Towards a Theory of Software Components

Kung-Kiu Lau
School of Computer Science
The University of Manchester
UK
kung-kiu@cs.man.ac.uk

Mario Ornaghi
DSI
Università Milano
Italy
ornaghi@dsi.unimi.it
Overview

- What are software component models?
- What are software components?
- Our software component model
- Towards a theory for our model
# Systems vs Components

<table>
<thead>
<tr>
<th>Component-based systems</th>
<th>Component-based software engineering</th>
</tr>
</thead>
<tbody>
<tr>
<td>system-level focus</td>
<td>component-level focus</td>
</tr>
<tr>
<td>no component models (?)</td>
<td>component models, e.g. EJB</td>
</tr>
<tr>
<td>communication, concurrency, processes, protocols, etc</td>
<td>software composition, middleware</td>
</tr>
<tr>
<td>temporal, non-functional properties: reliability, fault-tolerance, deadlock-freedom, safety, etc.</td>
<td>functional &amp; non-functional properties: correctness, reliability, robustness, maintainability</td>
</tr>
</tbody>
</table>
Software Component Models

A software component model defines:

- what components are
  - syntax of components
  - semantics of components
- how to compose components
Current Component Definitions

- **Szyperski:**
  “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:**
  “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.”
Current Component Definitions (Continued)

- Heineman and Council

  “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.”

Comparison wrt component models:

<table>
<thead>
<tr>
<th>Definition</th>
<th>Based on CM</th>
<th>Defines CM</th>
</tr>
</thead>
<tbody>
<tr>
<td>Szyperski</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>Meyer</td>
<td>No</td>
<td>No</td>
</tr>
<tr>
<td>Heineman and Council</td>
<td>Yes</td>
<td>No</td>
</tr>
</tbody>
</table>
Current Software Component Models

<table>
<thead>
<tr>
<th>Semantics</th>
<th>Components</th>
<th>Composition</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Name</td>
<td>??</td>
</tr>
<tr>
<td></td>
<td>Interface</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Code</td>
<td></td>
</tr>
<tr>
<td></td>
<td>or</td>
<td></td>
</tr>
<tr>
<td></td>
<td>provided services</td>
<td></td>
</tr>
<tr>
<td></td>
<td>required services</td>
<td></td>
</tr>
<tr>
<td>Typical examples</td>
<td>objects</td>
<td>method calls</td>
</tr>
<tr>
<td></td>
<td>architectural units</td>
<td>ADL connectors</td>
</tr>
</tbody>
</table>
## Categories based on Component Syntax

<table>
<thead>
<tr>
<th>Component Syntax</th>
<th>Models</th>
</tr>
</thead>
<tbody>
<tr>
<td>Object–oriented Programming Languages</td>
<td>JavaBeans, EJB</td>
</tr>
<tr>
<td>Programming Languages with IDL mappings</td>
<td>COM, CCM, Fractal</td>
</tr>
<tr>
<td>Architecture Description Languages</td>
<td>ADLs, UML2.0, KobrA, Koala, SOFA, PECOS, Pin</td>
</tr>
</tbody>
</table>
Categories based on Component Semantics

<table>
<thead>
<tr>
<th>Component Semantics</th>
<th>Models</th>
</tr>
</thead>
<tbody>
<tr>
<td>Classes</td>
<td>JavaBeans, EJB</td>
</tr>
<tr>
<td>Objects</td>
<td>COM, CCM, Fractal</td>
</tr>
<tr>
<td>Architectural Units</td>
<td>ADLs, UML2.0, KobrA, Koala, SOFA, PECOS, Pin</td>
</tr>
</tbody>
</table>
Categories based on Composition

Category 1
(JavaBeans)

Category 2
(EJB, COM, CCM)

Category 3
(Koala, SOFA, KobrA)

Category 4
(ADLs, UML2.0, PECOS, Pin, Fractal)
An Idealised Component Life Cycle

- **Design**: Builder, Repository
- **Deployment**: Assembler
- **Run-time**: RTE

- **Component (source or binary)**: blue
- **Design phase composition operator**: red
- **Component (binary)**: yellow
- **Deployment phase composition operator**: red
- **Component instance**: yellow

FMCO 2005
This life cycle reflects CBSE desiderata:

<table>
<thead>
<tr>
<th>Components</th>
<th>Repository</th>
</tr>
</thead>
<tbody>
<tr>
<td>Components pre-exist</td>
<td>repository</td>
</tr>
<tr>
<td>Components produced &amp; used independently</td>
<td>builder &amp; assembler (+ repository)</td>
</tr>
<tr>
<td>Components can be copied and instantiated</td>
<td>design &amp; deployment + run-time phases</td>
</tr>
<tr>
<td>Composites can be made and used for further composition</td>
<td>composites in repository</td>
</tr>
</tbody>
</table>
Enterprise JavaBeans (EJB)

A = EJBA (EJB JAR file)
B = EJBB (EJB JAR file)
InsA = EJBA instance
InsB = EJBB instance

= Method calls
EJB Example

A = BookBean (EJB JAR file)
B = BookStoreBean (EJB JAR file)
InsA = BookBean instance
InsB = BookStoreBean instance
● = Method calls
Our Proposed Component Model

- **Components** encapsulate **computation**
  (no method calls to other components)

- **(Exogenous) Connectors** encapsulate **communication** (data & control)
What is a Software Component?

A software component is:

- **not** simply a part of the whole system
- **very different** from
  - traditional software units:
    - code fragments, functions, procedures, subroutines, modules, classes/objects, programs, packages
  - more modern units:
    - DLLs, services
Software Components: A Definition

A software component is a software unit with the following defining characteristics:

- encapsulation
- compositionality
Encapsulation

A component $C$ encapsulates

- data
  ($C$ has private data)
- computation
  ($C$’s computation happens entirely within $C$)
- An **oo object** also encapsulates data, but it does **not** encapsulate computation.

```
Object A

m1(...){
  ...
  B.m2(...)
}

Object B

m2(...)
```

- **Port-connector** type components, as in e.g. ADLs, UML2.0 and Koala, **can** encapsulate data.

However, they usually do **not** encapsulate computation.
Compositionality

Components should be **compositional**:

- the composition of two components $C_1$ and $C_2$ should yield another component $C_3$
  
- $C_3$ should also have the defining characteristics of **encapsulation** and **compositionality**

Thus **compositionality** $\rightarrow$ **composition** preserves or propagates encapsulation

*Compositionality in terms of other (non-functional) properties of sub-components is an open issue*
• **Classes and objects** are **not** compositional:
  – They can only be ‘composed’ by method calls such a ‘composition’ does not yield another class or object.
  – oo method calls break encapsulation

• **Port-connector** type components **can** be composed, but they are **not** compositional if they do not have (computation) encapsulation.
Interfaces

Encapsulation entails that access to components must be provided by interfaces

- **Classes/objects** do **not** have interfaces:
  - Access to (the methods of) objects, if permitted, is direct, not via interfaces
  - ‘Interfaces’ in oo languages like Java are themselves classes/objects, so are not interfaces to components.

- **Port-connector** type components have their ports as their interfaces.
Composition Operators

It should be possible to construct systems or composite components by composing components.

Such composition operators should work only on interfaces (in view of encapsulation),

and they must ensure compositionality.

- **Glue code** is certainly not suitable as composition operators.
- **Neither** are oo method calls or ADL connectors.
- In fact, **none** of the so-called ‘component models’, e.g. EJB, have proper composition operators.
Component Theories

So-called component models do not define theories of components with encapsulation and compositionality.

Definitions like the one by Szyperski also do not have these concepts.

We are undertaking research in developing and implementing such theories (or ‘component models’).

As a first step, we have proposed exogenous connectors as composition operators.

Components with these connectors as interfaces are compositional.

Of course our components also have encapsulation.
Components and Connectors: Traditional View

Supposedly:

- component = computation
- connector = interaction

Actually:

- component = control + computation
- connector = communication

(components not truly independent)
Traditional Connection Schemes

(a) direct message passing

(b) indirect message passing
Exogenous Connection Scheme

component = computation

connector = communication (control & data)
Connection Schemes: Comparison

<table>
<thead>
<tr>
<th>Connection Scheme</th>
<th>Component Models</th>
<th>Control mixed with computation</th>
<th>Calls initiated from outside component</th>
</tr>
</thead>
<tbody>
<tr>
<td>Direct message passing</td>
<td>EJB, CCM, COM, UML2.0, KobrA</td>
<td>✓</td>
<td>✗</td>
</tr>
<tr>
<td>Indirect message passing</td>
<td>JavaBeans, ADLs, PECOS, Koala, Pin</td>
<td>✓</td>
<td>✗</td>
</tr>
<tr>
<td>Exogenous connection</td>
<td>Our proposed model</td>
<td>✗</td>
<td>✓</td>
</tr>
</tbody>
</table>

- Exogenous connectors **encapsulate** control (& data)
  - initiate & coordinate

- Components are truly **independent & decoupled**
Creating and Using Exogenous Connectors

We need a hierarchy of connectors:

- (unary) invocation connectors (level 1)
- (n-ary) connectors for invocation connectors (level 2)
- (n-ary) level-\(m\) connectors for connectors on levels \(k < m\)
Type Hierarchy of Connectors

**Basic types**

Component, Result;

**Connector types**

\[ L_1 \equiv \text{Invocation} \equiv \text{Component} \rightarrow \text{Result}; \]

\[ L_2 \equiv \text{Component} \rightarrow \text{Result}; \]

\[ L_3 \equiv \text{Component} \rightarrow \text{Result}; \]

\[ L \equiv \text{Component} \rightarrow \text{Result}; \]

\[ L \equiv L \times \ldots \times L \rightarrow \text{Result} \]

where \( L \) is either \( L_1 \) or \( L_2 \);

\[ \ldots \]

- level-1 and level-2 connectors are not polymorphic
- connectors at higher levels are polymorphic
Example: Bank System

Acme

C2

Exogenous connection
- Hierarchical system structure, reflecting connector hierarchy
- Components are Java classes that do not call each other

```java
package system;
import java.lang.reflect.*;
import java.io.*;
import connectors.*;
public class BankSystem {
    public static void main (String[] args) {
        // create instances of components
        ATM atm = new ATM("1"); ... Bank bank4 = new Bank("4");
        // create level-one connectors
        Invocation invATM = new Invocation((Object) atm); ... Invocation invB4 = new Invocation((Object) bank4);
        // create level-two connectors
        Invocation[] invsBank12 = new Invocation[2]; invsBank12[0] = invB1; invsBank12[1] = invB2;
        Selector s1 = new Selector(invsBank12);
        // create level-three connectors
        Connector[] consBC1 = new Connector[2]; consBC1[0] = invBC1; consBC1[1] = s1;
        Pipem p2 = new Pipem(consBC1); ... Pipem p3 = new Pipem(consBC2);
        // create level-four connectors
        Connector[] consAB = new Connector[2]; consAB[0] = p2; consAB[1] = p3;
        Selectorm s3 = new Selectorm(consAB);
        // create level-five connectors
        Connector[] consm = new Connector[2]; consm[0] = invATM; consm[1] = s3;
        Pipem p1 = new Pipem(consm);
        // Display menu and initiate operations
        switch(Integer.parseInt(args[0]))) {
            case 1:
                System.out.println("Your balance is:");
                p1.execute(ms, params);
                break; ...
            } }
    }
```
Comparison with Acme and C2 Implementations

(a) Acme/ArchJava

```java
public class BankConsortium{

    // from ATM
    port in{
        requires String[] getCardNo (String[] inResult) ... 
    }
    
    // towards banks
    port out{
        provides String[] identifyBank (String[] inRes) ... 
    }
}
```

(b) C2/ArchStudio

```java
public class BankConsortium extends AbstractC2DelegateBrick{

    class BankConsortiumMP implements MessageProcessor{
        public void handle (Message m){
            NamedPropertyMessage m = (NamedPropertyMessage)m;
            if(m.getName().equals("cardNumber")){
                res = (String[])m.getParameter("data");
                result = identifyBank(res);
                if(res[1].equals("1")
                    num = "Bank1"
                else if (res[1].equals("2")
                    num = "Bank2"
                m = new NamedPropertyMessage("DoOperation");
                m.addParameter("data", result);
                m.addParameter("bankNo", num);
                sendToAll(m, topIface);
            }
        } }

    public String[] identifyBank (String[] inRes) ... {
    }
```

(c) Exogenous connection

```java
public class BankConsortium{

    public String[] identifyBank (String[] inResult) ... {
    }
```

- green: computation
- yellow: computation/control
- blue: communication

FMCO 2005
Encapsulation and Compositionality

- Our components are built from (encapsulated) computational units.
- Exogenous connectors allow us to:
  - make (computational) units into (encapsulated) components
  - compose components into composite (components).
Encapsulation and Compositionality (Continued)

A system of components and exogenous connectors

A component
Invocation connector
Computational unit
Encapsulation

A composite component
Compositionality
Level n>=2 connector
Encapsulation
Towards a Theory

A theory for components and connectors in this life cycle

FMCO 2005
Repositories

The signature for specifying a repository is characterized by:

- A sort $\text{Comp}$ for components, and a set of constants of sort $\text{Comp}$ identifying the current components
- A sort $\text{Conn}$ for connectors, and a set of constants of sort $\text{Conn}$, identifying the currently defined connectors
- A generic sort $\text{Binary}$ corresponding to the deployed binaries
- A generic sort $\text{Instance}$ corresponding to the run-time instances of binaries
- A generic sort $\text{Data}$ for problem domain data
 Repositories (Continued)

- A family of request types $\mathcal{Q}$ and a family of result types $\mathcal{R}$;
  If $Q \in \mathcal{Q}$ and $R \in \mathcal{R}$, then $Q >> R$ is an operation type

Repositories are related to problem domains and the families $\mathcal{Q}$ and $\mathcal{R}$ are provided by some formal characterisation, depending on the problem domain.
Components

The minimal assumptions for a component in a repository are:

- Each component has an interface: \( C :: Q >>> R \) indicates that \( Q >>> R \) is an operation type supported by \( C \).

- Components can be deployed: the predicate \( \text{deployed}(id, C) \) indicates that component \( C \) has been deployed with identifier \( id \) and a deployed binary.
  
  Deployment preserves the interface \( id :: Q >>> R \) iff \( C :: Q >>> R \).

- Instances are uniquely identified and can hold data: \( \text{inst}(id, d) \) represents an instance with identifier \( id \) and data \( d \).
  
  By \( \text{instance}(\text{inst}(id, d), C) \) we indicate that \( \text{inst}(id, d) \) is actually an instance.
Components (Continued)

- Instances have the same identifiers as the deployed components:
  \[
  \text{instance}(\text{inst}(id, d), C) \rightarrow \exists C(\text{deployed}(id, C))
  \]

- Deployed components can be instantiated: \(\text{instantiate}(id, d)\) represents the operation of instantiating a deployed component \(id\) with (possible) initialisation data \(d\). It yields an instance \(\text{inst}(id, d_0)\).

- Instances can execute requests:
  - if \(id :: Q >> R\) and \(q \in Q\), the result of the execution function is of the following form:
    \[
    \text{exec}(\langle id, d \rangle, q) = \langle \langle id, d' \rangle, r \rangle
    \]
    It indicates that executing \(q\), \(id\) (possibly) updates \(d\) into \(d'\)
and yields the result \( r \in R \); we will also write:

\[
\text{next}(\langle id, d \rangle, q) = d' \\
\text{res}(\langle id, d \rangle, q) = r
\]

The execution function is an abstraction of the execution of the running code.

- For an instance \( \text{inst}(id, D) :: Q >> R \), we represent data, requests and results at an abstract level and we give a semantic relation \( \text{pre}, \text{post} \), where:

\[
q \in \text{pre}(Q, id) \text{ entails } \text{res}(\text{inst}(id, D), q) \in \text{post}(R)
\]
Invocation Connectors

We have invocation connectors and composition connectors:

\[ InvConn \leq Conn, \quad CompConn \leq Conn. \]

An invocation connector makes a unit into a component:

\[ ic(i : InvConn, u : Unit) : Comp \]

The semantics depends on both \( i \) and \( u \), and has to define:

- the interface relation \( ic(i, u) :: Q >> R \)
- the execution relation \( \text{exec}(\text{inst}(id, d), q) \) for a generic instance \( id \) holding data \( d \);
- the way of deploying and instantiating \( ic(i, u) \).
Composition Connectors

A composition connector composes components with suitable interfaces into a new one.

The composition type of a connector is defined by a set of inference rules:

\[ x_1 :: Q_1 >> R_1, \ldots, x_n :: Q_n >> R_n \Rightarrow cc(c, [x_1, \ldots, x_n]) :: Q >> R \]

indicating that from components

\[ C_1 :: Q_1 >> R_1, \ldots, C_n :: Q_n >> R_n \]

we can build a component \( cc(c, [C_1, \ldots, C_n]) :: Q >> R \).
To each inference rule we associate a compositional semantics, defining in a compositional way (i.e. depending only on the subcomponents):

- the way of deploying and instantiating the components built using the rule
- the \texttt{exec} function
Composition Connectors (Continued)

Deployment is defined bottom up, according to the following deployment expressions:

- **Basis.** We start by deploying the basic subcomponents: 
  
  \[ dp(id, ic(c, u) :: Q >> R) \]
  
  is a deployment expression iff 
  
  \[ ic(c, u) :: Q >> R \]
  
  is obtained by an invocation connector. It intuitively means that \( id \) identifies a binary deployed for \( ic(c, u) \).

- **Step.** If 
  
  \[ dp(i_j, D_j) :: Q_j >> R_j \]
  
  are deployment expressions and 
  
  \[ x_1 :: Q_1 >> R_1, \ldots, x_n :: Q_n >> R_n \rightarrow cc(c, [x_1, \ldots, x_n]) :: Q >> R \]
  
  is a composition rule, then 
  
  \[ dp(id, cc(c, [i_1, \ldots, i_n])) :: Q >> R \]
  
  is a deployment expression.

A component can give rise to many deployment expressions, according to the way of deploying its basic components. Indeed, a basic subcomponent with many occurrences can be deployed in different locations (i.e., with different deployment identifiers).
Instantiation is performed by instantiating the basic subcomponents, as prescribed by the invocation connectors and by (possible) further constraints related to the application considered.
An Example

The BankAccounts repository has invocation connectors for a suitable set of units, including the unit `account`, containing the code for storing, accessing and updating accounts, and the unit `pins`, storing, accessing and updating the PINs associated to the account numbers. In our model we have the constants:

\[
\text{account, pins : Unit.}
\]

The connectors of BankcAccount include:

\[
\text{orSelect, pipe : CompConn and atm : InvConn.}
\]
Their semantics is given as follows:

- **atm can only invoke the units** account and pins, i.e.:

  \[ \text{component}(ic(\text{atm}, U)) \iff U = \text{account} \lor U = \text{pins} \]

- **The data for the instances of** $ic(\text{atm}, \text{account})$ **are relations**
  
  \( \text{Account} = \{[n_1, a_1], \ldots, [n_k, a_k]\} \) **associating to each account number** $n_j$ **an amount** $a_j$:

  \[ \text{instance}(\text{inst}(\text{id}, ic(\text{accountInvk}, \text{account}), \text{Account})) \iff \text{Account} \in \text{rel}(n : \text{accountN}, a : \text{float}) \]
An Example (Continued)

- The interface relation of \( ic(atm, account) \) is

\[
ic(atm, account) \::\: withdraw(N, W) >> or(done, refused) |
\]
\[
deposit(N, D)) >> done |
\]
\[
getAmount(N) >> amount(N, A).
\]

- Let \( inst(b, ic(atm, account), Account_b) \) be an instance, where the identifier \( b \) identifies the bank where the instance binary has been deployed.
The request and result types have the following semantics:

\[
\begin{align*}
\text{withdraw}(b, n, w) & \in \text{pre}(id, \text{withdraw}(N, W)) \iff \exists a([n, a] \in \text{Account}_b) \\
\text{deposit}(b, n, w) & \in \text{pre}(id, \text{deposit}(N, W)) \iff \exists a([n, a] \in \text{Account}_b) \\
\text{getAmount}(b, n) & \in \text{pre}(id, \text{getAmount}(N)) \iff \exists a([n, a] \in \text{Account}_b) \\
\text{amount}(b, n, a) & \in \text{post}(id, \text{amount}(B, N, A)) \iff [n, a] \in \text{Account}_b \\
\text{sem}(\text{or}(\text{done}, \text{refused}), id) & \quad = \{\text{done}, \text{refused}\} \\
\text{sem}(\text{done}, id) & \quad = \{\text{done}\}
\end{align*}
\]

Remark: done, refused, or(done, refused) do not depend on id.

- Let \(\text{inst}(b, ic(atm, account), \text{Account}_b)\) be an instance. The run time behaviour of \(b\) is given by the corresponding binary. It is correct if it implements the following abstract execution function:

  \text{Withdraw, case 1. Let} \ a \ \text{be the amount of account} \ n \ (i.e.,}
An Example (Continued)

\[ [n, a] \in \text{Account}; \text{if } a \geq w, \text{ then:} \]

\[
\text{next}(\text{inst}(b, \text{Account}), \text{withdraw}(n, w)) = \text{Account}([n, a] := [n, a - w])
\]

\[
\text{res}(\text{inst}(b, \text{Account}), \text{withdraw}(n, w)) = \text{done}
\]

where \( \text{Account}([n, a] := [n, \text{Expression}]) \) is the relation obtained by replacing the tuple \([n, a]\) by \([n, \text{Expression}]\)

Withdraw, case 2. Let \( a \) be the amount of account \( n \); if \( a > w \), then:

\[
\text{next}(\text{inst}(b, \text{Account}), \text{withdraw}(b, n, w)) = \text{Account}
\]

\[
\text{res}(\text{inst}(b, \text{Account}), \text{withdraw}(b, n, w)) = \text{refused}
\]

Deposit:

\[
\text{next}(\text{inst}(b, \text{Account}), \text{deposit}(b, n, d)) = \text{Account}([n, a] := [n, a + d])
\]

\[
\text{res}(\text{inst}(b, \text{Account}), \text{deposit}(b, n, d)) = \text{done}
\]
An Example (Continued)

Get Amount. Let \( a \) be the amount of account \( n \)

\[
\text{next}(b, \text{Account}), \text{getAmount}(b, n)) = \text{Account}
\]

\[
\text{res}(b, \text{Account}), \text{getAmount}(b, n)) = \text{amount}(b, n, a)
\]

- The data for the instances of \( ic(\text{atm}, \text{pins}) \) are relations
  \[
  \text{Pins} = \{[n_1, p_1, b_{j_1}], \ldots, [n_k, p_k, b_{j_k}]\}
  \]
  associating a pin \( p_j \) and a bank \( b_{j_n} \) to each account number \( n_j \):

  \[
  \text{instance}(\text{inst}(\text{id}, \text{ic}(\text{pinsInvk}, \text{pins}), \text{Pins})) \leftrightarrow \text{Pins} \in \text{rel}(n: \text{accountN}, p: \text{pin}, b \in \{b1, b2\}).
  \]

- The interface relation of \( ic(\text{pinsInvk}, \text{Pins}) \) is

\[
\text{ic}(\text{atm}, \text{pins}) :: \text{withdrawReq}(N, P, W) >>
\]

\[
\text{or}(\text{withdraw}(b1, N, W), \text{withdraw}(b2, N, W), \text{refused}(N, P))
\]
• Let \( \text{inst}(id, Pins_{id}) \) be an instance. The request and result types have the following semantics:

\[
\text{withdrawReq}(id, n, p, w) \in \text{pre}(\text{withdrawReq}(N, P), id)
\]

iff \( n : \text{accountN}, p : \text{pinN} \)

\[
\text{withdraw}(id, b, n, w) \in \text{pre}(\text{withdraw}(B, N, W), id)
\]

iff \( \exists p([b, n, p] \in Pins_{id}) \)

\[
\text{sem(}\text{refused, id}) = \{ \text{refused}\}
\]
An Example (Continued)

- The run time behaviour of the connector is given by the binary code implementing it. It is correct if it implements the following abstract execution function:

\[
\begin{align*}
\text{next}(\text{inst}(id, \text{Pins}), \text{withdrawReq}(n, p, w)) &= \text{Pins} \\
\text{res}(\text{inst}(id, \text{Pins}), \text{withdrawReq}(n, p, w)) &= \text{withdraw}(b, n, w) \\
\text{if } &\left[ b, n, p \right] \in \text{Pins} \\
\text{res}(\text{inst}(id, \text{Pins}), \text{withdrawReq}(n, p, w)) &= \text{refused} \\
\text{if } &\forall b\left[ b, n, p \right] \notin \text{Pins}
\end{align*}
\]

For two banks \(b_1, b_2\) with a pin identifier \(id\), we can deploy:

\[
\begin{align*}
\text{dp}(b_1, \text{ic(atm, account)}) &:: \text{withdraw}(b_1, N, W) >> \text{or(done, refused)} \\
\text{dp}(b_2, \text{ic(atm, account)}) &:: \text{withdraw}(b_2, N, W) >> \text{or(done, refused)}
\end{align*}
\]
An Example (Continued)

\[
dp(id, ic(atm, pins)) :: withdrawReq(ID, N, P) >>
\]
\[
\text{or}(withdraw(b1, N, W), withdraw(b2, N, W), refused)
\]
\[
cc(orSelect, [b1, b2, noOp]) ::
\]
\[
\text{or}(withdraw(b1, N, W), withdraw(b2, N, W), refused) >>
\]
\[
\text{or}(done, refused)
\]
\[
cc(pipe, [id, cc(orSelect, [b1, b2])]) :: withdraw(id, N, W) >>
\]
\[
\text{or}(done, refused)
\]

An instantiation is:

\[
\text{instance}(atm, cc(pipe, [id, cc(orSelect, [b1, b2])]), [Account}_{b1}, Account_{b2}, Pins_{id}]
\]
Since the pipe requires that $\text{sem}(\text{withdraw}(bj, N, W), id)$ entails $\text{sem}(\text{withdraw}(bj, N, V), bj)$, the instantiating data must be such that:

$$[b_1, n, p] \in Pins_{id} \rightarrow \exists w ([n, w] \in Account_{b_1} \text{ and } [b_2, n, p] \in Pins_{id} \rightarrow \exists w ([n, w] \in Account_{b_2})$$
Conclusion

- We have implemented our model (deployment phase) in Java and .NET.
- We have implemented a container (in C#) for the deployment phase that automatically generates a running system from its architecture.
- We are in the process of implementing the design phase in Java and SPARK (components with contracts).
- Work on the theory is just beginning though :-(