BPEL4WS (Business Process Execution Language for Web Services)

Francisco Curbera, Frank Leymann, Rania Khalaf

IBM

Business Process Execution Language

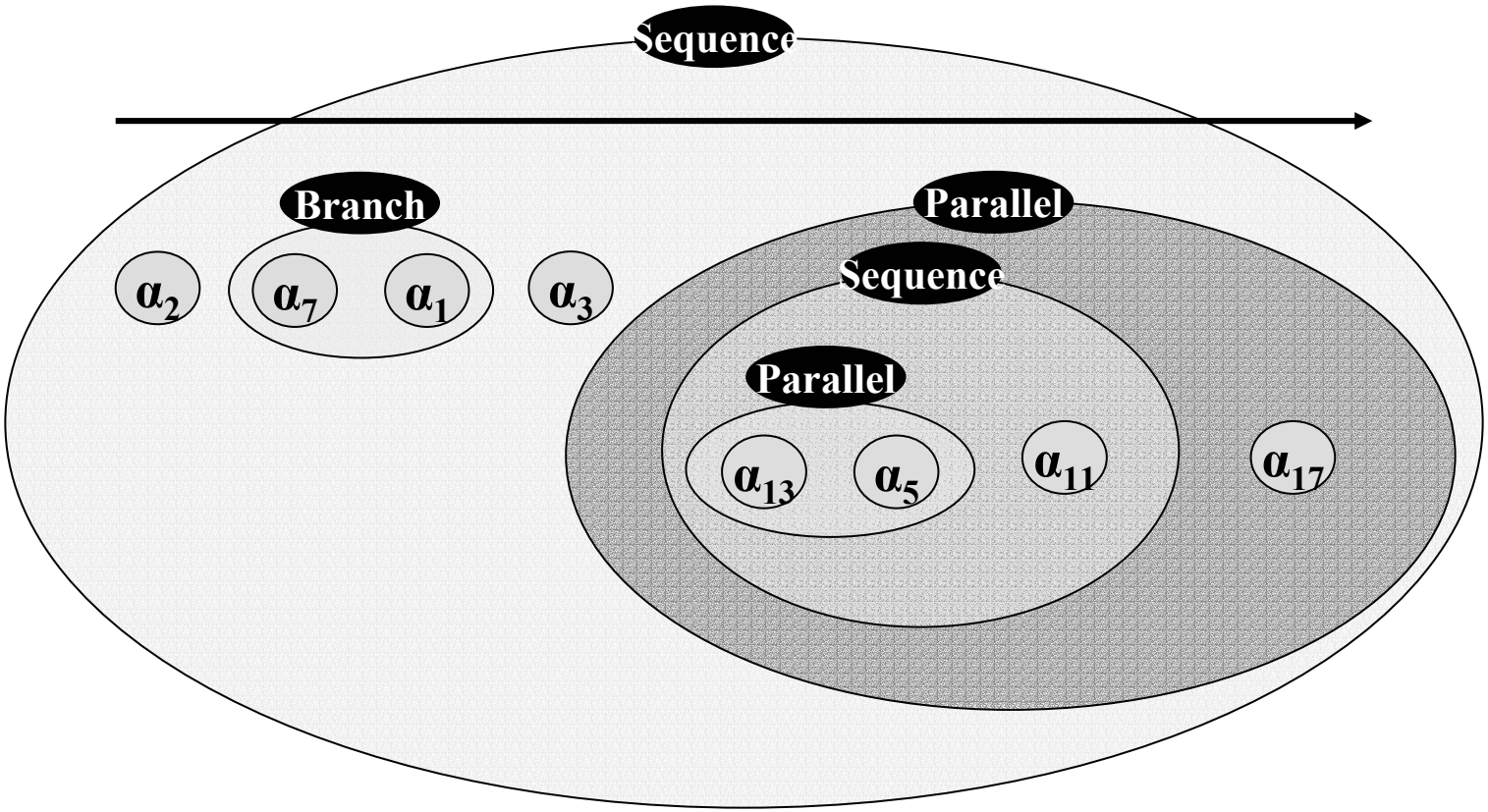
- BPEL4WS enables:
  - Defining business processes as coordinated sets of Web service interactions.
  - Define both abstract and executable processes.
    - Abstract processes are for e-commerce specifications.
    - Executable processes provide a model to integrating enterprise applications.
  - BPEL enables the creation of compositions of Web services
    - Composition based on abstract descriptions
- Where it comes from:
  - Strong roots in traditional flow models.
  - Plus many concepts from structured programming languages.
  - All laid on top of WSDL and core XML specifications.
  - Merges WSFL and XLANG concepts.

Algebraic/Calculus Approach To Flows

- Collection of „elementary“ activities
  - $A = \{\alpha_1, ..., \alpha_n\}$
  - RPC, wait, send,...
- Collection of „complex“ activities
  - $\Omega = \{\omega_1, ..., \omega_m\}$
  - Sequence, Parallel, Branch, Loop,...
  - Have other activities as parameters
    - Elementary as well as complex activities allowed
- $\omega_3(\alpha_2, \omega_1(\alpha_7, \alpha_1), \alpha_3, \omega_2(\omega_3(\omega_2(\alpha_{13}, \alpha_5), \alpha_{11}), \alpha_{17}))$

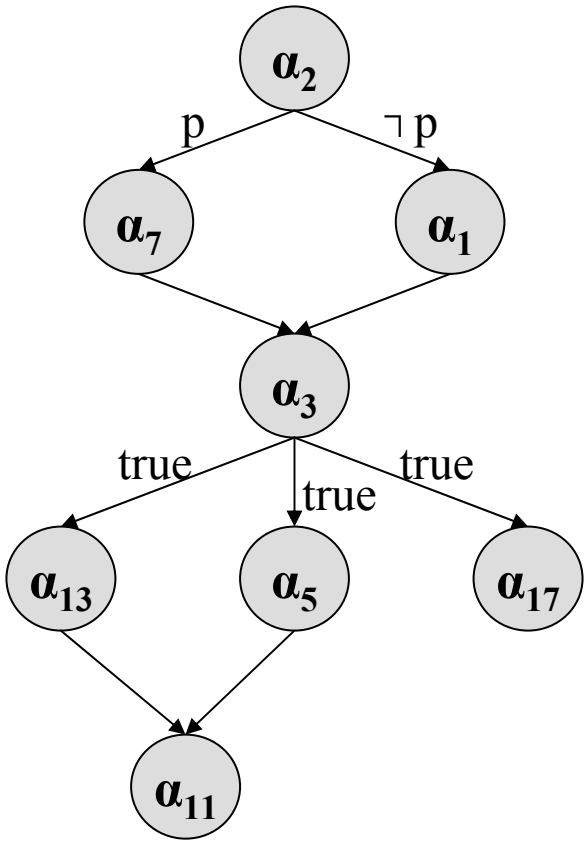
Algebraic Flows Representation

SEQ(α_2, BCH(α_7, α_1), α_3, PAR(SEQ (PAR (α_{13}, α_5), α_{11}), α_{17})))

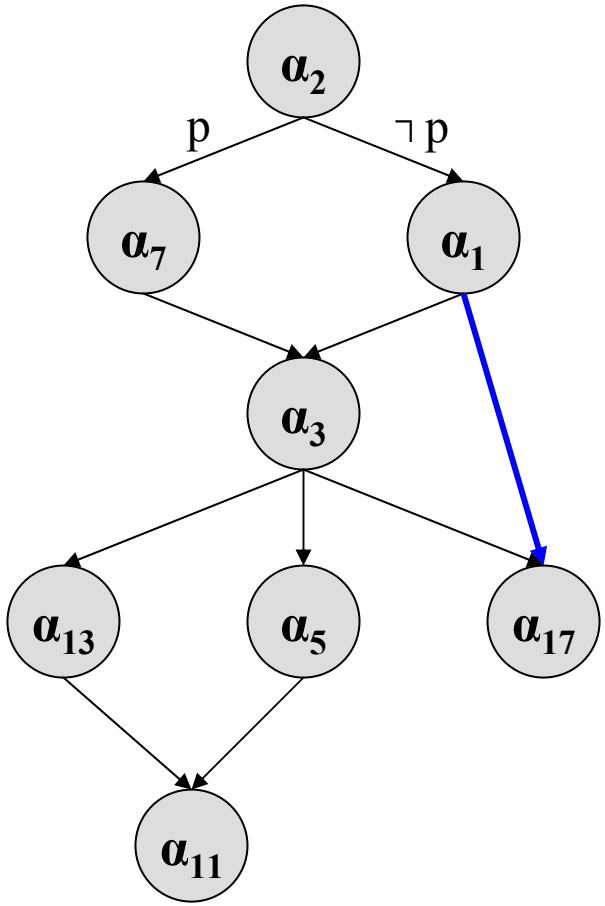


And As Pure Graph

SEQ(α_2, BCH(α_7, α_1), α_3, PAR(SEQ (PAR (α_{13}, α_5), α_{11}), α_{17})))



Finally: A “Non-Algebraic“ Graph



Structure of a BPEL4WS Process

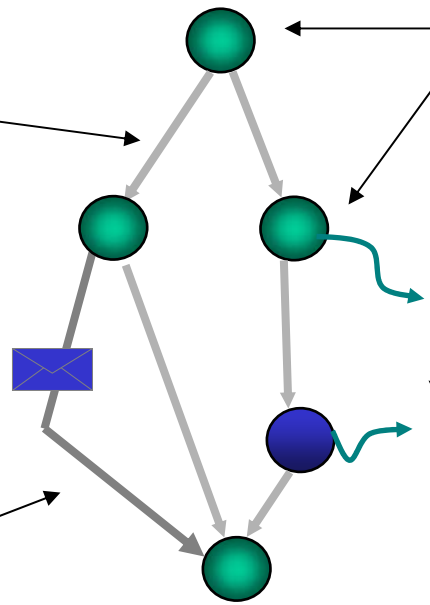
```
<process ...>

   <partners> ... </partners>
         <!-- Web services the process interacts with -->
   <variables> ... </variables>
         <!– Data used by the process -->
   <correlationSets> ... </correlationSets>
         <!– Used to support asynchronous interactions -->
   <faultHandlers> ... </faultHandlers>
         <!–Alternate execution path to deal with faulty conditions -->
   <compensationHandlers> ... </compensationHandlers>
         <!–Code to execute when “undoing” an action -->
   <eventHandlers> ... </eventHandlers>
         <!–Code for handling events-->
   (activities)*
         <!–  What the process actually does -->

</process>
```

Traditional Flow Models

Control links define execution flow as a directed acyclic graph

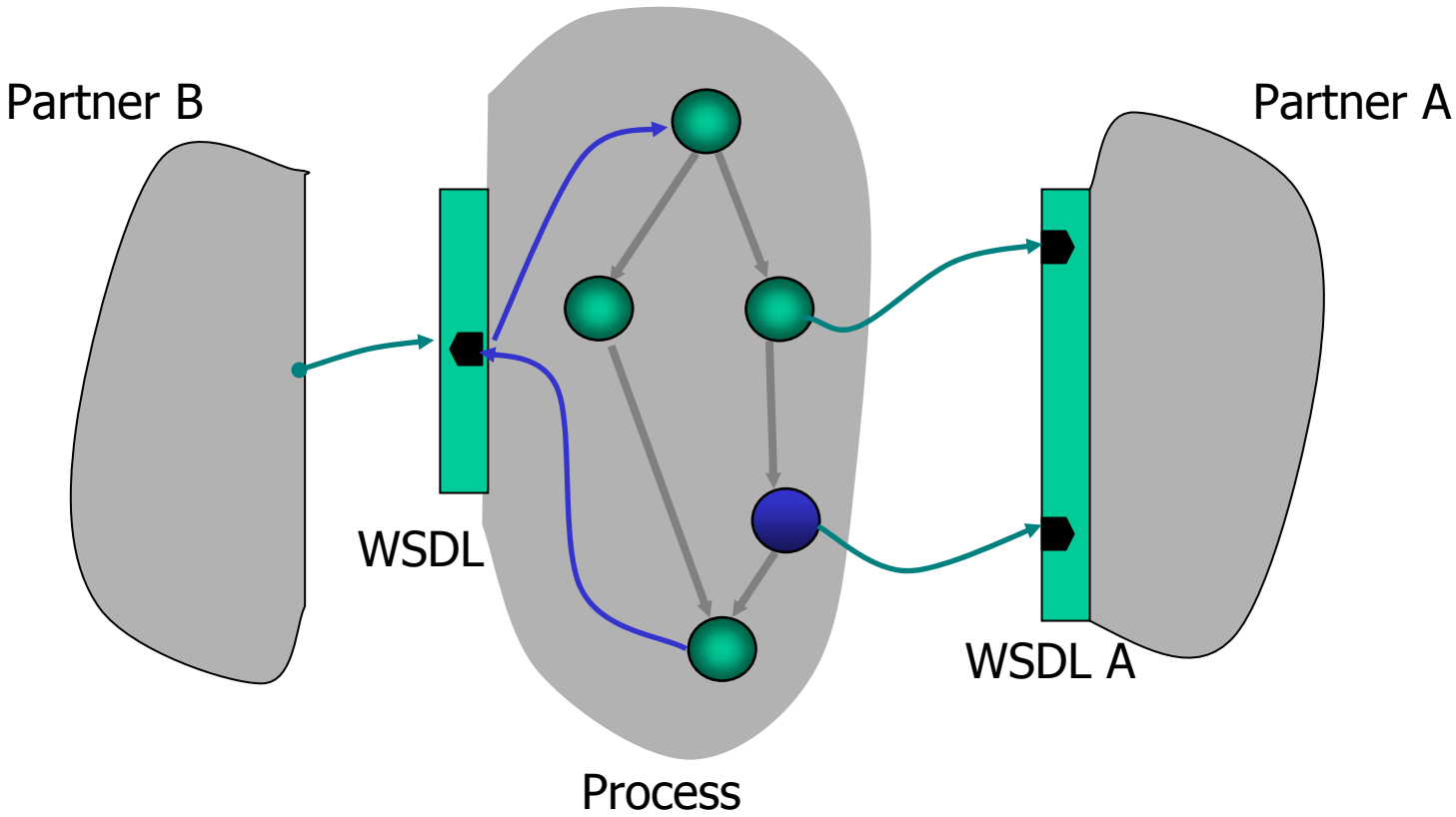


Activities represent units of processing.

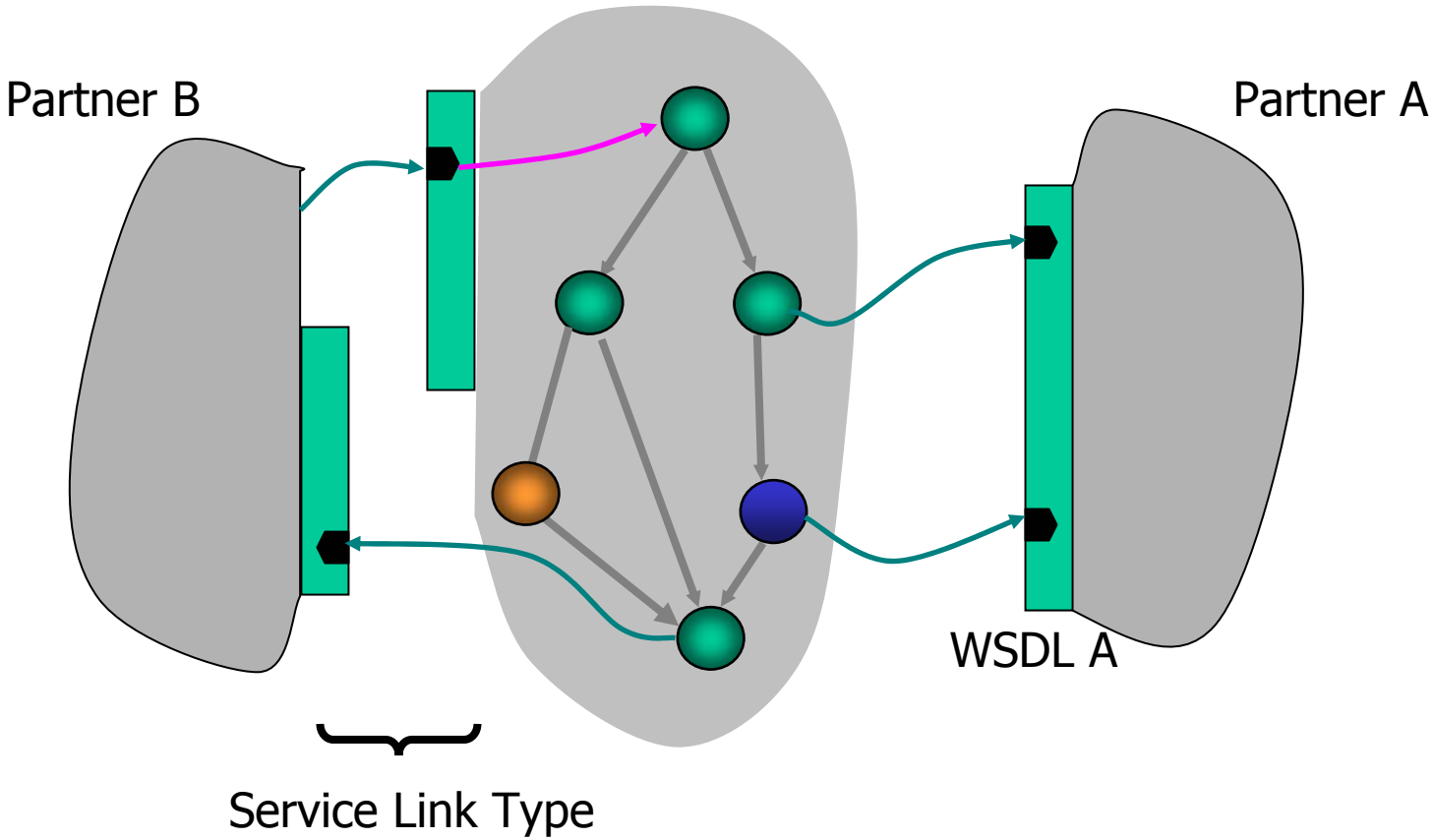
Flow of data is explicitly modeled through data links.

Activities are mapped to application invocations or human actions

BPEL and WSDL Partners



BPEL and WSDL Partners



Partner Definitions and Links

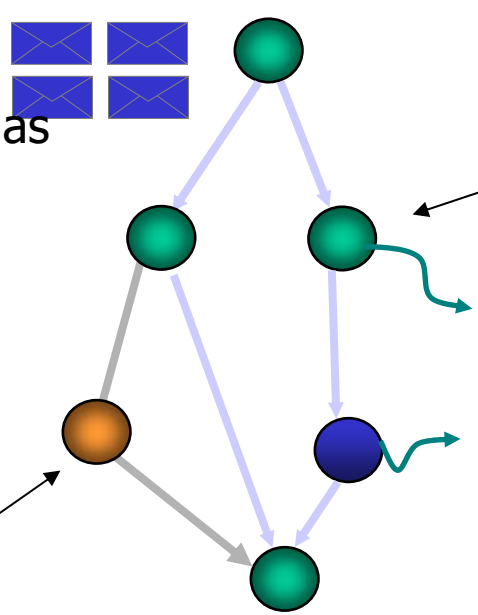
```
<partner name="..." serviceLinkType="..."
         partnerRole="..." myRole="..."/>
```

<!– A partner is accessed over a WS “channel”, defined by a service link type -->

```
<serviceLinkType name="...">
  <role name="...">
    <portType name="..."/>*
  </role>
  <role name="...">
    <portType name="..."/>*
  </role>
</serviceLinkType>
```

<!– A SLT defines two roles and the portTypes that each role needs to support -->

BPEL Data Model



scoped variables typed as WSDL messages

Activities input/ output is kept in scoped variables.

Assignment activities move data around.

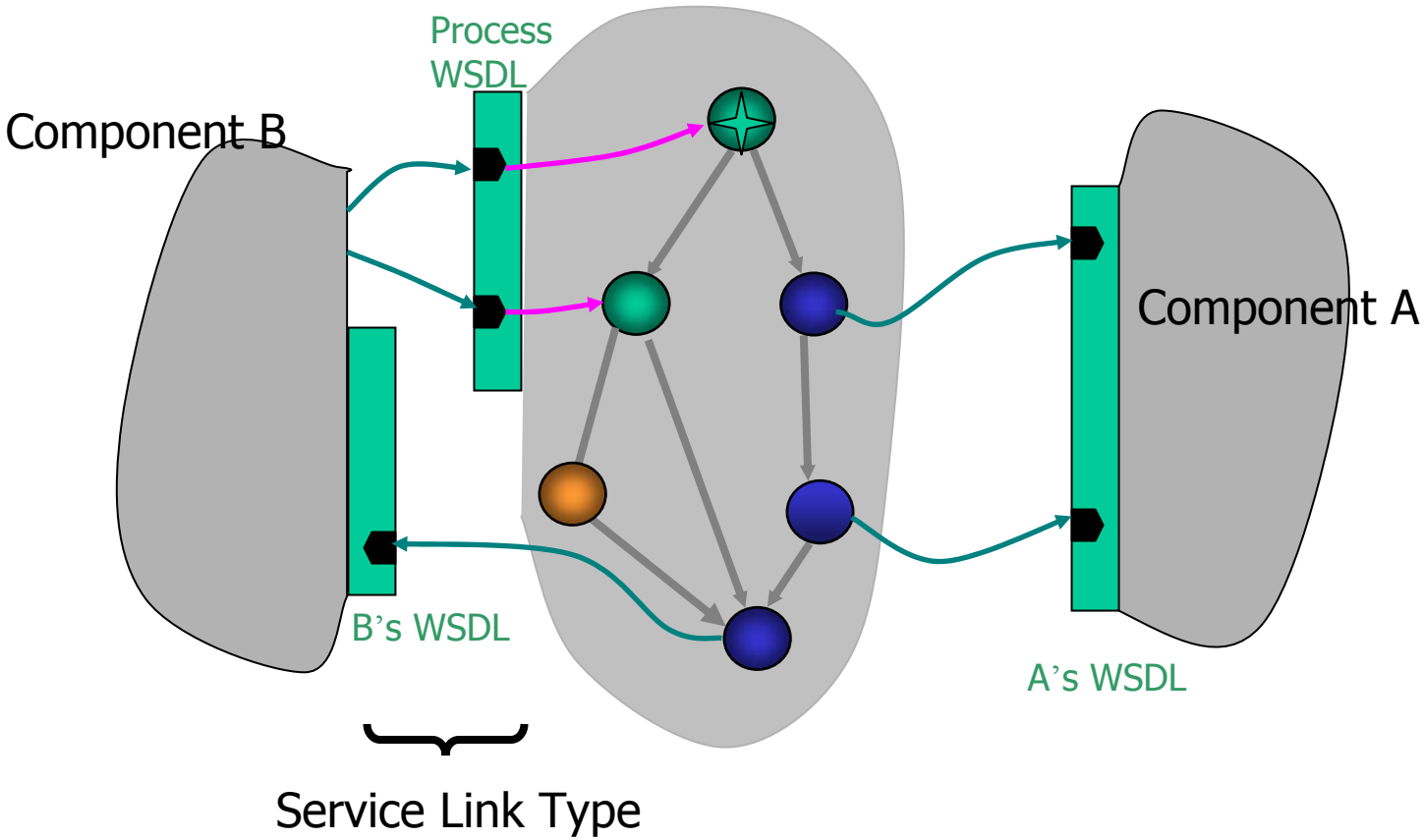
```
<variable name="..." message="..."/>*
```

BPEL Basic Activities

```
<invoke partner="..." portType="..." operation="..."
        inputVariable="..." [outputVariable="..."]/>
   <!-- process invokes an operation on a partner:      -->

<receive partner="..." portType="..." operation="..."
         variable ="..." [createInstance="..."]/>
   <!-- process receives invocation from a partner:      -->
<reply partner="..." portType="..." operation="..."
       variable ="..."/>
   <!-- process send reply message in partner invocation:     -->
<assign>
   <copy>
     <from variable="..."/> <to variable="..."/>
   </copy>+
</assign>
   <!– Data assignment between variables:      -->
```

BPEL Composition of Web Services



More Basic Activities

```
<throw faultName="..." faultvariable="..."/>
```

<!-- process detects processing error and switches into fault processing mode -->

```
<terminate/>
```

<!– pull the plug -->

```
<wait for="..."? until="..."?/>
```

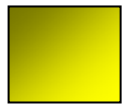
<!-- process execution stops for a specified amount of time-->

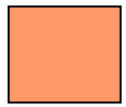
```
<empty>
```

<!– Do nothing; a convenience element -->

BPEL Structured Activities

- Structured activities





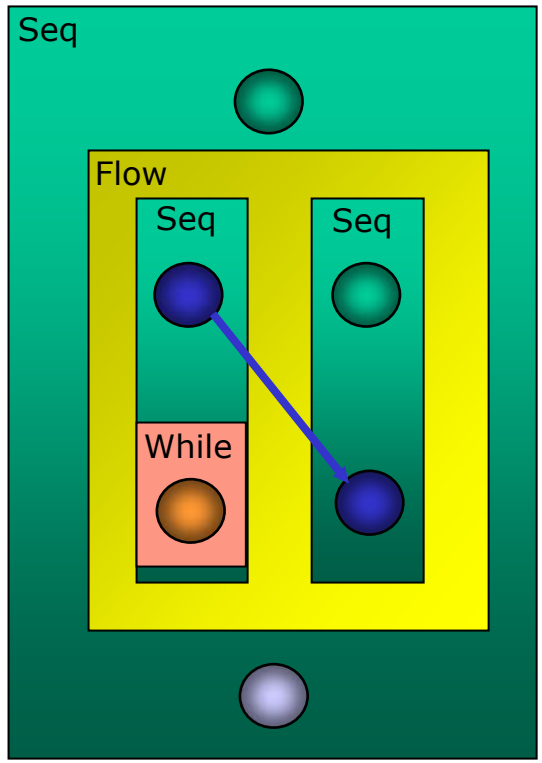
```
<sequence>
    <!– execute activities sequentially-->
<flow>
    <!– execute activities in parallel-->
<while>
    <!– iterate execution of activities until condition
    is violated-->
<pick>
    <!– several event activities (receive message, timer event) scheduled for
    execution in parallel; first one is selected and corresponding code executed. -->
```

- Links

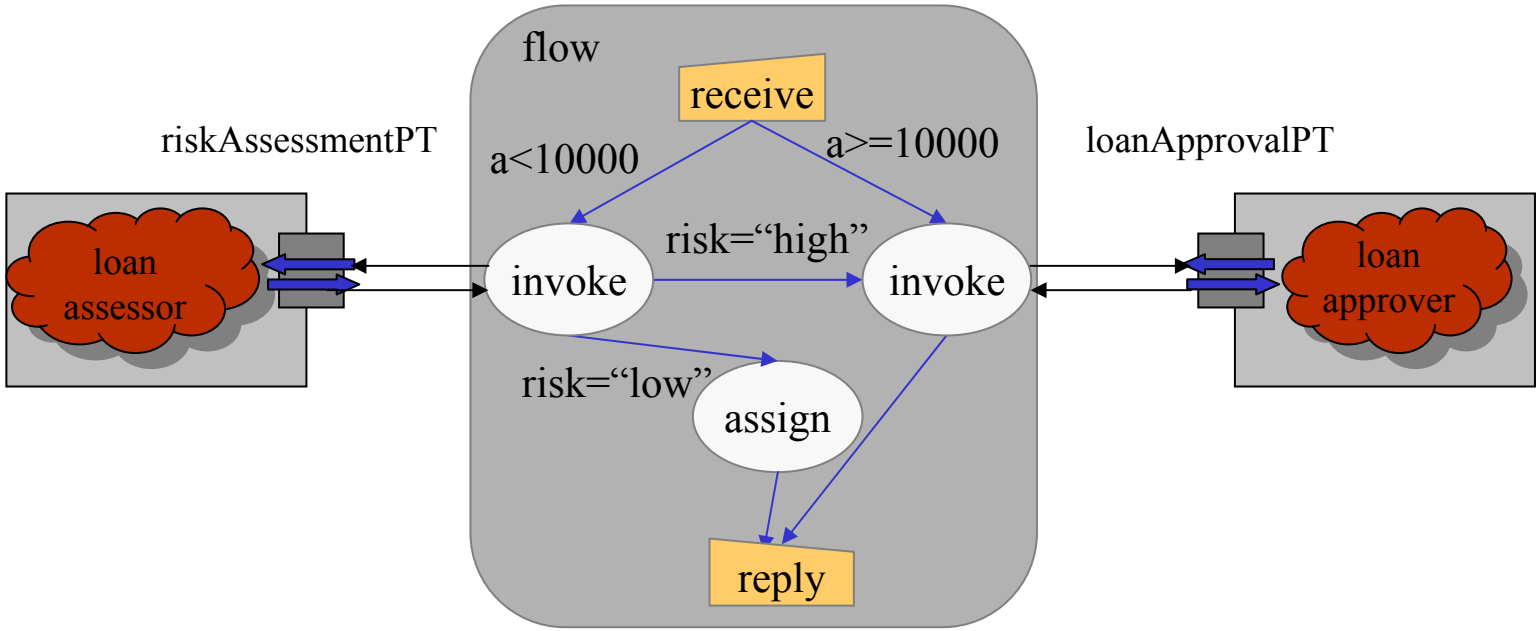
```
<link ...>
    <!– defines a control dependency between a
    source activity and a target -->
```

Nesting Structured Activities. Example

```
<sequence>
   <receive .../>
   <flow>
      <sequence>
         <invoke .../>
         <while ... >
            <assign> ... </assign>
         </while>
      </sequence>
      <sequence>
         <receive .../>
         <invoke ... >
      </sequence>
   </flow>
   <reply>
</sequence>
```

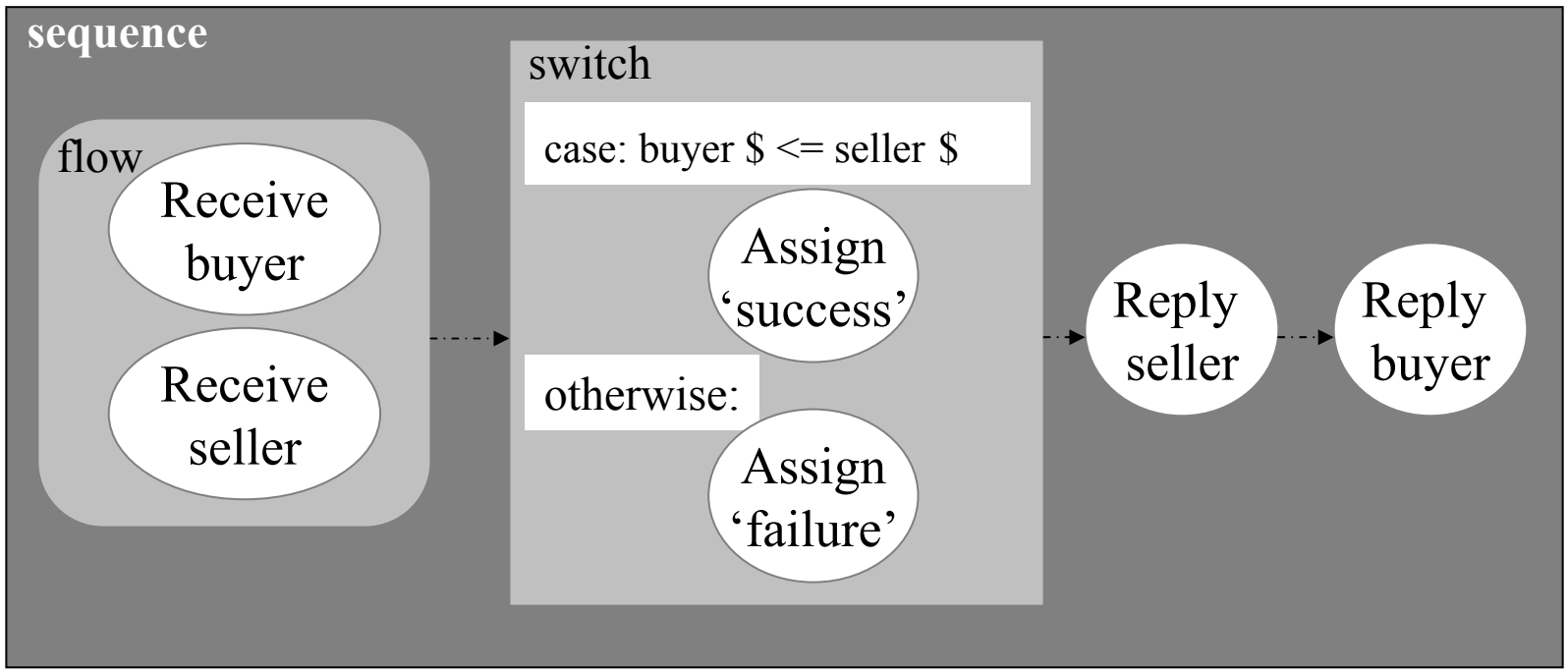


Graph-oriented Authoring Style



1. Customer asks for a loan, giving name and amount info.
   Two services are involved:
   - A risk assessor which can approve the loan if the risk is low
   - A loan approver which checks the name and decides whether to approve the loan.
2. The reply goes back to the customer.

“Structured” Authoring Style

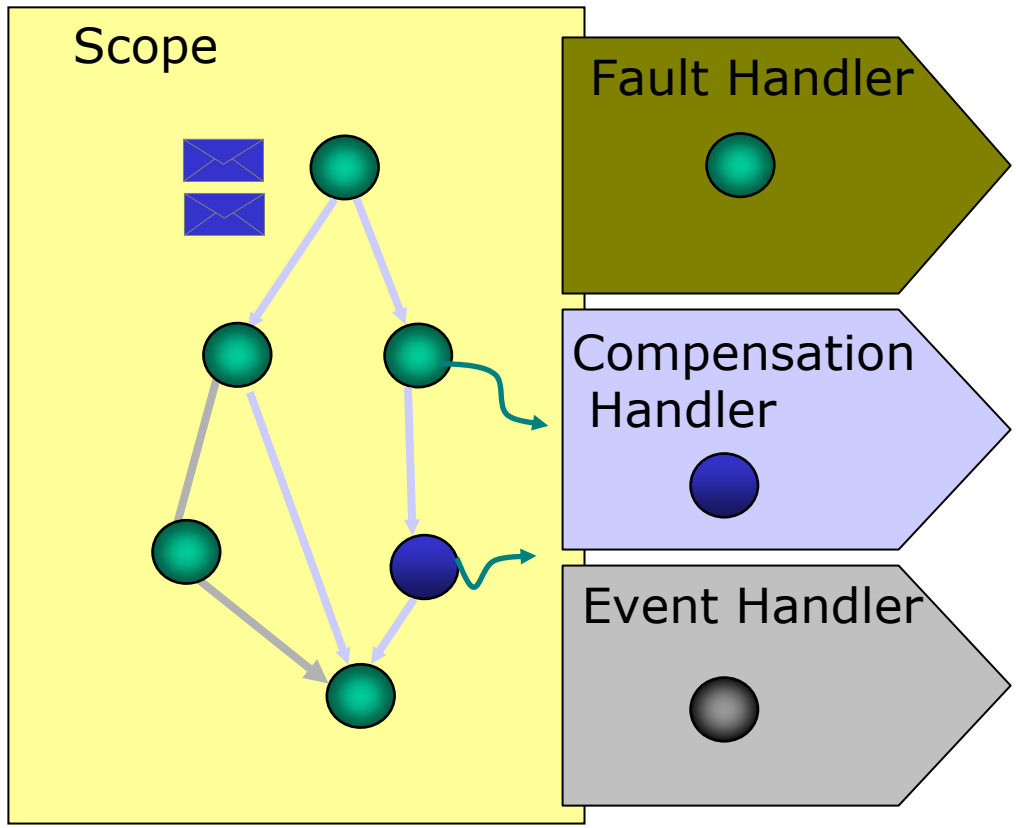


BPEL Handlers and Scopes

A **scope** is a set of (basic or structured) activities.

Each scope can have three types of **handlers** associated:

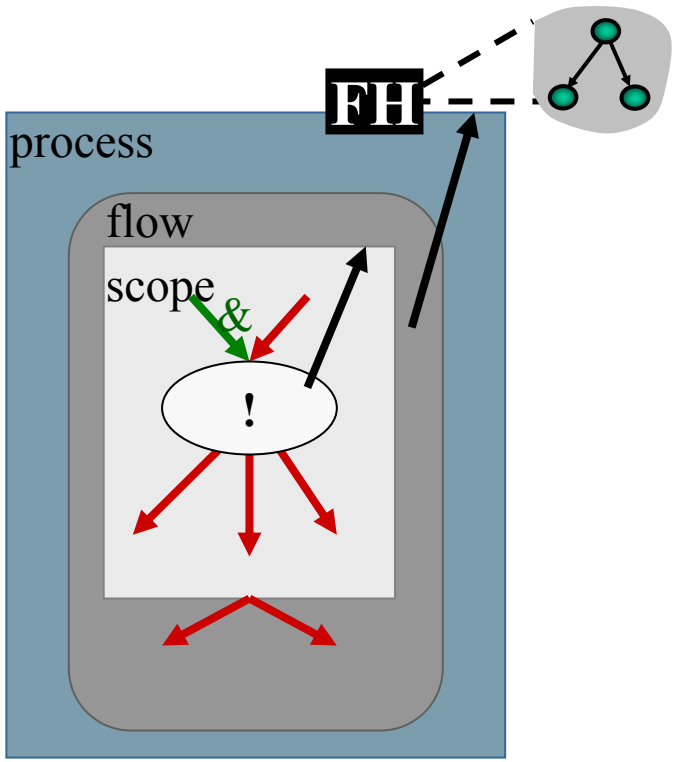
- **Fault handlers.** Many can be attached, for different fault types.
- **Compensation handlers.** A single compensation handler per scope.
- **Event Handlers**. Many can be attached, for different message types or alarms

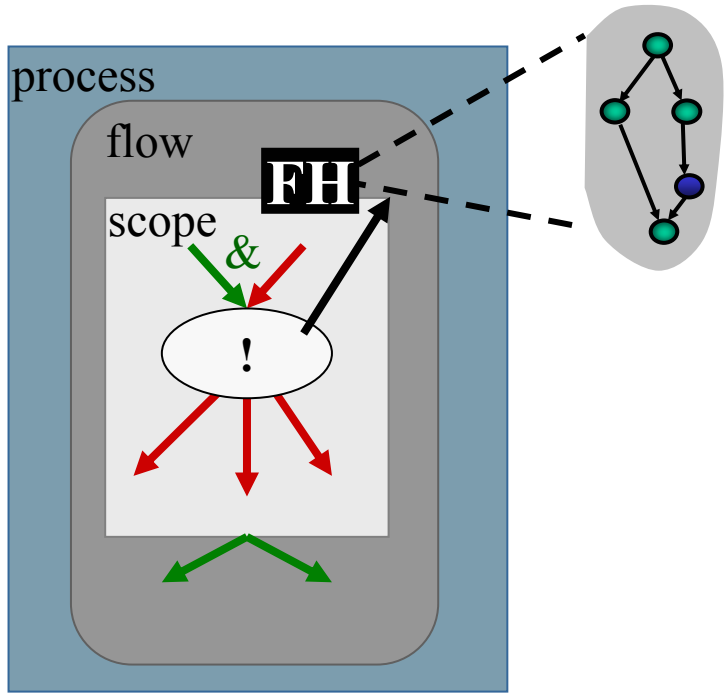


Fault and Compensation Handlers

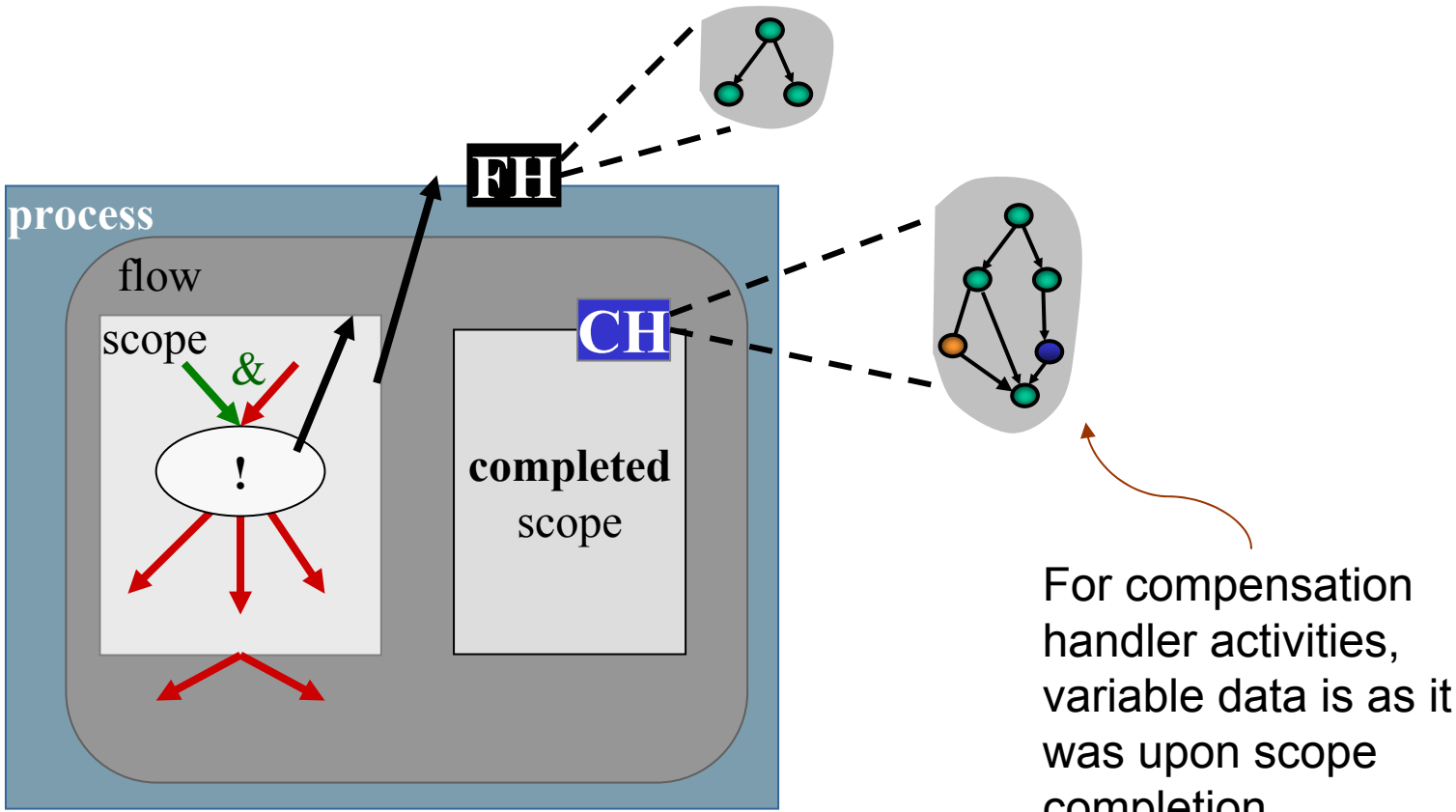
- A compensation handler is used to reverse the work performed by an already completed scope
  - A compensation handler can only be invoked by the fault handler or compensation handler of its immediate enclosing scope
- A fault handler defines alternate execution paths when a fault occurs within the scope.
- Typical scenario:
  1. Fault is thrown (retuned by invoke or explicitly by process)
  2. Execution of scope is terminated
  3. Appropriate fault handler located (with usual propagation semantics)
  4. Main execution is compensated to “undo” business effects of unfinished work.

Fault Handling





Compensation Example



For compensation handler activities, variable data is as it was upon scope completion

Event Handlers

- onMessage:
  - As long as a scope is active, messages received that match the handler info trigger the handler to run
  - The handler runs as many times as matching messages arrive and the scope is active.
- onAlarm:
  - More than one may be defined and when the time designated arrives the associated handler runs as long as the scope is still active
- No links can cross the boundaries of these handlers from/into the main flow.

What is Correlation?

- BPEL4WS can model many types of interactions:
  - simple stateless interactions
  - Stateful, long running, asynchronous interactions.
- Correlation sets (CSs) provide support for the latter:
  - CSs represent the data that is used to maintain the state of the interaction (a “conversation”).
  - At the process end of the interaction, CSs allow incoming messages to reach the right process instance.
- What is a correlation set?
  - A set of business data fields that capture the state of the interaction (“correlating business data”). For example: a “purchase order number”, a “customer id”, etc.
  - Each set is initialized once per pass through scope
  - Its values do not change in the course of the interaction.

Defining Correlation Sets

```
<correlationSet name="..." properties="..."/>
```

<!– A CS is a named set of properties. Properties are defined as WSDL extensibility elements: -->

```
<bpws:property name="..." type="..."/>
```

<!– A property has a simple XSD type and a global name (Qname) -->

```
<bpws:propertyAlias propertyName="..."
                    messageType="..." part="..."
                    query="..."/>
```

<!– A property is “mapped” to a field in a WSDL message type. The property can thus be found in the messages actually exchanged. Typically a property will be mapped to several different message types and carried on many interactions, across operations and portTypes -->

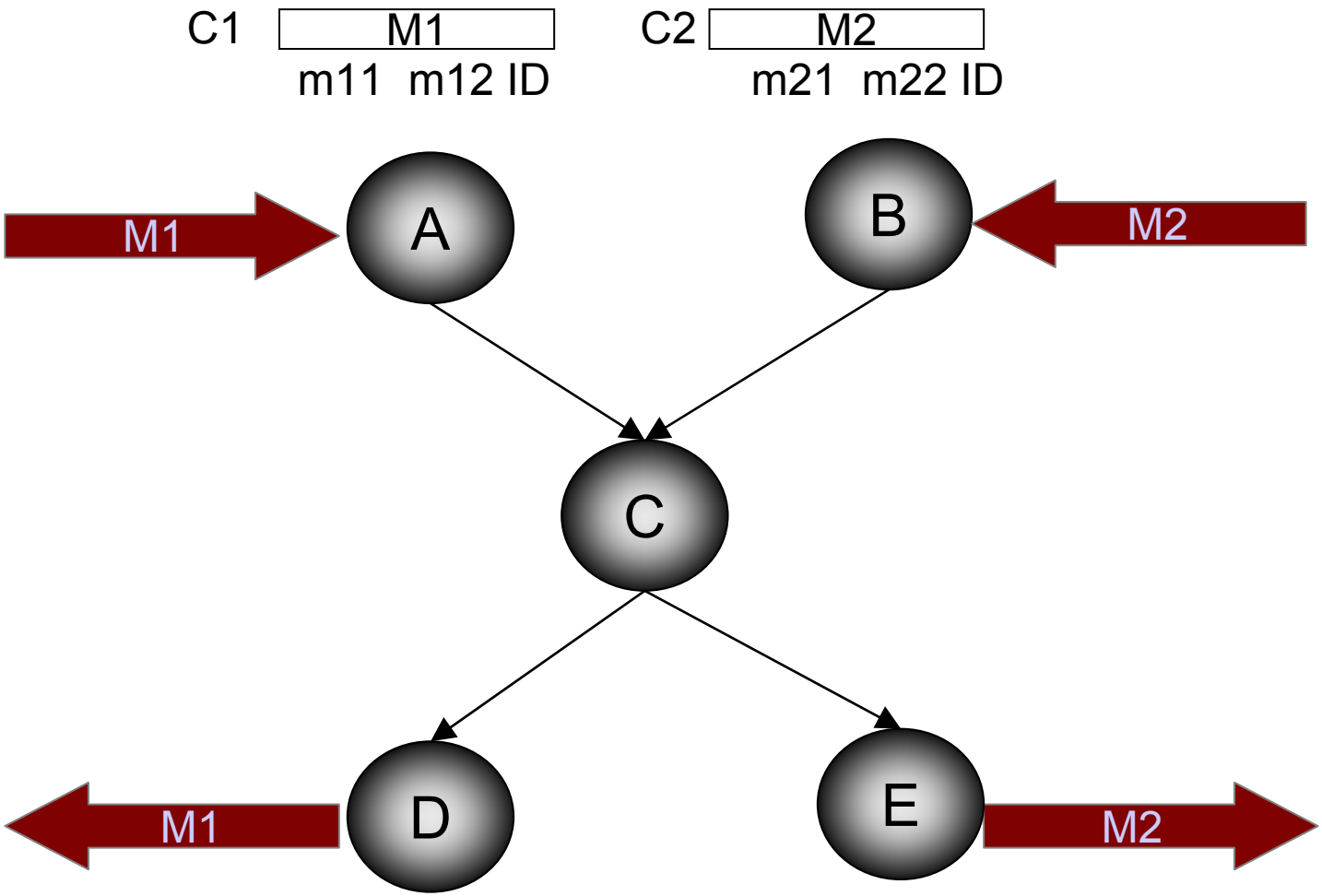
Using Correlation

```
<receive partner="..." operation="..." portType="..."
         variable="...">
    <correlations>
         <correlation set="PurchaseOrder"
                      initiate="yes"/>
     </correlations>
</receive>
```

<!– An input or output operation identifies which correlation sets apply to the messages received or sent. That CS will be used to assure that the message is related to the appropriate stateful interaction.

<!– A CS is initialized once in an interaction where the set appears with the “initiate” attribute set to “yes”. Its value may never be changed afterward in the same run of the scope -->

Multiple Start Correlation



BPEL4WS Status

- Submitted to OASIS.
- V1.0 Published August 10, 2002 by BEA, IBM, and Microsoft.
- V1.1 Published April 2003 by BEA, IBM, Microsoft, SAP, and Siebel Systems.
- Several Java implementations available.

Resources

- BPEL4WS 1.1:
  - http://www-106.ibm.com/developerworks/webservices/library/ws-bpel/
- BPWS4J Java Implementations:
  - http://www.alphaworks.ibm.com/tech/bpws4j
  - http://www.collaxa.com/
- developerWorks articles on BPEL:
  - http://www-106.ibm.com/developerworks/library/ws-bpelcol.html