMANAB
InsAB = XMANAB instance
● = design phase
🟥 = deployment phase
} composition connector

X-MAN

Component and System Life Cycles

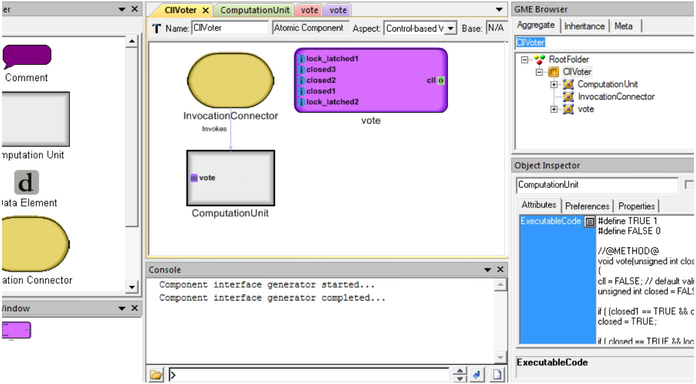
- Component life cycle is separate from system life cycle
- In component design phase, X-MAN components (both atomic and composite) are defined and constructed and deposited in the repository of the X-MAN tool
- In system design/component deployment phase, X-MAN components are retrieved from the repository 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

X-MAN: Example

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.

X-MAN: Example (cont’d)

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++):

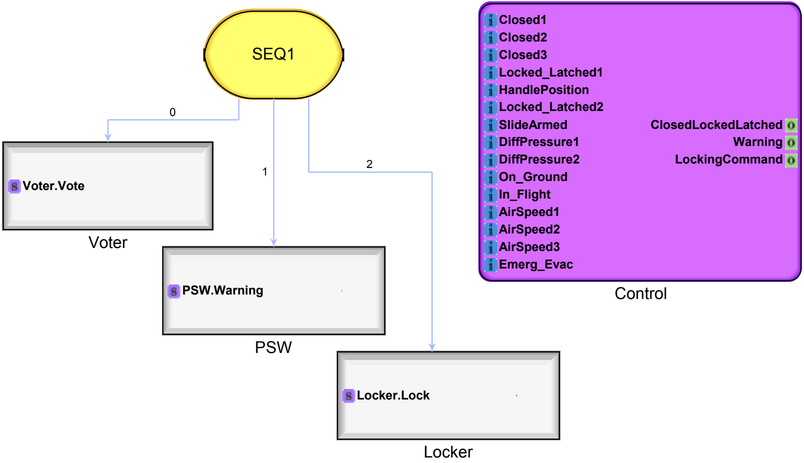


```
Component interface generator started...
Component interface generator completed...
```

```
#define TRUE 1
#define FALSE 0

//@METHOD@
void vote(unsigned int clos
{
cll = FALSE; // default val
unsigned int closed = FALS

if ( (closed1 == TRUE && c
closed = TRUE;

if ( closed == TRUE && loc
```

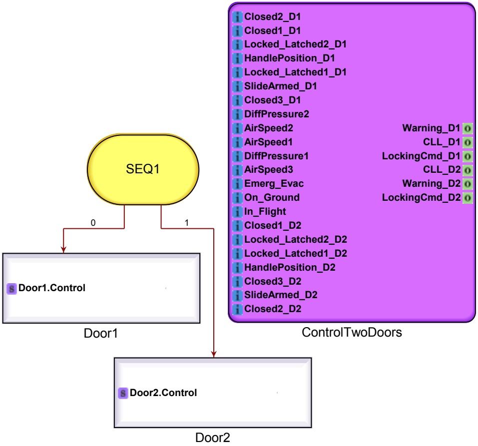
X-MAN: Example (cont'd)

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:



X-MAN: Example (cont’d)

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

Future Challenges and New Desiderata

Well-known benefits of CBD

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

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

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

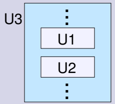
Towards Increased Scale, Complexity and Safety

Additional Desiderata for Composition

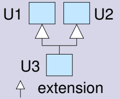
- hierarchical (algebraic) composition mechanisms
- (algebraic) composition operators

Existing Software Composition Mechanisms

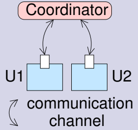
Containment



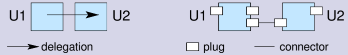
Extension



Coordination



Connection



(a) Direct message passing

(b) Indirect message passing

A Taxonomy of Software Composition Mechanisms

	Unit of Composition	Composition Mechanism				
		Containment	Extension	Connection	Coordination	
Programming View	Function	Function nesting		Higher-order function Function call		
	Procedure	Procedure nesting		Procedure call		
	Class	Class nesting Object composition Object aggregation	Multiple inheritance	Object delegation		
	Mixin		Mixin inheritance			
	Mixin/Class		Mixin-class inheritance			
	Trait		Trait composition	Trait composition		
	Trait/Class		Trait-class composition	Trait-class composition		
	Subject		Subject composition			
	Feature		Feature composition			
	Aspect/Class		Weaving			
	Module	Module nesting		Module connection		Construction View
CBD View	Architectural unit			Port connection		
	Fragment box		Invasive composition	Invasive composition		
	Process			Channels	Data coordination	
	Web service				Orchestration (Control coordination)	
	Encapsulated component				Exogenous composition (Control coordination)	

[43] K.-K. Lau and T. Rana, A Taxonomy of Software Composition Mechanisms, Proc. 36th EUROMICRO Conference on Software Engineering and Advanced Applications, pages 102–110, 2010, IEEE.

Algebraic Software Composition Mechanisms

Composition Mechanism				Algebraic ?
Containment	Extension	Connection	Coordination	
	Mixin-class inheritance	Function call	Data coordination	No
		Procedure call		
	Trait-class composition	Module connection	Orchestration	
		Object delegation		
	Weaving	Trait-class composition		
Function nesting	Multiple inheritance	Higher-order function	Exogenous composition	Yes
Procedure nesting	Mixin inheritance	Trait composition		
Module nesting	Trait composition	Port connection		
Class nesting	Subject composition	Invasive composition		
Object composition	Feature composition	Channels		
Object aggregation	Invasive composition			

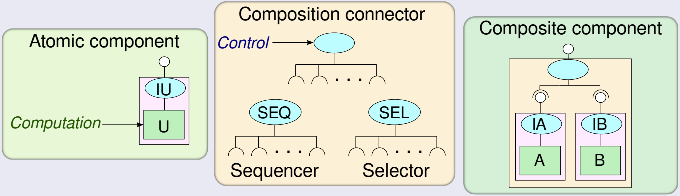
Algebraic Composition Mechanism				Composition operator ?
Containment	Extension	Connection	Coordination	
	Mixin inheritance	Higher-order function	Exogenous composition	Yes
	Subject composition			
Function nesting Procedure nesting	Multiple inheritance	Trait composition		No
Module nesting	Trait composition	Port connection		
Class nesting	Feature composition	Invasive composition		
Object composition Object aggregtion	Invasive composition	Channels		

[43] K.-K. Lau and T. Rana, A Taxonomy of Software Composition Mechanisms, Proc. 36th EUROMICRO Conference on Software Engineering and Advanced Applications, pages 102–110, IEEE, 2010.

Compositional Construction

The X-MAN Component Model

X-MAN: Encapsulated Components + Exogenous Composition



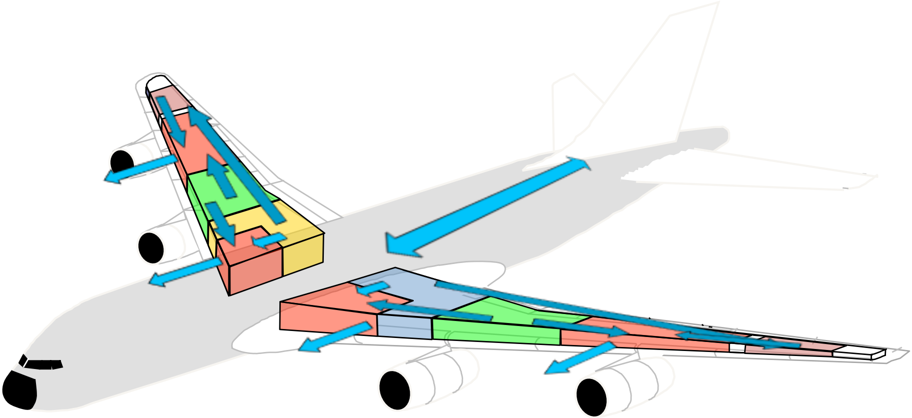
Projects (European Artemis JU)

- 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)

[42] K.-K. Lau, M. Pantel, D. Chen, M. Persson, M. Törngren and C. Tran, Component-based Development, in A. Rajan and T. Wahl, editors, *CESAR – Cost-efficient Methods and Processes for Safety-relevant Embedded Systems*, Chapter 5, pages 179-212, Springer-Verlag Wien, 2013.

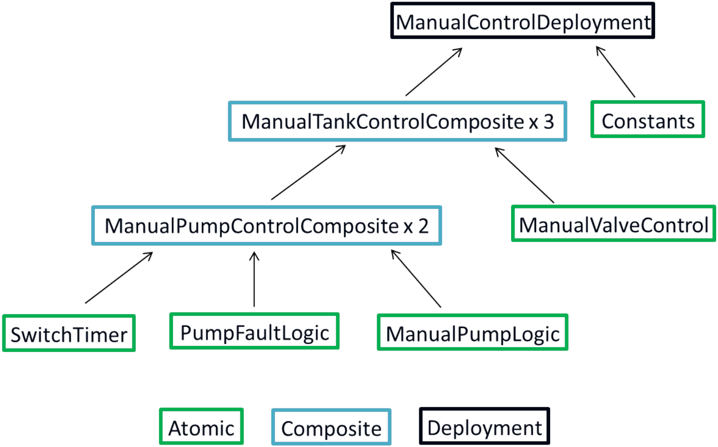
Compositional Construction in X-MAN

CESAR Project: Aircraft Fuel System



Compositional Construction in X-MAN

Aircraft Fuel System: Component-based Design



ManualControlDeployment

ManualTankControlComposite x 3

Constants

ManualPumpControlComposite x 2

ManualValveControl

SwitchTimer

PumpFaultLogic

ManualPumpLogic

Atomic

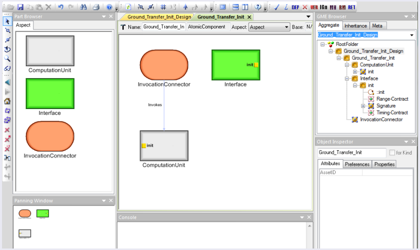
Composite

Deployment

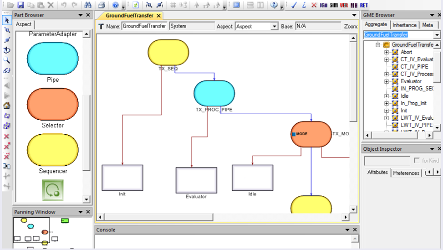
Compositional Construction in X-MAN

Aircraft Fuel System: Composition in Two Phases

Component Design



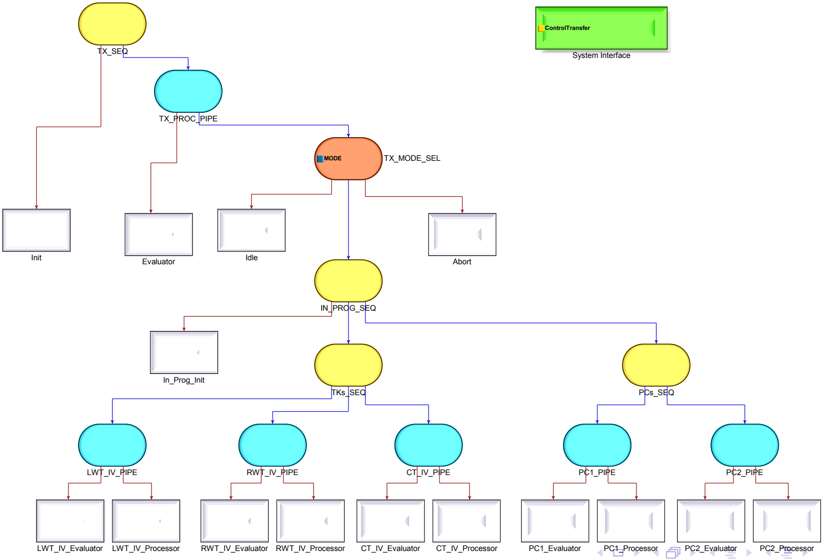
Component Deployment



[36] N. He, D. Kroening, T. Wahl, K.-K. Lau, F. Taweel, C. Tran, P. Rümmer and S. Sharma, Component-based Design and Verification in X-MAN, in Proc. Embedded Real Time Software and Systems, 2012.

Compositional Construction in X-MAN

Aircraft Fuel System: Hierarchical (Algebraic) Composition



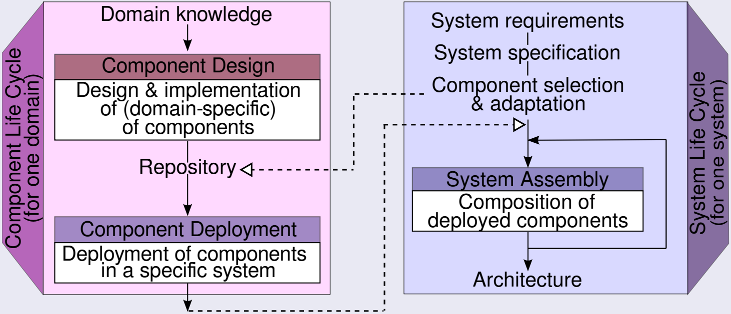
Compositional V&V

From Compositional Construction to Compositional V&V

Compositional V&V must be based on:

- compositional construction with
- separate component and system life cycles

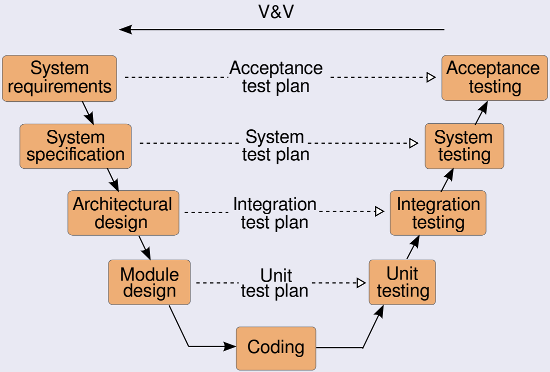
Component and System Life Cycles



Compositional V & V

Need to adapt the V model accordingly.

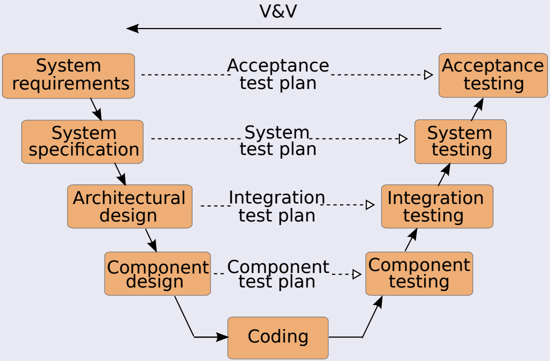
The V Model: Modular System Development



Compositional V & V

The straightforward adaptation does not work.

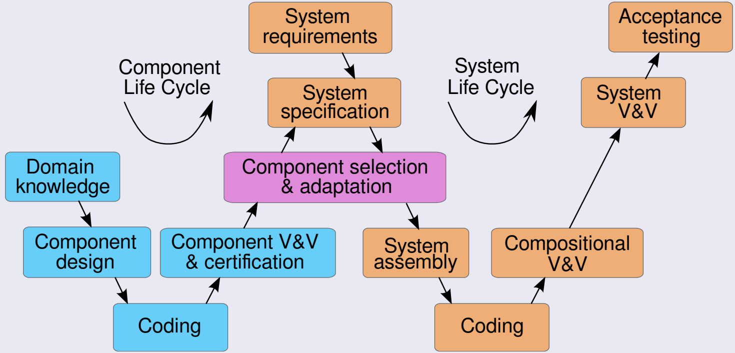
The V Model: Component-based System Development?



Compositional V & V

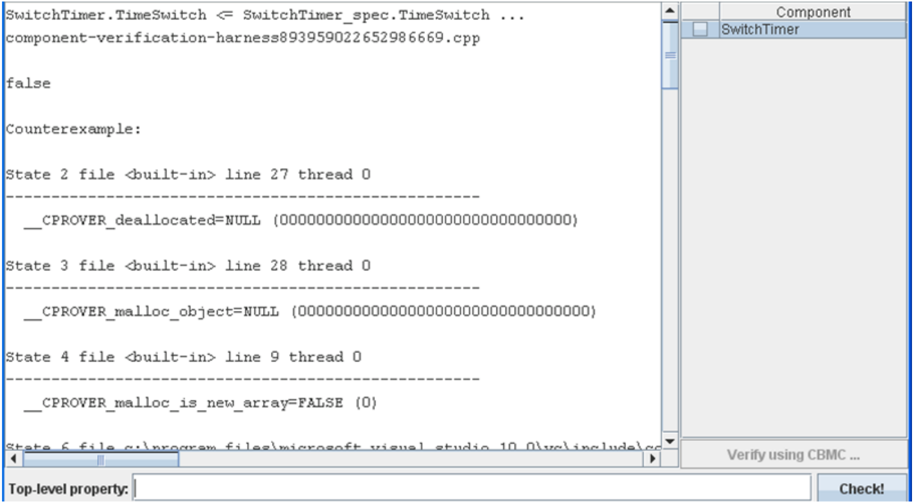
Need one V for each life cycle.

The W Model



Compositional V & V

Aircraft Fuel System: X-MAN Model Checker

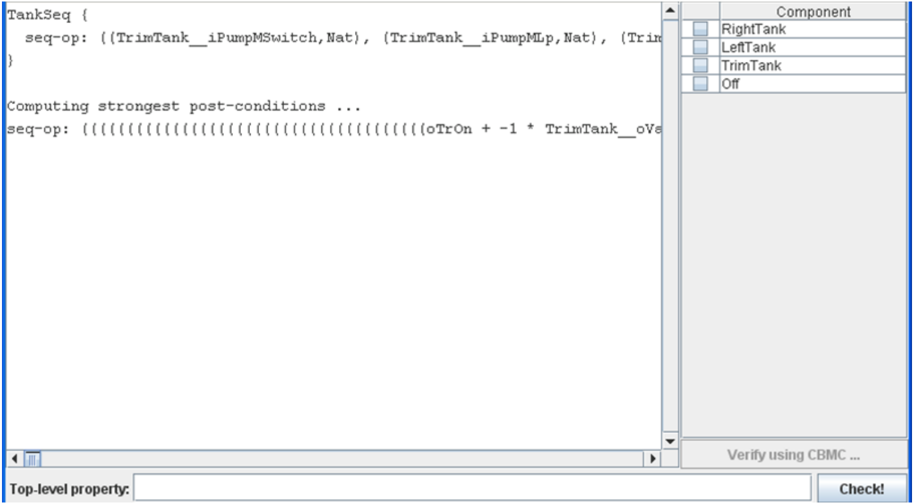


```
SwitchTimer.TimeSwitch <= SwitchTimer_spec.TimeSwitch ...
component-verification-harness8939590226529866669.cpp

false

Counterexample:

State 2 file <built-in> line 27 thread 0
----------------------------------------------------
  __CPROVER_deallocated=NULL (00000000000000000000000000000000)

State 3 file <built-in> line 28 thread 0
----------------------------------------------------
  __CPROVER_malloc_object=NULL (00000000000000000000000000000000)

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\cc
```

[36] N. He, D. Kroening, T. Wahl, K.-K. Lau, F. Taweel, C. Tran, P. Rümmer and S. Sharma, Component-based Design and Verification in X-MAN, in Proc. Embedded Real Time Software and Systems, 2012.

Compositional V & V

Aircraft Fuel System: X-MAN Theorem Prover

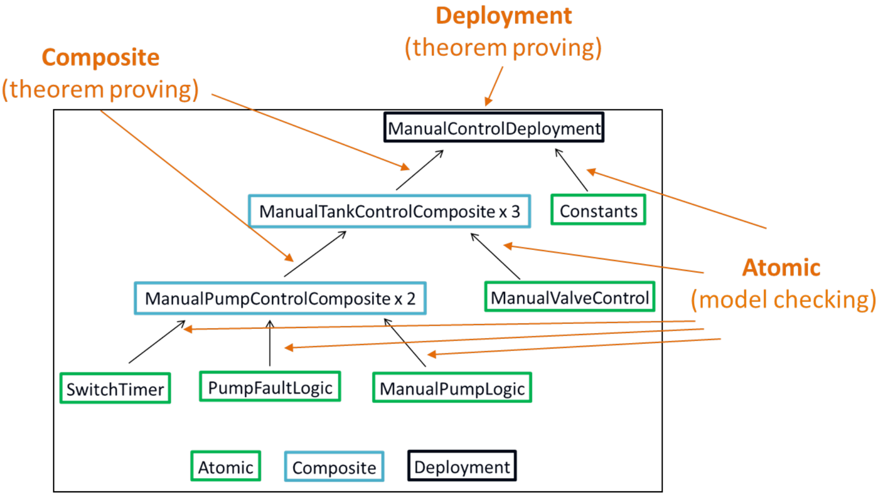


```
TankSeq {
  seq-op: ((TrimTank__iPumpMSwitch,Nat), (TrimTank__iPumpMLp,Nat), (Trim
}


Computing strongest post-conditions ...
seq-op: ((((((((((((((((((((((((((((((((((((oTrOn + -1 * TrimTank__oVs
```

	Component
☐	RightTank
☐	LeftTank
☐	TrimTank
☐	Off

Compositional V & V

Aircraft Fuel System: Proving at Multiple Levels



Compositional V & V

Aircraft Fuel System: Proving at Atomic and Composite Levels

Atomic level:

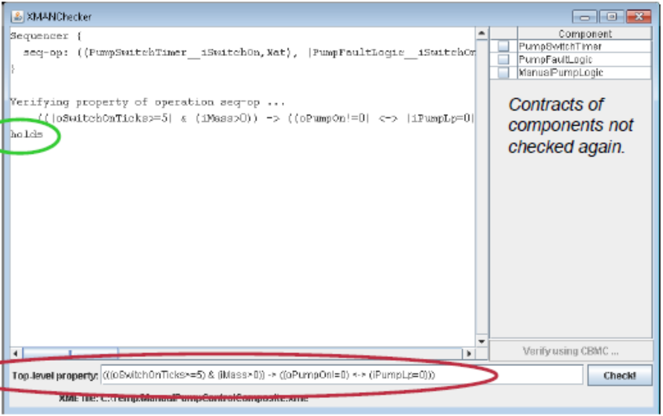
```
// Pump always on if switch is on and pump is not faulty
\exists nat FALSE;(
   (FALSE=0) &
   ( ((iSwitchOn!=FALSE) & (iPumpFaulty=FALSE) &
(iMass>0))
    <->
   (oOn!=FALSE) )
)
```

Composite level:

```
// Pump switch timer greater than zero if-and-only-if switch is on
((oSwitchOnTicks>0) <-> (iSwitchOn!=0))
&
// Pump always off if switches off
((oSwitchOnTicks=0) -> (oPumpOn=0))
&
// Pump always off if zero mass in tank
((iMass=0) -> (oPumpOn=0))
&
// Pump always off if on for 5 ticks or more and still LP detected
(((oSwitchOnTicks>=5) & (iPumpLp!=0)) -> (oPumpOn=0))
&
// Pump always on if mass in tank and switch on, but less than 5 ticks
(((iMass>0) & (oSwitchOnTicks>0) & (oSwitchOnTicks<5)) -> (oPumpOn!=0))
&
// If switch on for more than 5 ticks and mass in tank, pump on iff pump pressure
(((oSwitchOnTicks>=5) & (iMass>0)) -> ((oPumpOn!=0) <-> (iPumpLp=0)))
```

Compositional V & V

Aircraft Fuel System: Top-level Proof



```
Sequencer {
  seq-op: ((PumpSwitchTimer__iSwitchOn,Nat), |PumpFaultLogic__iSwitchOn
}

Verifying property of operation seq-op ...
     ((|oSwitchOnTicks>=5| & (iMass>0)) -> ((oPumpOn!=0| <-> |iPumpLp=0|
holds
```

Top-level property: (((oSwitchOnTicks>=5) & (iMass>0)) -> ((oPumpOn!=0) <-> (iPumpLp=0)))

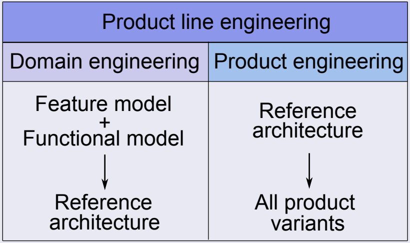
XML file: C:\Temp\ManualPumpControlComposite.xml

Contracts of components not checked again.

Compositional Product Line Engineering?

Current PLE practice

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



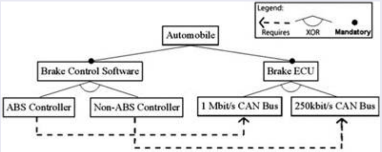
Product line engineering	
Domain engineering	Product engineering
Feature model + Functional model ↓ Reference architecture	Reference architecture ↓ All product variants

Compositional Product Line Engineering?

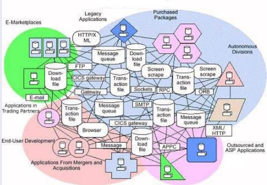
For scalability

Use tree-like product architectures and hence reference architecture ?

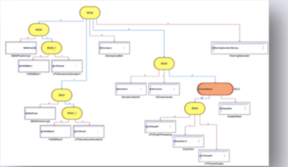
Feature Model Tree



‘Spaghetti’ Products

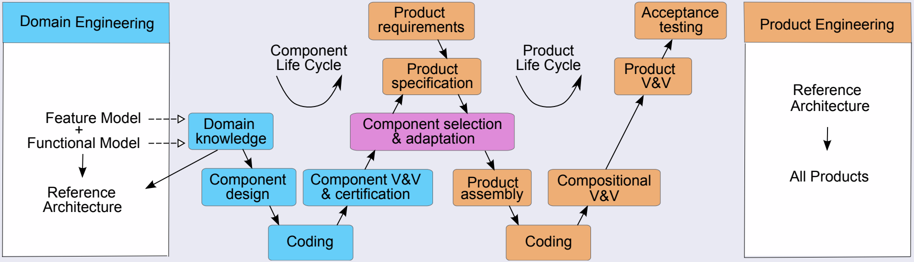


‘Tree’ Products



Compositional PLE with V & V

PLE with the W Model



Domain Engineering

Feature Model
+
Functional Model

Reference
Architecture

Domain knowledge

Component design

Coding

Component V&V & certification

Component Life Cycle

Component selection & adaptation

Product requirements

Product specification

Product assembly

Coding

Compositional V&V

Product Life Cycle

Product V&V

Acceptance testing

Product Engineering

Reference
Architecture

All Products

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 ?

[1] Darwin: An Architecure Description Language.
`http://www-dse.doc.ic.ac.uk/Research/Darwin/darwin.html`.

[2] Microsoft .NET Homepage.
`http://www.microsoft.com/net/`.

[3] The UniCon Architecture Description Language.
`http://www.cs.cmu.edu/~Vit/unicon/referencemanual/Reference_Manual_2.html`.

[4] The Wright Architecture Description Language.
`http://www.cs.cmu.edu/~able/wright/`.

[5] Web services tutorial.
`http://www.w3schools.com/Webservices/default.asp`.
Accessed: 2014-06-08.

[6] Common object request broker architecture: Core specification, March 2004.

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

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

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

[10] Apache Tuscany SCA web page.
`http://tuscany.apache.org/sca-overview.html`.

[11] C. Atkinson, J. Bayer, C. Bunse, E. Kamsties, O. Laitenberger, R. Laqua, D. Muthig, B. Paech, J. Wüst, and J. Zettel.
Component-based Product Line Engineering with UML.
Addison-Wesley, 2001.

[12] Douglas K. Barry.
Web Services, Service-Oriented Architectures, and Cloud Computing, Second Edition: The Savvy Manager's Guide.
Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2nd edition, 2013.

[13] D. Bartlett.
Corba component model(ccm): Introducing next-generation corba, 2001.

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

[15] S. Becker, H. Koziolek, and R. Reussner.
The Palladio component model for model-driven performance prediction.
J. Syst. Softw., 82(1):3–22, January 2009.

[16] F. Bolton.
Pure Corba.
SAMS, 2001.

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

[18] M. Broy, A. Deimel, J. Henn, K. Koskimies, F. Plasil, G. Pomberger, W. Pree, M. Stal, and C. Szyperski.
What characterizes a software component?
Software – Concepts and Tools, 19(1):49–56, 1998.

[19] E. Bruneton, T. Coupaye, and M. Leclercq.
An open component model and its support in Java.
In *Proc. 7th Int. Symp. on Component-based Software Engineering, LNCS 3054*, pages 7–22. Springer -Verlag, 2004.

[20] Eric Bruneton, Thierry Coupaye, Matthieu Leclercq, Vivien Quma, and Jean-Bernard Stefani.
The fractal component model and its support in java.
Software: Practice and Experience, 36(11-12):1257–1284, 2006.

[21] T. Bures, P. Hnetynka, and F. Plasil.
SOFA 2.0: Balancing Advanced Features in a Hierarchical Component Model.
In *Proc. SERA 2006*, pages 40–48. IEEE, 2006.

[22] T Bures and F. Plasil.
Communication style driven connector configurations.
In *Proc. SERA 2004*, pages 102–106. Springer, 2004.

[23] T. Bures *et al.*

ProCom - the Progress Component Model Reference Manual, 2008.

[24] L.F. Capretz.
Y: A new component-based software lifecycle model.
Journal of Computer Science, 1(1):76–82, 2005.

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

[26] 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.

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

[28] I. Crnkovic, M. Chaudron, and S. Larsson.
Component-based development process and component lifecycle.
In *Proc. Int. Conf. on Software Engineering Advances*, pages 44–53, 2006.

[29] I. Crnkovic, S. Sentilles, A. Vulgarakis, and M.R.V. Chaudron.
A classification framework for software component models.
IEEE Transactions on Software Engineering, 37(5):593–615, October 2011.

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

[31] M. Fortes da Cruz and P. Raistrick.
AMBERS: Improving Requirements Specification Through Assertive Models and SCADE/DOORS Integration.
In F. Redmill and T. Anderson, editors, *The Safety of Systems, Proc. 15th Safety-critical Systems Symposium*, pages 217–241, Bristol, UK, February 2007. Springer London.

[32] The Fractal Web Site.
`http://fractal.ow2.org/`.

[33] D. Garlan, R.T. 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.

[34] A.P. Gaufillet and B.S. Gabel.
Avionic software development with TOPCASED SAM.
In *Proc. Embedded Real Time Software and Systems 2010*, 2010.

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

[36] N. He, D. Kroening, T. Wahl, K.-K. Lau, F. Taweel, C. Tran, P. Rümmer, and S. Sharma.
Component-based design and verification in X-MAN.
In *Proc. Embedded Real Time Software and Systems*, 2012.

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

[38] IBM.
Service Component Architecture (SCA), Document Version 1.0, March 2010.
`http://public.dhe.ibm.com/software/htp/cics/pdf/sca_whitepaper.pdf`.

[39] JavaBeans web page.
`http://docs.oracle.com/javase/tutorial/javabeans/`.

[40] K. Kaur and H. Singh.
Candidate process models for component based software development.
Journal of Software Engineering, 4(1):16–29, 2010.

[41] G. Kotonya, I. Sommerville, and S. Hall.
Towards a classification model for component-based software engineering research.
In *Proc. 29th EUROMICRO Conference*, pages 43–52. IEEE Computer Society, 2003.

[42] K.-K. Lau, M. Pantel, D. Chen, M. Persson adn M. Törngren, and C. Tran.
Component-based development.
In A. Rajan and T. Wahl, editors, *CESAR – Cost-efficient Methods and Processes for Safety-relevant Embedded Systems*, chapter 5, pages 179–212. Springer-Verlag Wien, 2013.

[43] K.-K. Lau and T. Rana.
A taxonomy of software composition mechanisms.
In *Proc. 36th EUROMICRO Conference on Software Engineering and Advanced Applications*, pages 102–110. IEEE, 2010.

[44] K.-K. Lau, F. Taweel, and C. Tran.
The W Model for component-based software development.
In *Proc. 37th EUROMICRO Conference on Software Engineering and Advanced Applications*, pages 47–50. IEEE, 2011.

[45] K.-K. Lau and C. Tran.
X-MAN: An MDE tool for component-based system development.
In *Proc. 38th EUROMICRO Conference on Software Engineering and Advanced Applications*, pages 158–165. IEEE, 2012.

[46] K.-K. Lau, P. Velasco Elizondo, and Z. Wang.
Exogenous connectors for software components.
In G.T. Heineman *et al.*, editor, *Proc. 8th Int. Symp. on Component-based Software Engineering, LNCS 3489*, pages 90–106. Springer-Verlag, 2005.

[47] K.-K. Lau and Z. Wang.
A taxonomy of software component models.
In *Proc. 31st Euromicro Conference on Software Engineering and Advanced Applications*, pages 88–95. IEEE Computer Society Press, 2005.

[48] K.-K. Lau and Z. Wang.
Software component models.
IEEE Trans. on Software Engineering, 33(10):709–724, October 2007.

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

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

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

[52] R.B. Natan.

CORBA: A Guide to Common Object Request Broker Architecture.
McGraw-Hill, 1995.

[53] OMG.
Omg unified modeling language specification, November 2007.
`http://www.omg.org/cgi-bin/doc?formal/07-11-01.pdf`.

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

[55] D. S. Platt.
Introducing Microsoft .NET.
Microsoft Press, 3rd edition, 2003.

[56] F. Plášil, D. Balek, and R. Janecek.
Sofa/dcup: Architecture for component trading and dynamic updating.
In *Proceedings of the ICCDS98*, pages 43–52. IEEE Press, 1998.

[57] R. Reussner, S. Becker, E. Burger, J. Happe, M. Hauck, A. Koziolek, H. Koziolek, K. Krogmann, and M. Kuperberg.
The Palladio component model.
Technical report, Karlsruhe Institute of Technology - Faculty of Informatics, March 2011.

[58] S. Sentilles, A. Vulgarakis, T. Bures, J. Carlson, and I. Crnkovic.
A component model for control-intensive distributed embedded systems.
In MichelR.V. Chaudron, Clemens Szyperski, and Ralf Reussner, editors, *Component-Based Software Engineering*, volume 5282 of *Lecture Notes in Computer Science*, pages 310–317. Springer Berlin Heidelberg, 2008.

[59] SOFA 2 web site.
`http://sofa.ow2.org/`.

[60] I. Sommerville.
Software Engineering.
Addison Wesley, 7th edition, June 2004.

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

[62] C. Szyperski, D. Gruntz, and S. Murer.

Component Software: Beyond Object-Oriented Programming.
Addison-Wesley, second edition, 2002.

[63] The V-model. Development standard for IT-systems of the Federal Republic of Germany, IABG.
`http://www.v-modell.iabg.de`.

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

[65] R. van Ommering, F. van der Linden, J. Kramer, and J. Magee.
The Koala component model for consumer electronics software.
IEEE Computer, 33(3):78–85, 2000.

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