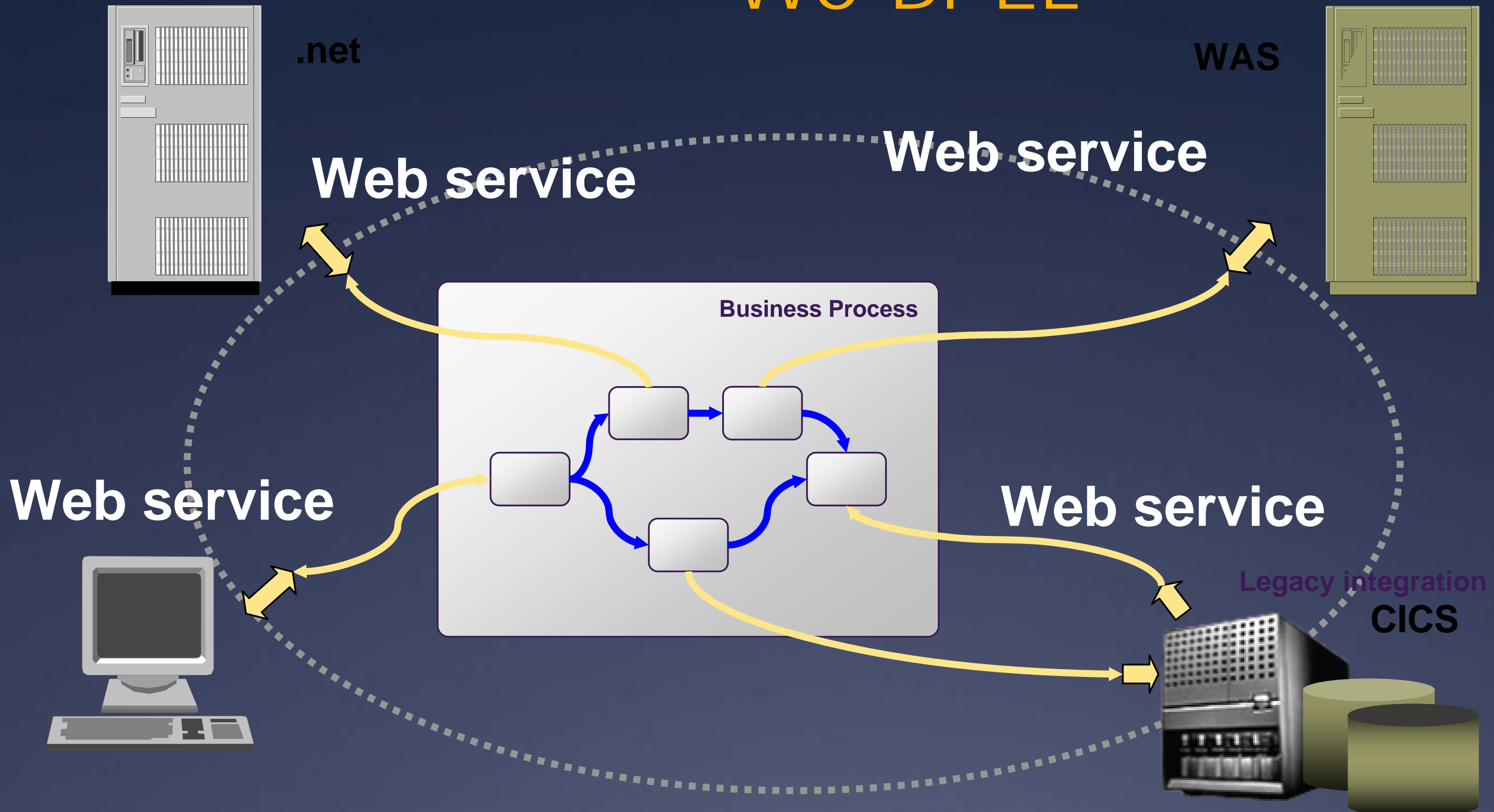
WS-BPEL

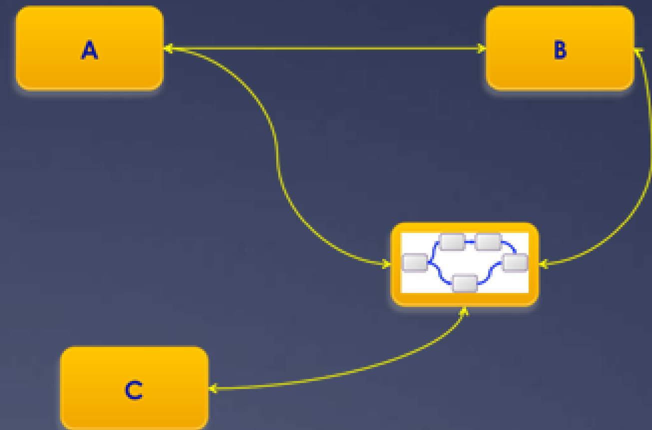
WS-BPEL



Enabling users to describe business process activities as Web services and define how they can be connected to accomplish specific tasks

What would you do?

* Control flow (procedural?)
* Data passing/data flow
* Events, Timeouts
* Parallelism
* Exception handling
* Transaction
* Expose as a service
* Correlation
* Late-binding



WS-BPEL Specifications

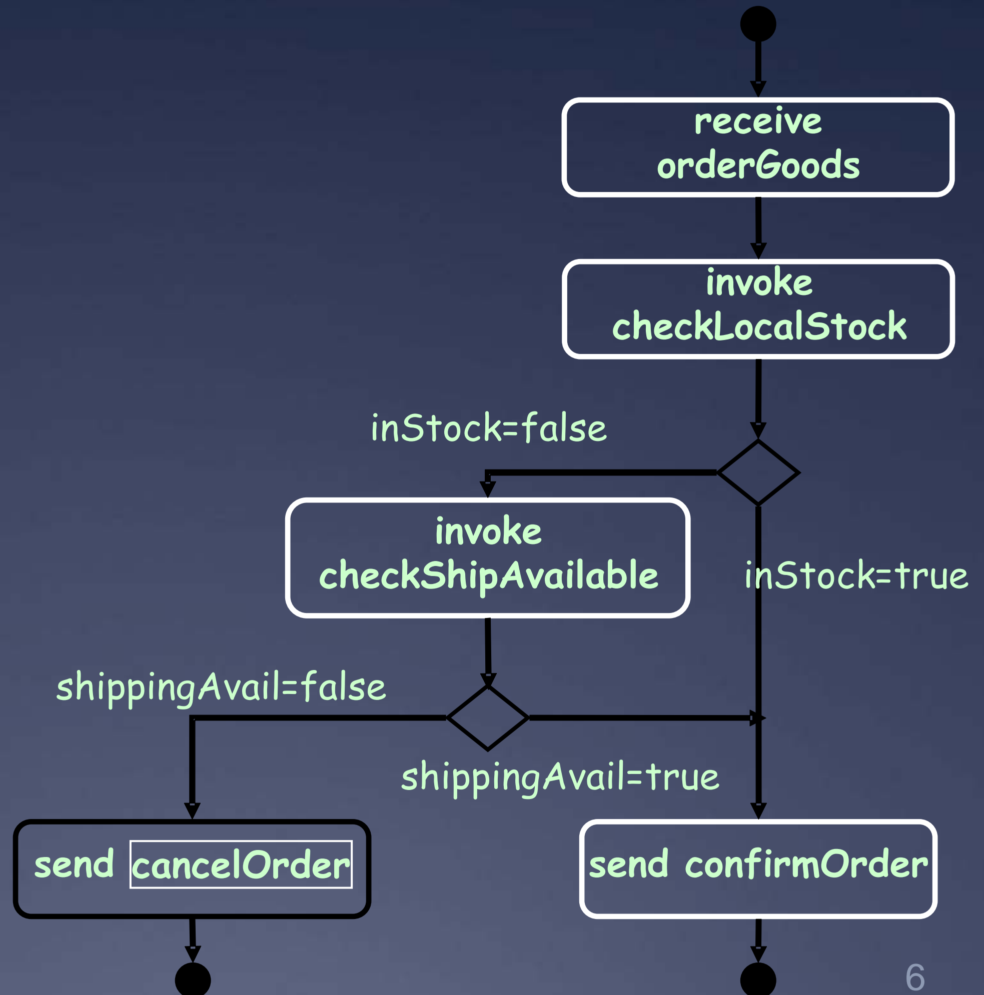
* BPEL4WS 1.0 (7/2002)
  * Original proposal from BEA, IBM, Microsoft
  * Combined ideas from IBM's WSFL and Microsoft's XLANG
* BPEL4WS 1.1 (5/2003)
  * Revised proposal submitted to OASIS
  * With additional contributions from SAP and Siebel
* **WS-BPEL 2.0 - April 11, 2007**
  * Oasis standard
  * **Primer - May 11, 2007**

WS-BPEL in the WS-* Stack

You are here

Stack			Category
WS-BPEL			Business Processes
WSDL, Policy, UDDI, Inspection			Description
Security	Reliable Messaging	Transactions / Coordination	Quality Of Service
SOAP (Logical Messaging)		Other protocols	Transport and Encoding
XML, Encoding		Other services	

Service composition? Composition of operations



Basic Activities

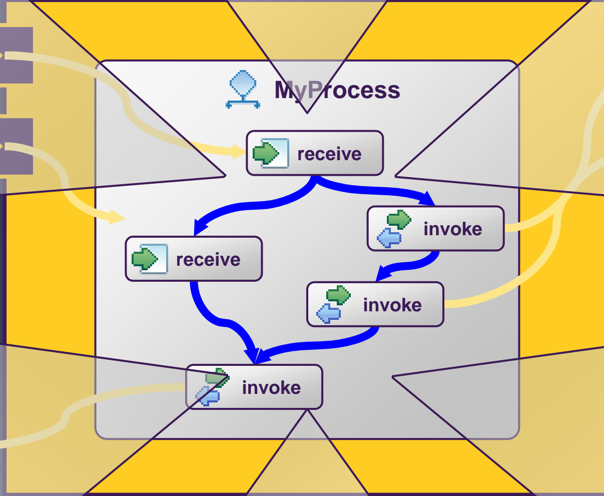
receive
exit
reply
throw
invoke
rethrow
assign
wait
validate
compensate
empty
compensateScope
extensionActivity

Variables

WSDL Message
42
XML Schema Type
XML Schema Element

Partner Links

partner link
partner link
Port Type 1
Partner Link Type
Port Type 2



Structured Activities

flow
pick
sequence
forEach
if-else
while
repeatUntil
scope

Handlers

event handler
fault handler
compensation handler
termination handler

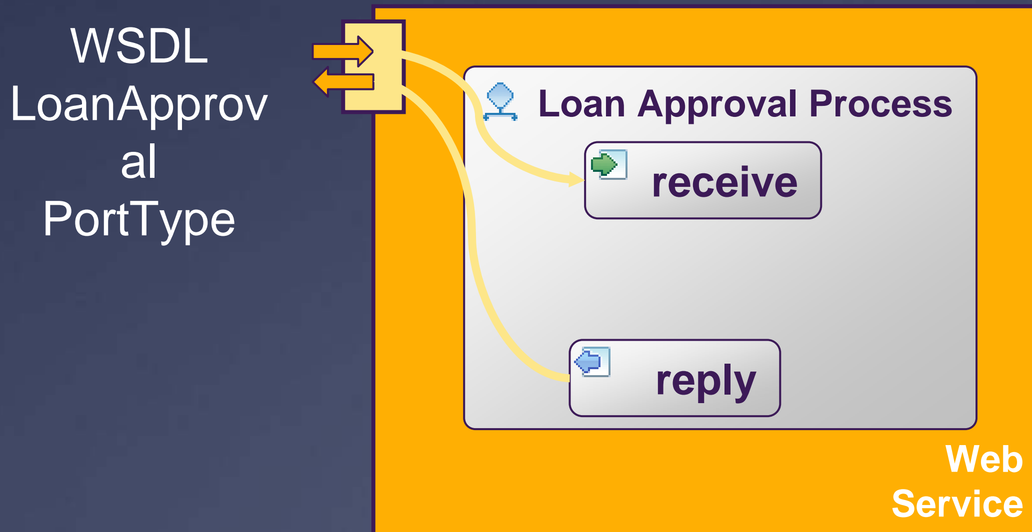
Properties Correlation Sets

Property 1
Property 2

BPEL and WSDL

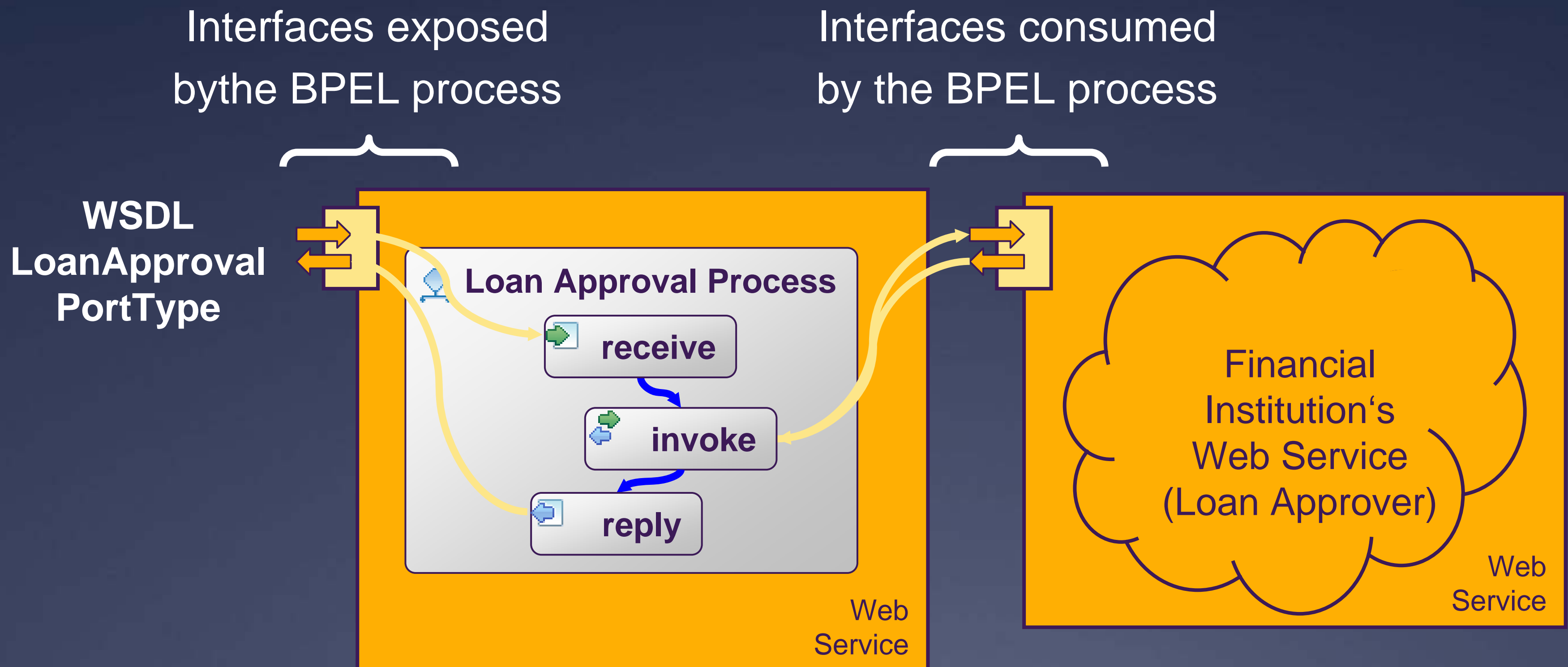
- BPEL processes are exposed as WSDL 1.1 services
  - Message exchanges map to WSDL operations
  - WSDL can be derived from partner definitions and the role played by the process in interactions with partners

WSDL
LoanApprov
al
PortType

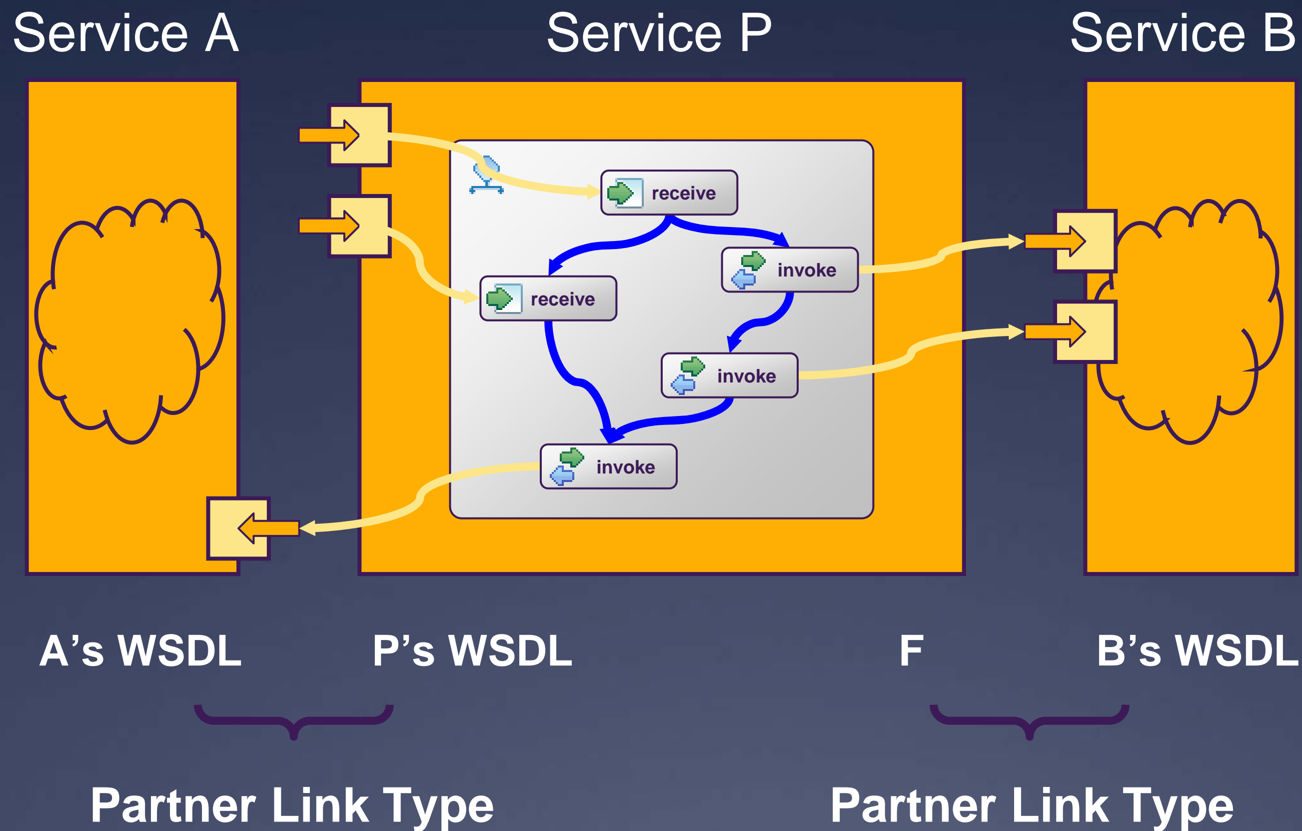


Recursive Composition

* BPEL processes interact with WSDL services exposed by business partners



Distributed / P2P (compare with WF)



Partner Links

* Partner link: instance of typed connector
  * Partner link type specifies required and/or provided portTypes (NOT the actual service)
  * Channel along which a peer-to-peer conversation with a partner takes place

I declare what
I need and
what I offer

Link Types

```
<plnk:partnerLinkType name="purchasingLT">
<plnk:role name="purchaseService"
portType="pos:purchaseOrderPT" />
</plnk:partnerLinkType>

<plnk:partnerLinkType name="invoicingLT">
<plnk:role name="invoiceService"
portType="pos:computePricePT" />
<plnk:role name="invoiceRequester"
portType="pos:invoiceCallbackPT" />
</plnk:partnerLinkType>
```

Partner links

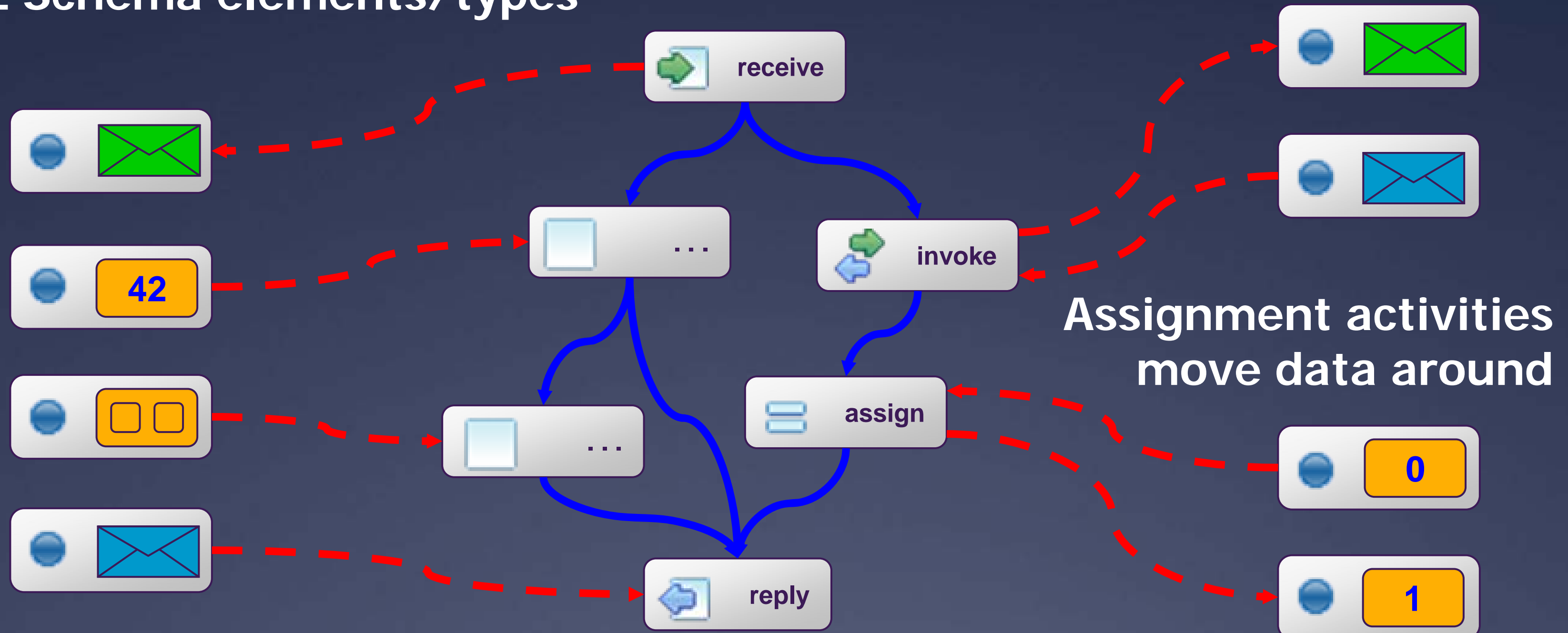
```
<partnerLinks>
<partnerLink name="purchasing"
partnerLinkType="lns:purchasingLT" myRole="purchaseService" />

<partnerLink name="invoicing"partnerLinkType="lns:invoicingLT"
myRole="invoiceRequester" partnerRole="invoiceService" />

<partnerLink name="shipping" partnerLinkType="lns:shippingLT"
myRole="shippingRequester" partnerRole="shippingService" />

<partnerLink name="scheduling” partnerLinkType="lns:schedulingLT"
partnerRole="schedulingService" />
</partnerLinks>
```

BPEL Data Model: Variables

Scoped variables typed as
WSDL messages or
XML Schema elements/types

Activities' input and output
kept in scoped variables

Assignment activities
move data around



Variables

```
<variables>
<variable name="PO" messageType="lns:POMessage" />
<variable name="Invoice" messageType="lns:InvMessage" />
…
</variables>
```

Process model

* A little confusing….
  * Combination of structured and flow approach

Main Basic Activities

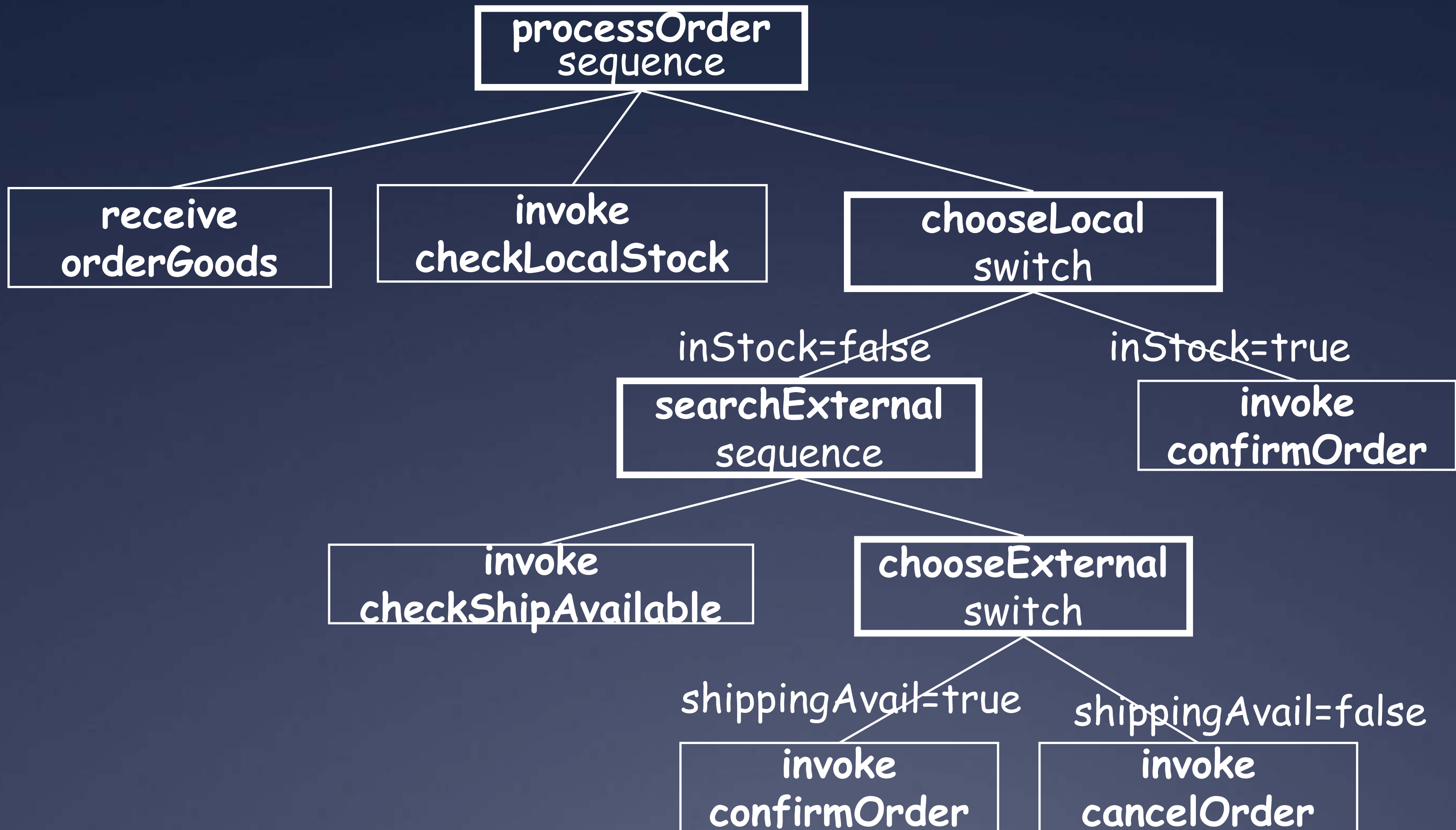
receive Do a blocking wait for a matching message to arrive

reply Send a message in reply to a formerly received message

invoke Invoke a one-way or request-response operation

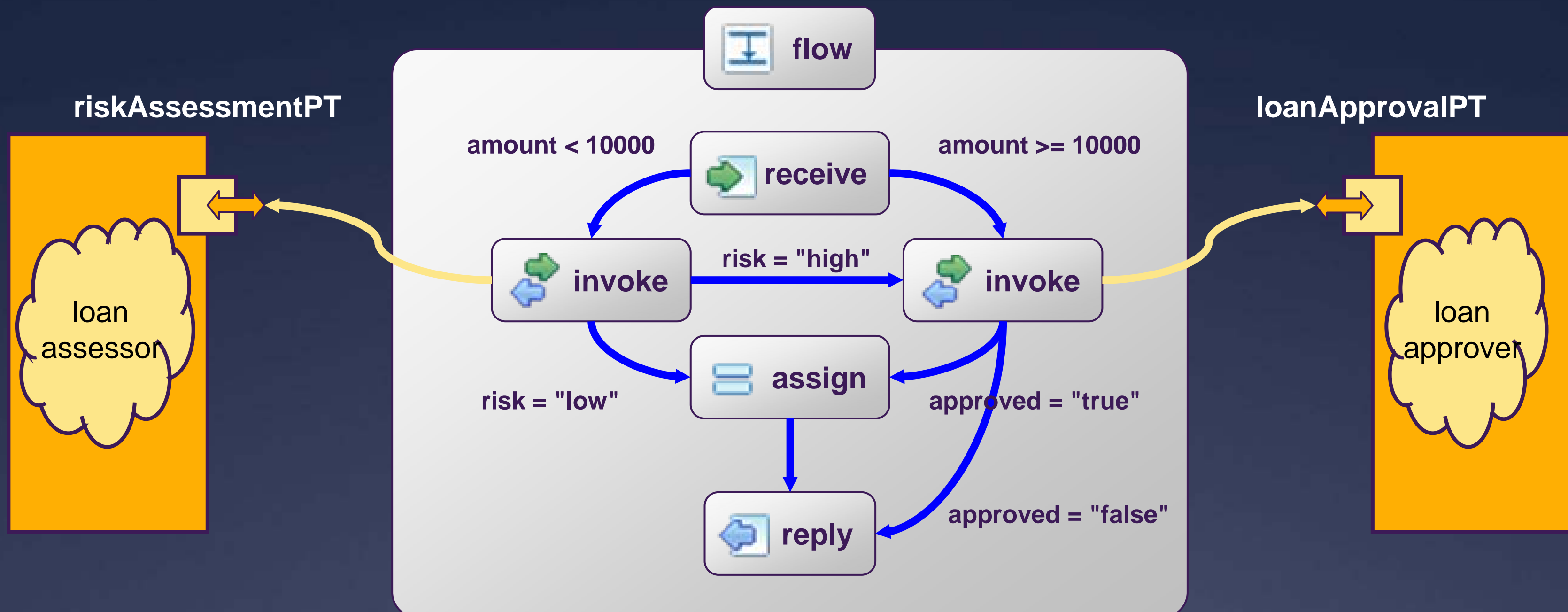
assign updates values of variables or partner links

Example of structured model



copyright Springer-Verlag, 2004

Graph-Oriented Authoring Style



1. A customer asks for a loan, providing name and amount info
2. Two services are involved:
   a) A risk assessor which can approve the loan if the risk is low
   b) A loan approver which checks the name and approves/disapproves the loan
3. The reply is returned to the customer

Receive

```
<receivepartnerLink="purchasing" portType="lns:purchaseOrderPT"
        operation="sendPurchaseOrder" variable="PO"
createInstance="yes">
<documentation>Receive Purchase Order</documentation>
</receive>
```

>>>COMPLETION LOGIC<<<

Reply

```
<replypartnerLink="purchasing" portType="lns:purchaseOrderPT"
      operation="sendPurchaseOrder" variable="Invoice">
<documentation>Invoice Processing</documentation>
</reply>
```

To a *receive*, *onMessage*, *onEvent*

IMA+Reply-> WSDL request/response (but, I do not reply right away, that's why it is two activities)

Invoke

```
<invokepartnerLink="shipping" portType="lns:shippingPT"
        operation="requestShipping"
inputVariable="shippingRequest"
outputVariable="shippingInfo">
<documentation>Decide On Shipper
</documentation>
…
</invoke>
```

One way or request/response

More Basic Activities…

Wait for a given time period or until a certain time has passed

Immediately terminate execution of a business process instance

```
<wait>
<until>'2009-12-24T18:00+01:00'</until>
</wait>
```

```
<wait>
<for>’P1Y10M28DT10H37M46S’</for>
</wait>
```

Structured Activities

Contained activities are executed in parallel, partially ordered through control links

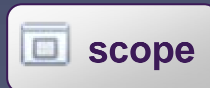
Block and wait for a suitable message to arrive (or time out)

Contained activity is performed sequentially or in parallel, controlled by a specified counter variable

Select exactly one branch of activity from a set of choices

Contained activities are performed sequentially in lexical order

Contained activity is repeated while a predicate holds

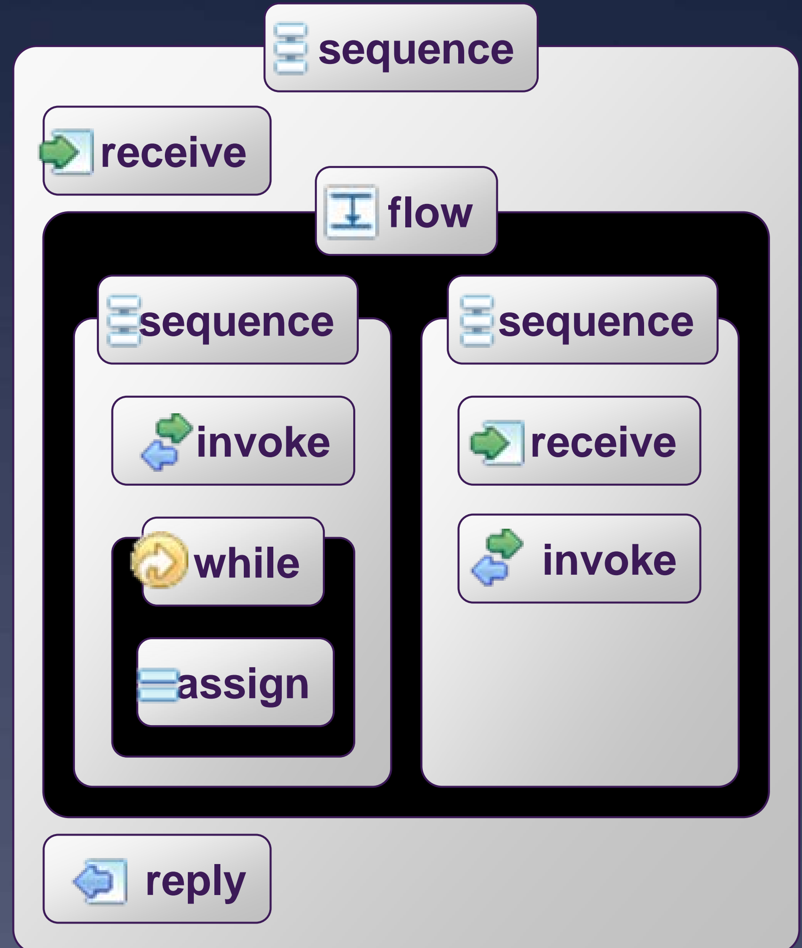


Associate contained activity with its own local variables, fault handlers, compensation handler, and event handlers

Contained activity is repeated until a predicate holds

Nesting Structured Activities

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



Sequence

sequence

```
<sequence>
<wait>
<until>'2009-12-24T18:00+01:00'</until>
</wait>
<invoke partnerLink="CallServer" portType="AutomaticPhoneCall"
     operation="TextToSpeech" inputVariable="seasonGreetings" />
</sequence>
```

If then else

```
<ifxmlns:inventory="http://supply-chain.org/inventory"
xmlns:FLT="http://example.com/faults">
<condition>
bpel:getVariableProperty('stockResult','inventory:level') > 100
</condition>
<flow>
<!-- perform fulfillment work -->
</flow>
<elseif>
      …
</elseif>
<else>
<throw faultName="FLT:ItemDiscontinued" />
</else>
</if>
```

loops

while

repeatUntil

forEach

```
<while>
<condition>$orderDetails> 100</condition>
<scope>...</scope>
</while>
```

```
<forEachcounterName="po" parallel="yes">
<startCounterValue>1</startCounterValue>
<finalCounterValue>
count($purchaseOrder/itemsList/*)
</finalCounterValue>
<scope>…
</scope>
</forEach>
```

Parallelism and completions in forEach

* A parallel forEach is like having a "flow" (parallel execution) with N+1 copies of the same activity
  * Each will have a counter variable i with a different value
* A default Message Exchange property, which keeps together the correct pair of receive/replies or OnMessage/replies, is automatically and implicitly declared in the scope.

Let the fun begin… *ForEach* completion logic

* Completion is determined by a *completion condition*
  * Evaluated at the start of the loop
  * Completes after M out of N activities
  * prevents some of the children from executing (in the serial case), or forces early termination of some of the children (in the parallel case)
* *countCompletedBranchesOnly*
* *completionConditionFailure* (automatic, immediate failure of the loop if not enough instances left)

```
<completionCondition>
<branches countCompletedBranchesOnly="yes"> 10 <branches>
</completionCondition>
```

Flow (concurrency)

```
<sequence>
<flow>
<invoke partnerLink="Seller" name=“A”... />
<invoke partnerLink="Shipper" name=“B” ... />
</flow>
<invoke partnerLink="Bank" name="transferMoney" ... />
</sequence>
```

Links within flows (synch dependencies)

Links across scopes

```
<flow>
<links>
<link name="XtoY" />
<link name="CtoD" />
</links>
<sequence name="X">
<sources><source linkName="XtoY" /></sources>
<invoke name="A" ... />
<invoke name="B" ... />
</sequence>
<sequence name="Y">
<targets><target linkName="XtoY" /></targets>
<receive name="C" ...>
<sources><source linkName="CtoD" /></sources>
</receive>
<invoke name="E" ... />
</sequence>
<invoke name="D" ...>
<targets><target linkName="CtoD" /></targets>
</invoke>
</flow>
```

The “fun” part of links…

* Links vs goto…
  * Free vs block-structured
* Join conditions on targets (bool exp)
  * Default is OR

More fun with links

- Link status: 3 values (true, false, unset)
  - Set to "unset" at start of the flow
- When an activity terminates, the link status is set to true or false
  - Done by evaluating the transition condition of the link
  - DEPENDS ON RACE CONDITIONS. Trans cond may depend on variables set by parallel activities…

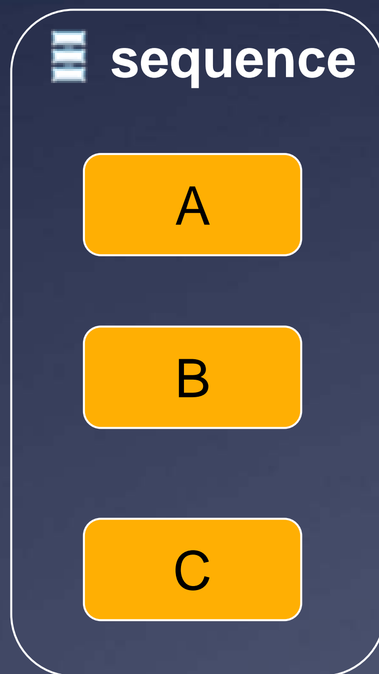
Even more fun

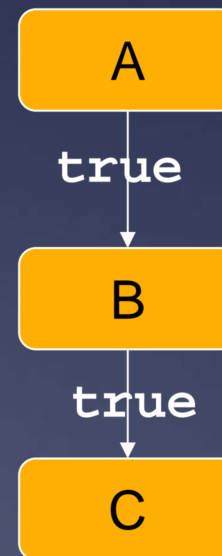
- IF an activity has incoming links, theni) all link statuses must be setand ii) join conditions must be evaluated before it can start
- An activity completes. Now, the link status is changed. Check target
  - Is it ready as per the process structure?
  - Are all input links set (true or false)?
- If so, evaljoinCondition
  - If true, start
  - **If false, throw a fault**
    - **unless `suppressJoinFailure` set**

Death path elimination

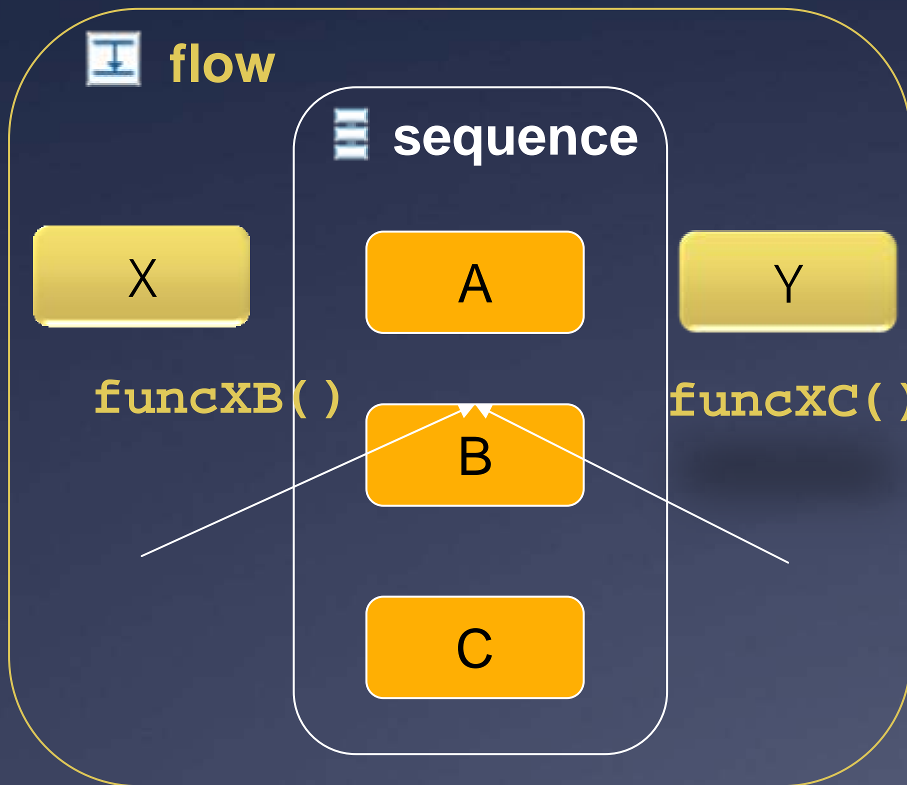
* Needed because of the rule that all link statuses should be set
* When a target activity is not performed due to the value of the <joinCondition> (implicit or explicit) being false, its outgoing links MUST be assigned a false status
* If, during the performance of structured activity A, the semantics of A dictate that activity B nested within A will not be performed as part of the execution of A, then the status of all outgoing links from B MUST be set to false.

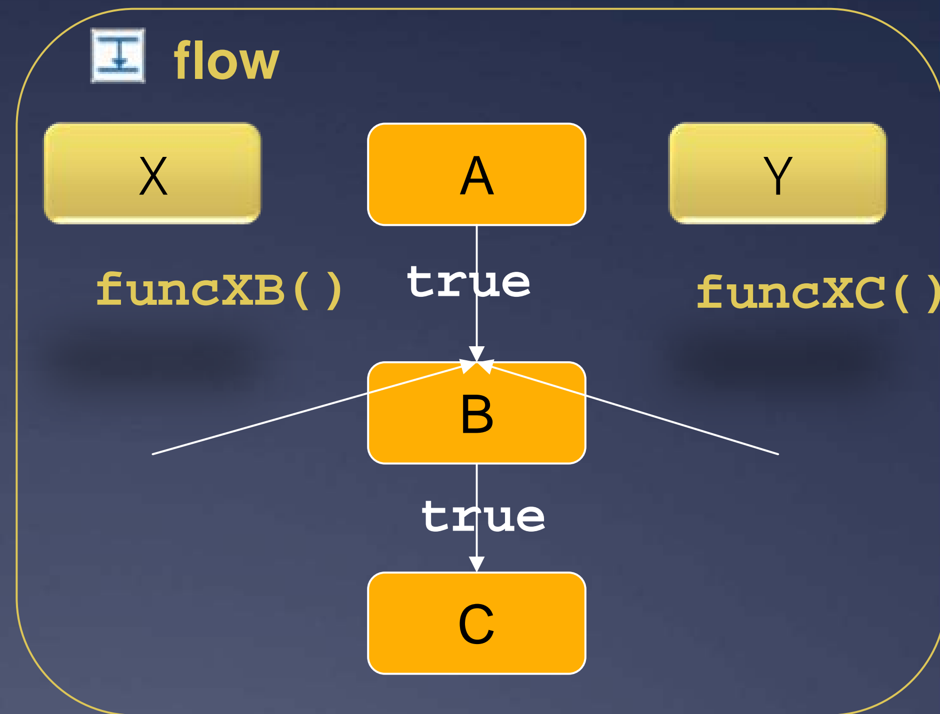
What's the difference?



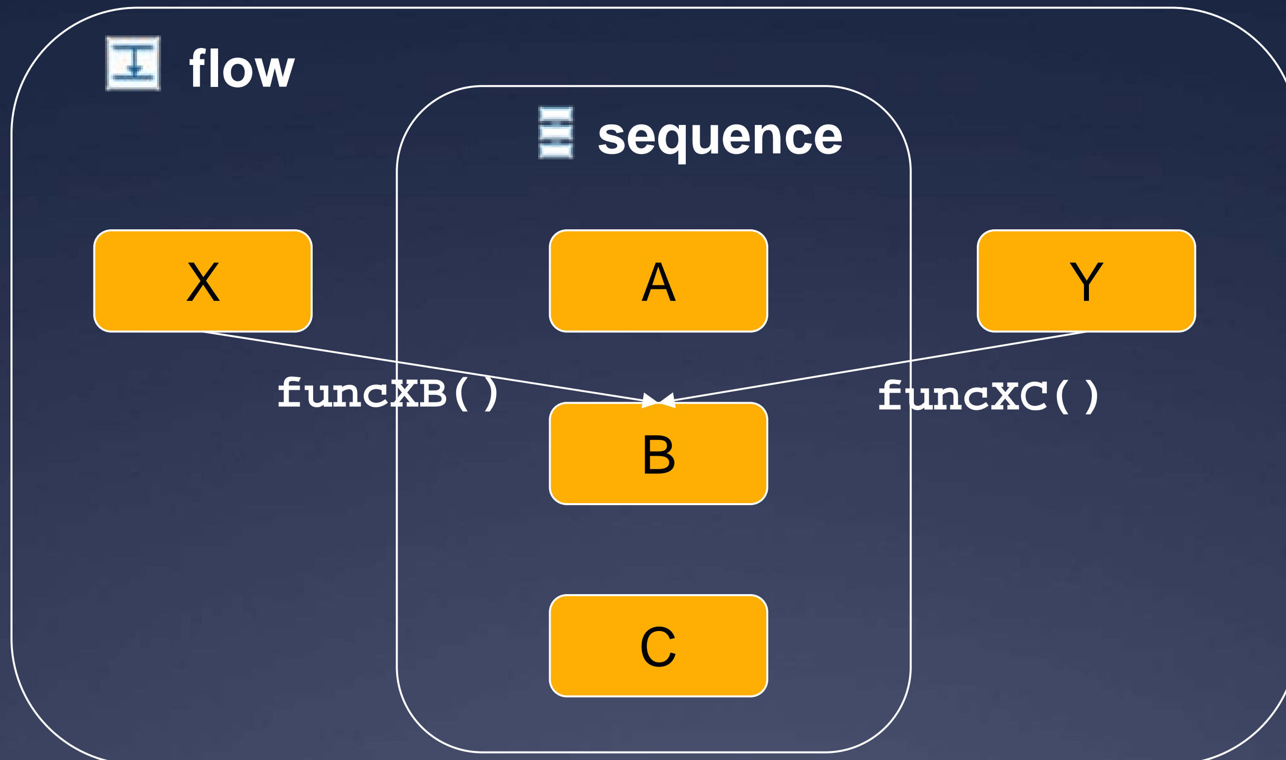


What’s the difference?



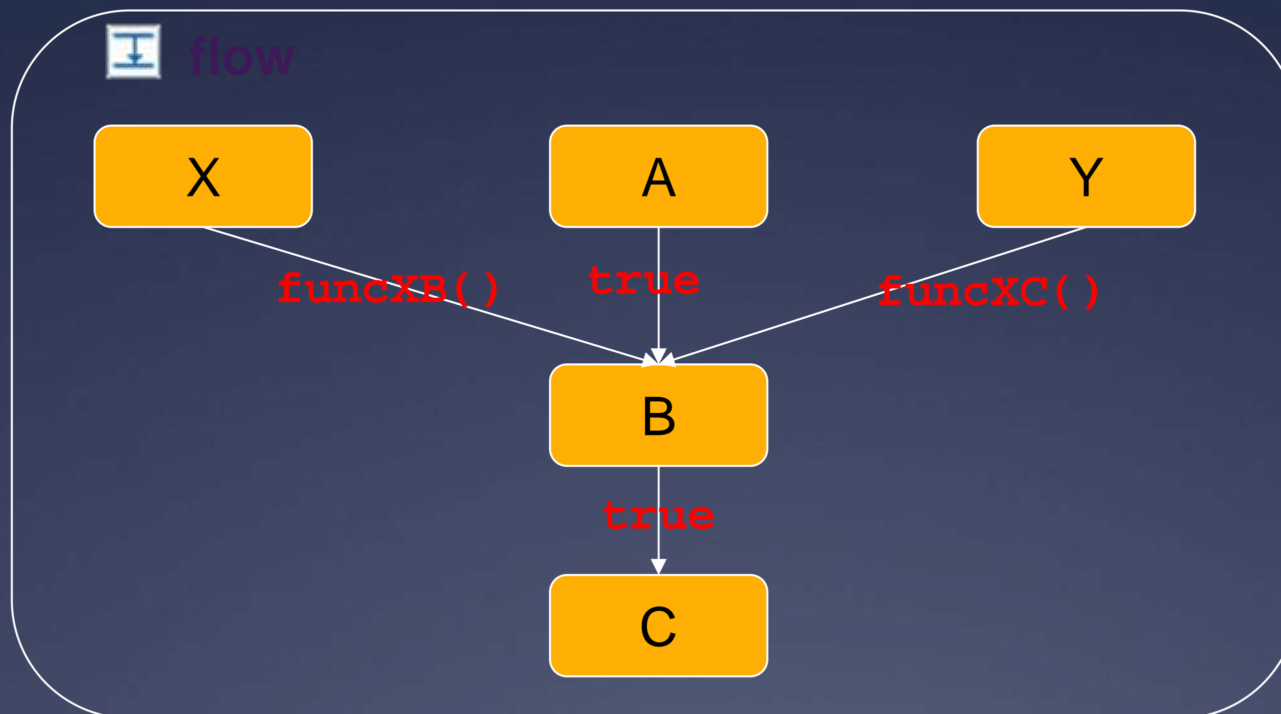


Links, structured activities, and join failures



- B cannot start until the status of links from X and Y is determined and the join condition (OR) is evaluated.
- If both funcXB() and funcYB() evaluate to false, the standard fault bpel:joinFailure will be thrown. **The <flow> is interrupted and neither B nor C will be performed**.
- If attribute suppressJoinFailure of the <flow> activity is set to yes, then **B will be skipped but C will be executed** because the bpel:joinFailure will be suppressed.

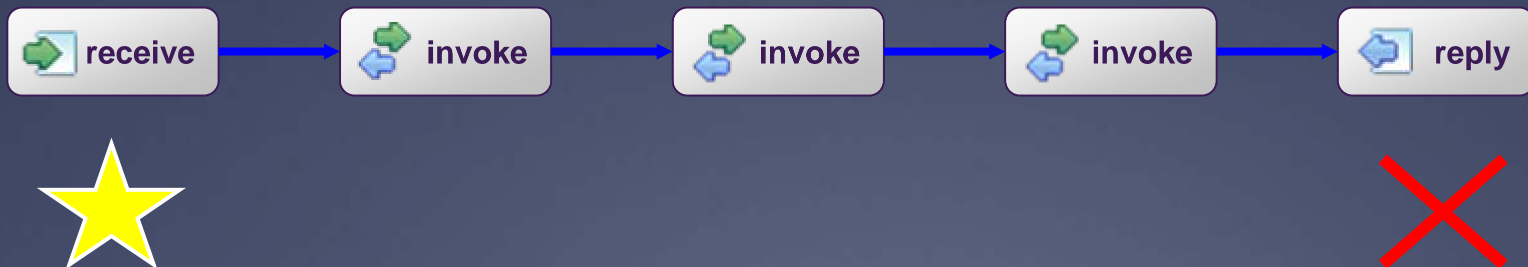
Links, structured activities,and join failures (II)



If done trough links instead of sequence, B starts regardless of the value of the conds of X and Y!!!! Observe the subtle differences in the use of links vs structured activities to define the ABC sequence….

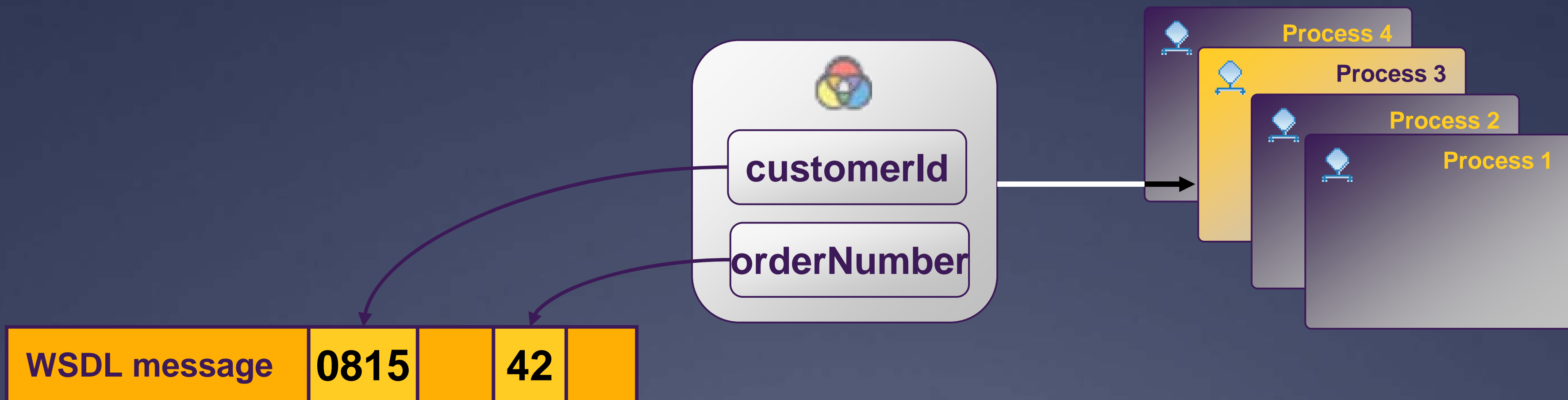
Process Instance Lifecycle

* Business processes defined in BPEL represent **stateful Web services**
  * When a process is started, a new instance is created
  * The destruction of BPEL process instances is by design implicit



Properties and Correlation

* Messages in long-running conversations are correlated to the correct process instance
  * Typed properties defined in WSDL are named and mapped (aliased) to parts of several WSDL messages used by the process



Correlation sets

* Bound on specially marked send or receive
  * Initiated once, then never changes value in the scope
* Other activities can refer to it for correlation

```
<correlationSetsxmlns:cor="http://example.com/supplyCorrelation">

<!-- Order numbers are particular to a customer,
        this set is carried in application data -->
<correlationSet name="PurchaseOrder"
      properties="cor:customerIDcor:orderNumber" />

<!-- Invoice numbers are particular to a vendor,
      this set is carried in application data -->
<correlationSet name="Invoice"
      properties="cor:vendorIDcor:invoiceNumber" />

</correlationSets>
```

```
<receive partnerLink="Buyer" portType="SP:PurchasingPT"
   operation="PurchaseRequest" variable="PO">
<correlations>
<correlation set="PurchaseOrder" initiate="yes" />
</correlations>
</receive>
```

Scopes, exceptions, and transactions in WS-BPEL

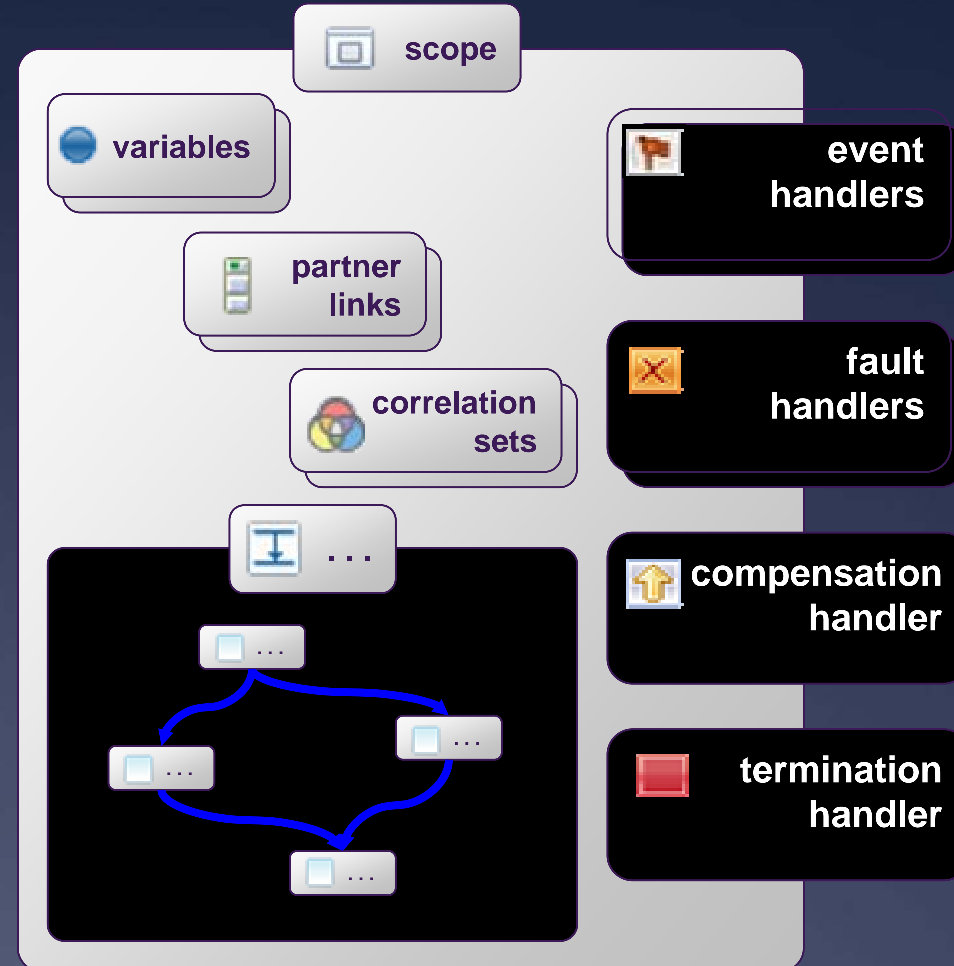
Scopes

* provide the context which influences the execution behavior of its enclosed activities.
* context includes variables, partner links, message exchanges, correlation sets, event handlers, fault handlers, a compensation handler, and a termination handler
* Each scope has a required primary activity that defines its normal behavior.
  * can be a complex structured activity, with many nested activities to arbitrary depth.

Scopes and Handlers

* Scope
  * Local variables
  * Local partner links
  * Local correlation sets
  * Set of activities (basic or structured)

* Handlers
  * Event handlers
    * Message events or timer events (deadline or duration)
  * Fault handlers
    * Dealing with different exceptional situations (internal faults)
  * Compensation handler
    * Undoing persisted effects of already completed activities
  * Termination handler
    * Dealing with forced scope termination (external faults)



Event handlers

* Defines how to respond to normal events
  * Messages receipt (similar to receive, BUT it does not prevent scope termination)
  * Alarms: for (period), until (deadline), repeatEvery (repeat every T while parent scope is active)
* Upon event occurrence, a scope is executed

Messages and alarms

```
<onMessagepartnerLink="buyer"
portType="orderEntry" operation="inputLineItem" variable="lineItem">
<!-- activity to add line item to order -->
</onMessage>
```

```
<!-- set an alarm to go off
      3 days and 10 hours after the last order line -->
<onAlarm>
<for>'P3DT10H'</for>
<!-- handle timeout for order completion -->
</onAlarm>
```

On and pick

```
<pick>
<onMessagepartnerLink="buyer"
portType="orderEntry” operation="inputLineItem” variable="lineItem">
<!-- activity to add line item to order -->
</onMessage>
<onMessagepartnerLink="buyer" portType="orderEntry”
          operation="orderComplete" variable="completionDetail">
<!-- activity to perform order completion -->
</onMessage>
<!-- set an alarm to go off
       3 days and 10 hours after the last order line -->
<onAlarm>
<for>'P3DT10H'</for>
<!-- handle timeout for order completion -->
</onAlarm>
</pick>
```

Can use pick to create instances

Exception handling

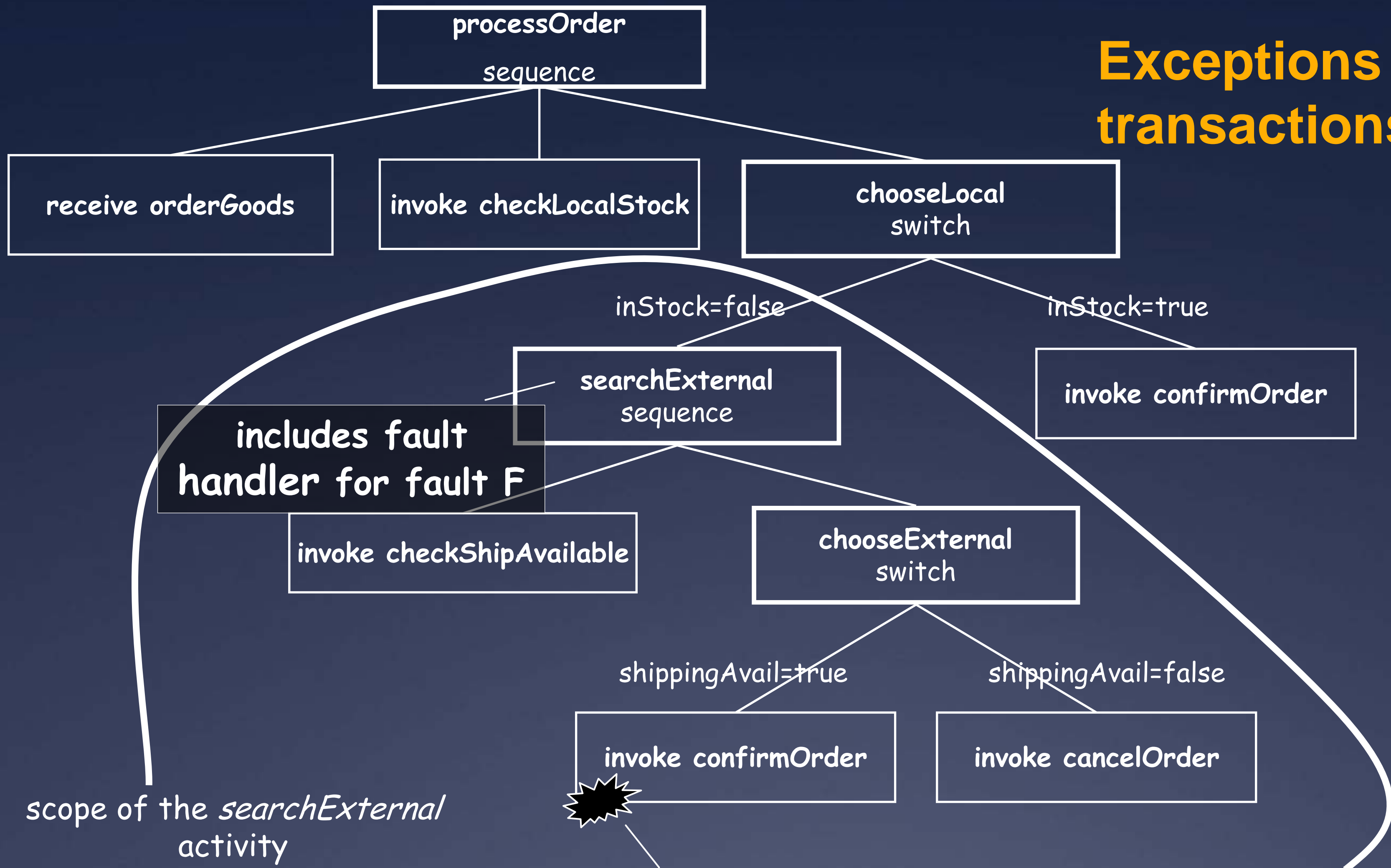
* Java-style, with a twist or two
* Throw and catch (and rethrow to enclosing scopes)
* Implicit faults (e.g., fault response to an invoke) or explicit throws
* Named and typed faults
* Fault handlers available as the scope starts
* Default fault handlers: compensate and rethrow
* *Memorize for later: if a fault happens, the scope where it happens is NOT considered as completed successfully*

```
<throw faultName=”x:foo" faultVariable="variable-name" />
…

<faultHandlers>
<catch faultName="x:foo">
<empty />
</catch>
<catch faultVariable="bar" faultMessageType="tns:barType">
<empty />
</catch>
<catch faultName="x:foo"  faultVariable="bar"
faultMessageType="tns:barType">
<empty />
</catch>
<catchAll>
<empty />
</catchAll>
</faultHandlers>
```

Fault behaviors

- Interrupt receive, invoke, reply
  - Discard answers to invokes
  - Assign MAY complete
- Interrupt structured activities
  - Termination handlers: special kind of fault handlers
  - Used to terminate enclosed scopes
  - Cannot throw faults, they just clean things up before termination

Exceptions and transactions



due to the behavior of the default handler, implicitly associated with each activity, a fault F occurring in activity *invoke confirmOrder* would propagate up until activity searchExternal, where the handler resides

copyright Springer-Verlag, 2004

Back to links: Completions and source faults

* Fact 1: The associated source activity MUST complete before the <transitionCondition> of a link is evaluated.
* Fact 2: A <scope> may suffer an internal fault and yet complete (unsuccessfully) if there is a corresponding fault handler associated with the <scope> and that fault handler completes without throwing a fault.

Faults in condition evaluation

* What happens?
  * Completion status of source activity SA is not affected
  * If the target of the link is outside the source activity's enclosing scope then the status of the link is false. If the target is within the enclosing scope the status is irrelevant since the scope has faulted.
  * Fault propagated to enclosing scope of SA
* All other remaining outgoing links from SA with targets within the source activity's enclosing scope MUST NOT have their transition conditions evaluated and remain in the unset state. However, if the target of a remaining outgoing link is outside the source activity's enclosing scope then the status of the link MUST be set to false.

And a quick detour through transactions in business processes

Transactions in BPEL

Transaction Management

* ACID properties of transactions
* Distributed transaction management
* Transactions in business processes

Atomic transactions

* A transaction is a logical unit of execution
* A transaction is a set of data operations that must be executed as a whole

* The database system must ensure that either (1) and (2) happen or that neither happens!

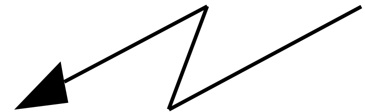
ACID transactions: keyabstractions

- Atomicity: a transaction is an atomic unit of processing and it is either performed entirely or not at all
- Consistency: a transaction's correct execution must take the database from one correct state to another
- Isolation (and serializability): the updates of a transaction must not be made visible to other transactions until it is committed
- Durability: if a transaction changes the database and is committed, the changes must never be lost because of subsequent failure

Transactionstates and operations

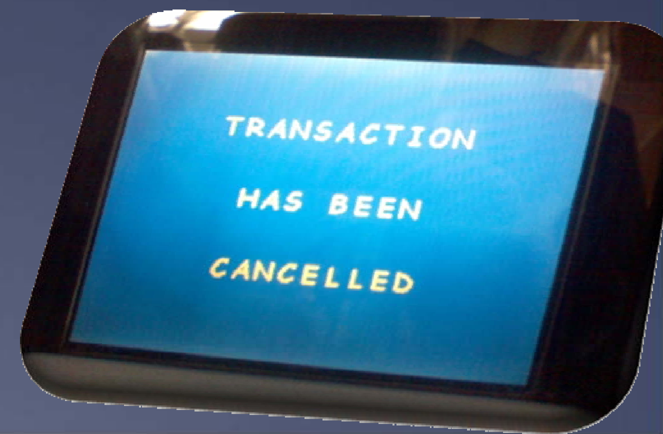
- BEGIN_TRANSACTION: marks the beginning of transaction execution
- READ or WRITE: specify read or write operations on the database items that are executed as part of a transaction.
- LOCK: locks the data to be updated in the database.
- COMMIT: signals a successful end of the transaction that any changes (updates) executed by the transaction can be safely committed to the database and will not be undone.
- ROLLBACK (or ABORT): signals that the transaction has ended unsuccessfully, so that any changes or effects that the transaction may have applied to the database must be undone.

Threeoutcomesfor a simple transaction

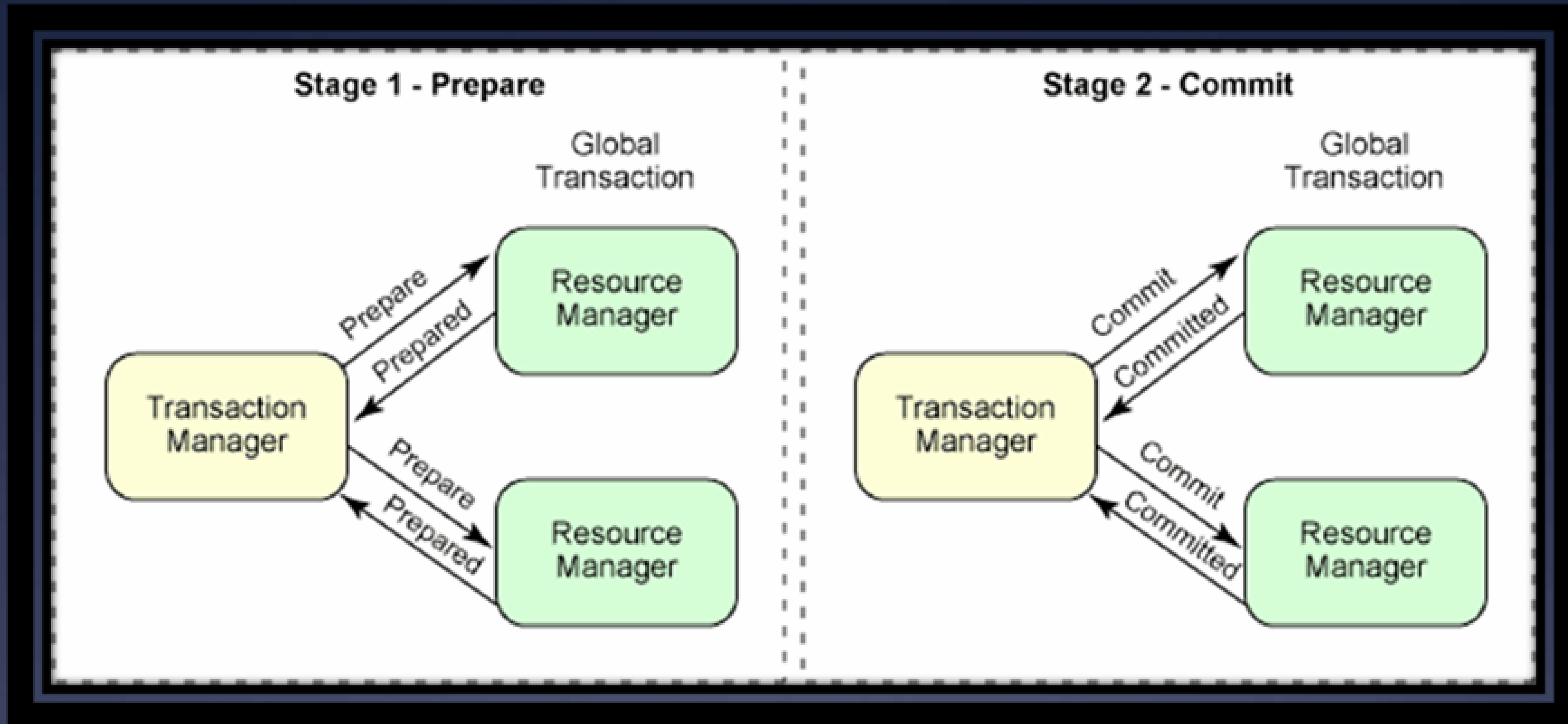
BEGIN WORK Operation 1 Operation 2 ... Operation k COMMIT WORK	BEGIN WORK Operation 1 Operation 2 ... (I got lost !) ROLLBACK WORK	BEGIN WORK Operation 1 Operation 2 ...  Rollback due to external cause like timeout, etc.
Successfulcompletion	Applicationrequests termination (Suicide)	Forcedtermination (Murder)

Two-phase commit (2PC)

- Protocol forachievingACIDityamong multiple participants.
  - But, what's the problem? Why do we needsomething different?
- Partiesfirstagree to attempt the transaction, then all must agree to either go forwardwith the entire unit of workornot.
- Central coordinatorrequired.



Example 2PC



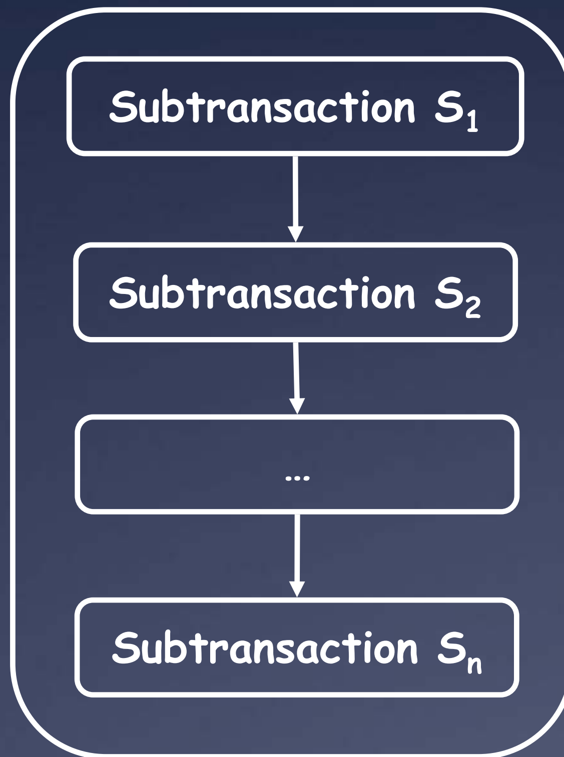
prepare phase: participants that can commit are required to allow completion if failure

commit/rollback phase: coordinator records sufficient information to complete in case of failure

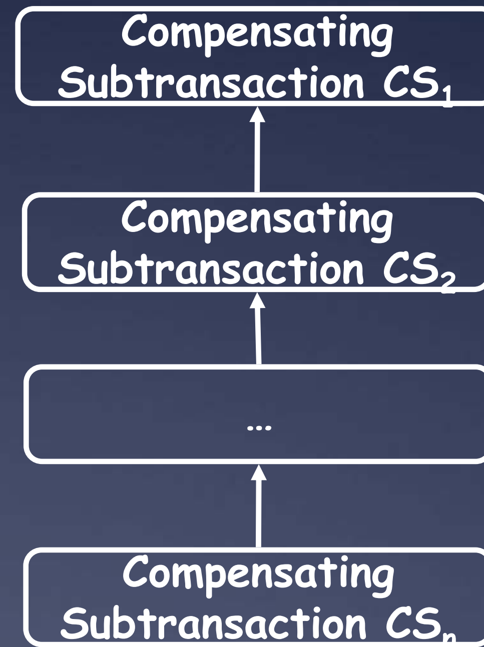
Transactions in processes and services

* Why we introduced transactions? And why 2PC?
  * Convenient abstraction
  * 2pc for distributed env as the one of web services
* Processes and services: Two other important characteristics
  * Long running (LRT)
  * Activities cannot be easily "undone" (and undo is application-specific)
* Implication:
  * 1. pure ACIDity is neither desired nor possible in the general case
  * 2. need specification of how to undo completed action

Long-lived transaction T (saga)



Forward execution



Compensation flow

Copyright Springer Verlag Berlin Heidelberg 2004

Transactions in BPEL

- Based on sagas
  - *Compensation handlers* undo activities
  - Available when the scope terminates **normally**

```
<scope>
<compensationHandler>
<invoke partnerLink="Seller"
portType="SP:Purchasing"
operation="CancelPurchase"
inputVariable="getResponse"
outputVariable="getConfirmation">
…
</invoke>
</compensationHandler>
<invoke partnerLink="Seller"
portType="SP:Purchasing"
      operation="Purchase"
inputVariable="sendPO"
outputVariable="getResponse">
…
</invoke>
</scope>
```

Inline declaration

```
<invoke partnerLink="Seller"
   portType="SP:Purchasing"
   operation="Purchase"
   inputVariable="sendPO"
   outputVariable="getResponse">
...
<compensationHandler>
<invoke partnerLink="Seller"
         portType="SP:Purchasing"
         operation="CancelPurchase"
         inputVariable="getResponse"
         outputVariable="getConfirmation">
...
</invoke>
</compensationHandler>
</invoke>
```

Invoking compensation handlers

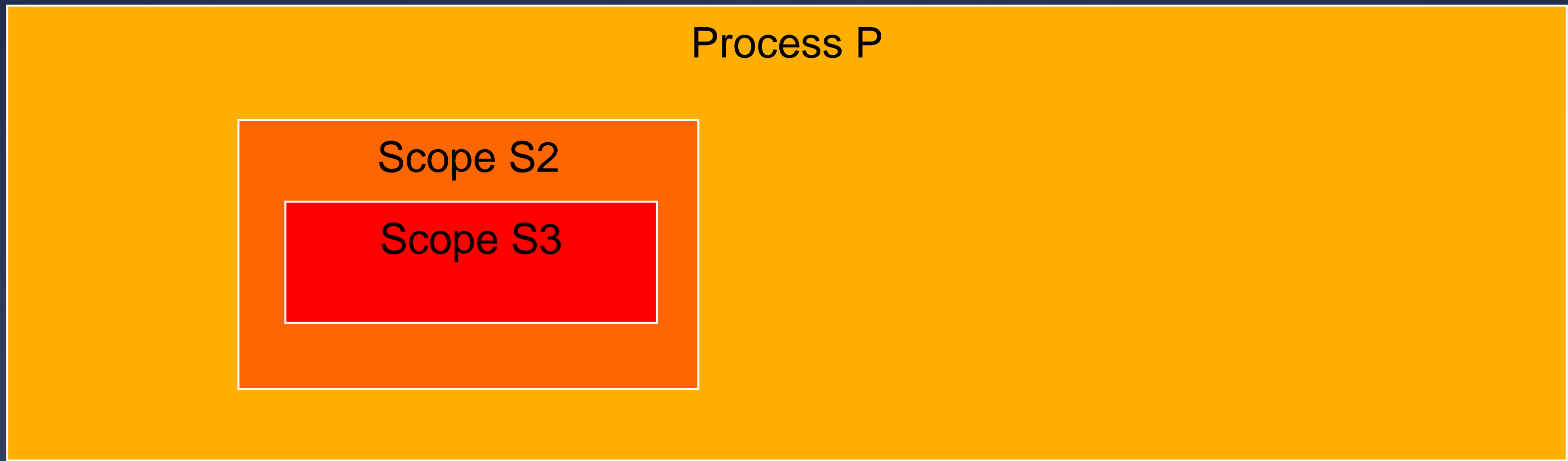
* Can compensate immediately enclosed scopes

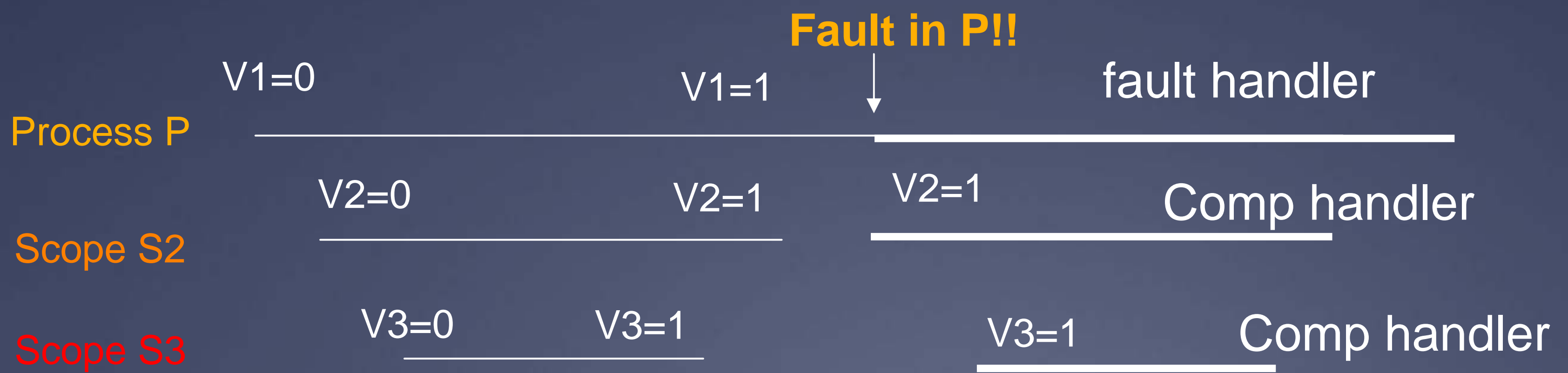
<compensateScope target="RecordPayment"/>

* Compensation handlers available for invocation only upon successful completion
* Attempts to compensate active or non started activities are like executing empty activities
  * No throwing of errors. So easier to write compensation handlers, do not need to query state

Default compensation

* Compensates all enclosed scopes in the *default order*
* Invoke handlers in reverse order of execution
  * But ONLY if order is deterministically prescribed by the process logic (eg not in flows)
  * For example: Compensation in reverse order of a while or repeat or non-parallel for
  * But: Parallel foreach compensated non deterministic order


Process P
Scope S2
Scope S3


Fault in P!!
V1=0
V1=1
fault handler
Process P
V2=0
V2=1
V2=1
Comp handler
Scope S2
V3=0
V3=1
V3=1
Comp handler
Scope S3

Isolation

* Isolated scope: exclusive access to resources
* Execution is as if it was serialized

Process Usage Patterns

* Aiming for a single approach for both
  …
  * Executable processes
    * Contain the partner's business logic behind an external protocol
  * Abstract processes
    * Define the publicly visible behavior of some or all of the services an executable process offers
    * Define a process template embodying domain-specific best practices

Why not just use java

- Naturally wsdl, xml
- Designed for parallel and p2p (pick, events, correlations…) not only for driving
- Transactions
- Complex parallel behavior

www.oasis.org

WS-BPEL 2.0

WS-BPEL 2.0 Extensions for People (BPEL4People)
http://www-128.ibm.com/developerworks/webservices/library/specification/ws-bpel4people

WS-BPEL 2.0 Extensions for Sub-Processes (BPEL-SPE)
http://www-128.ibm.com/developerworks/webservices/library/specification/ws-bpelsubproc

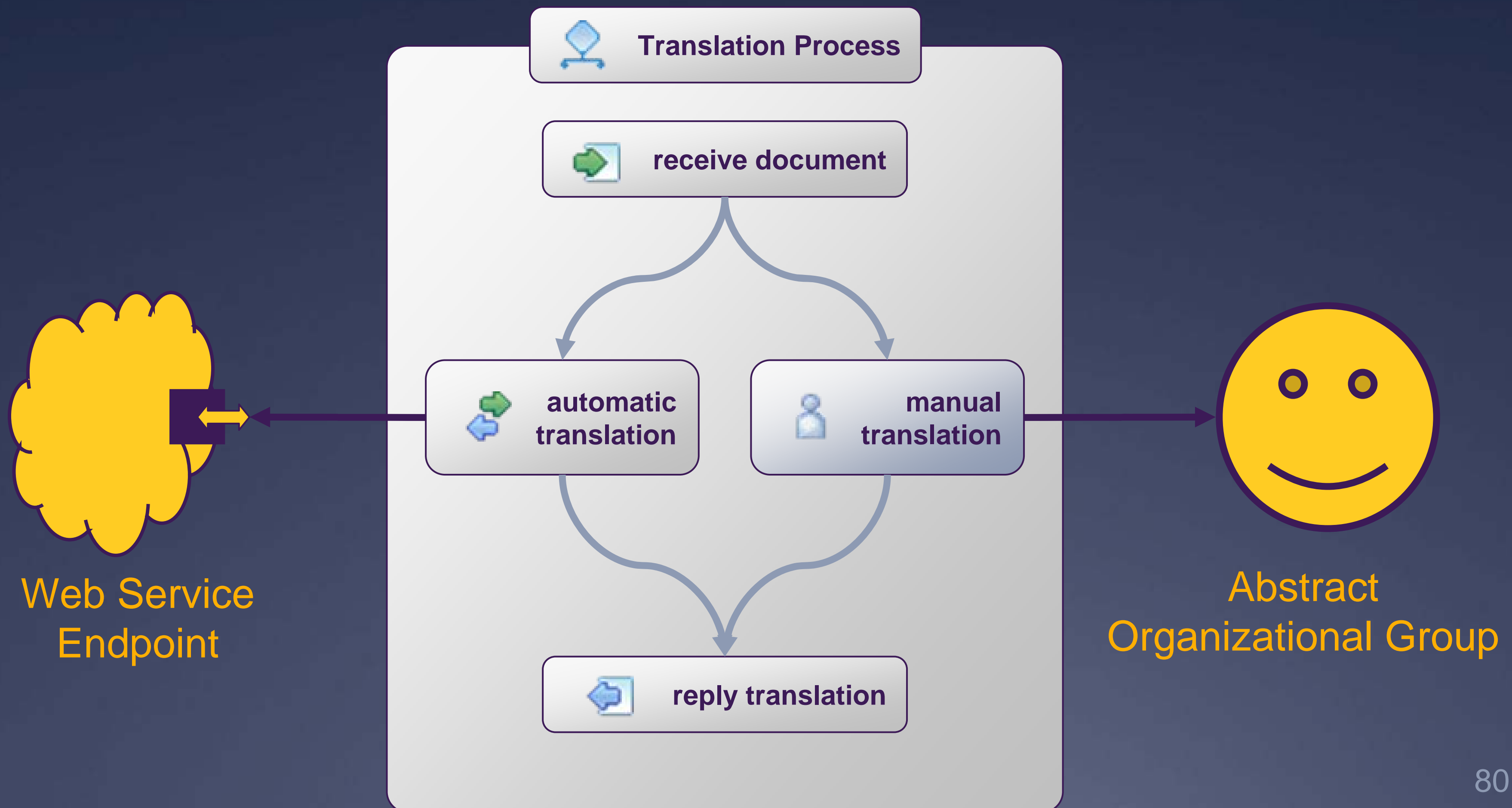
WS-BPEL 2.0 Extensions for Java (BPELJ)
http://www-128.ibm.com/developerworks/library/specification/ws-bpelj

www.osoa.org

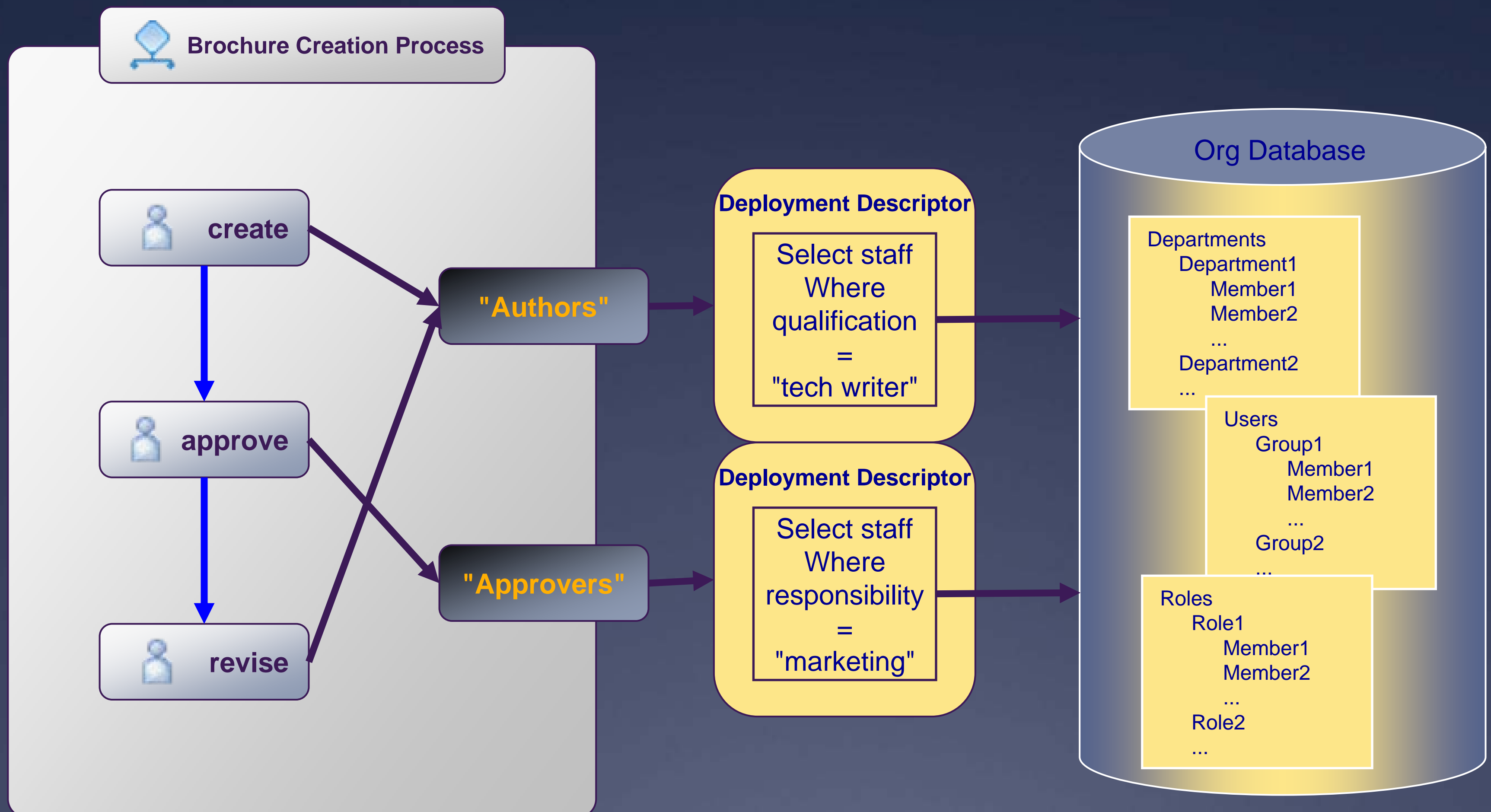
Service Component Architecture (SCA)
SCA Assembly Model
SCA WS-BPEL Client and Implementation Model

THANKS TO DIETER KOENIG FOR SOME OF THE SLIDES PRESENTED HERE

WS-BPEL Extensions for People

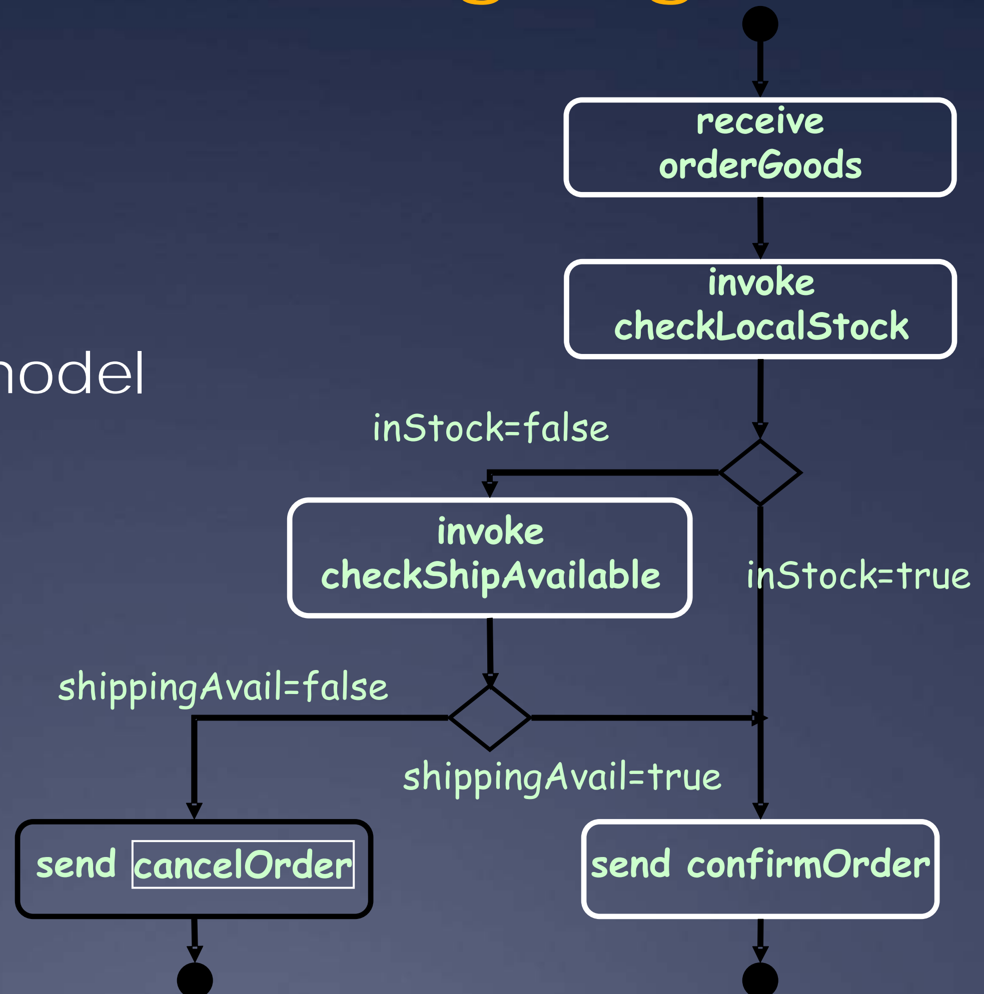


BPEL4 People – Logical People Group



Dimensions of a composition model and language

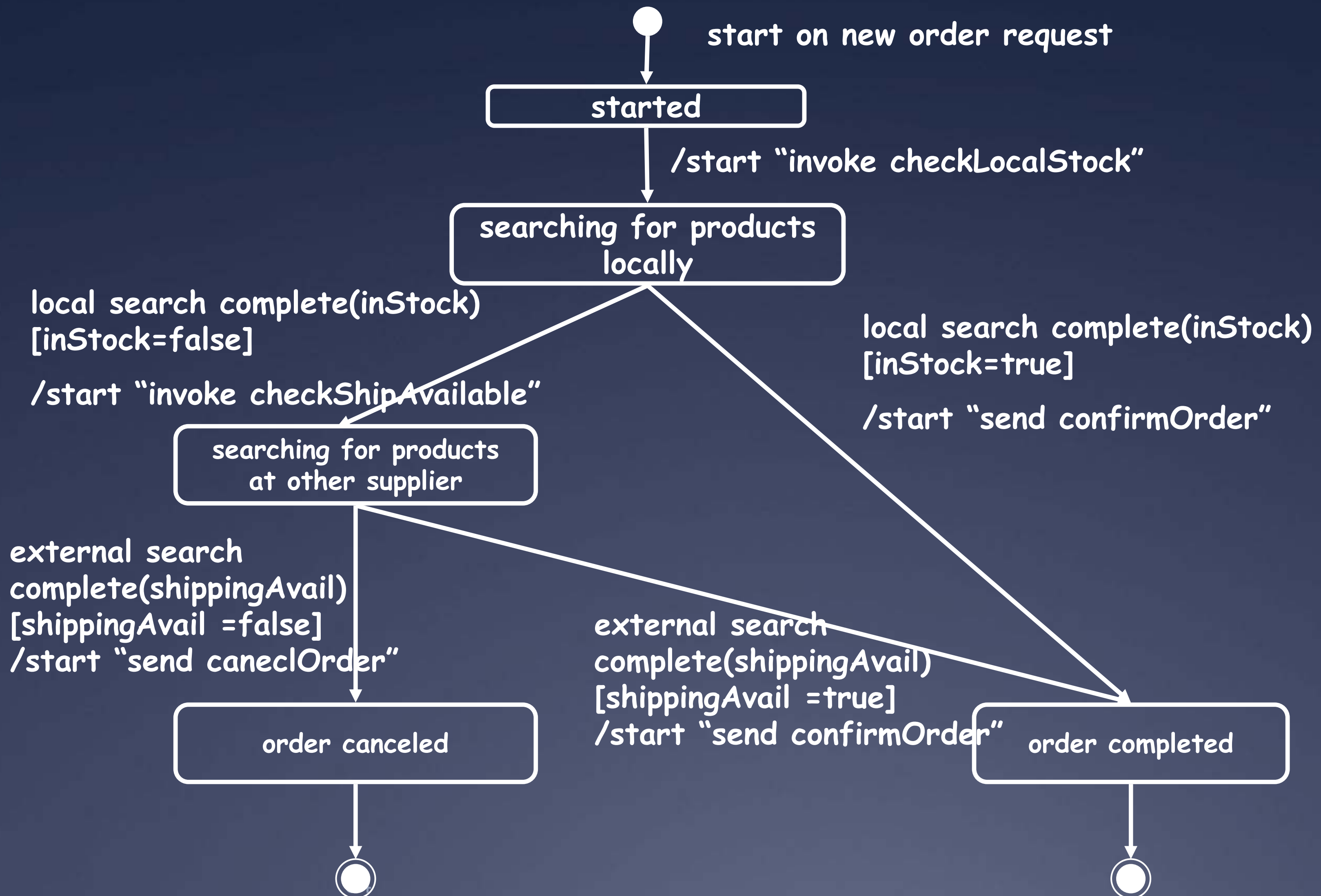
* Component model
* Orchestration model
* Data and data access model
* Service selection model
* Exceptions
* Correlation



Orchestration models - formalisms

* Activity diagrams (seen before)
* State charts
* Petri nets
* Activity decomposition
* Rule-based

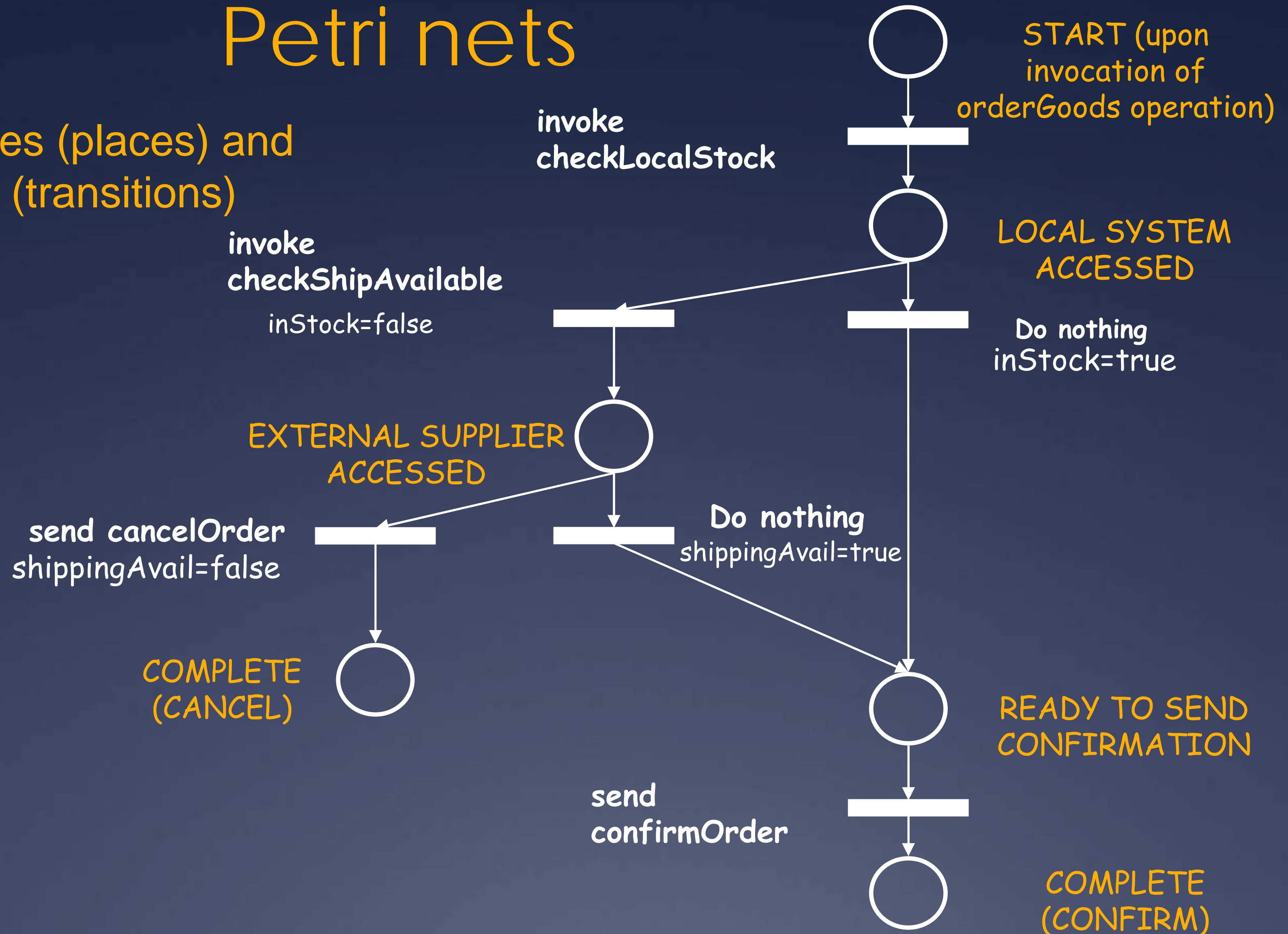
Statecharts



copyright Springer-Verlag, 2004

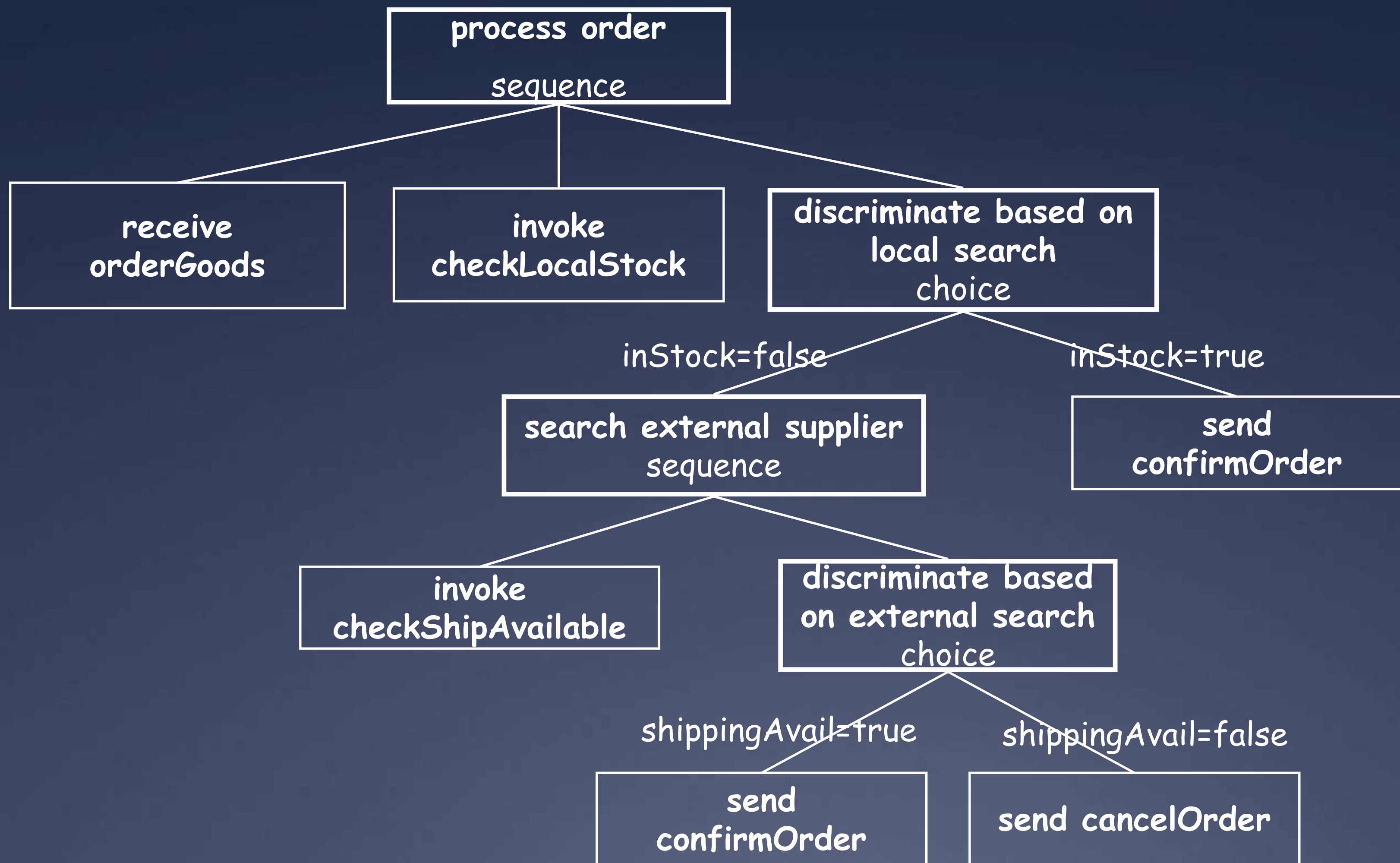
Petri nets

both states (places) and activities (transitions)



copyright Springer-Verlag, 2004

Activity decomposition



copyright Springer-Verlag, 2004

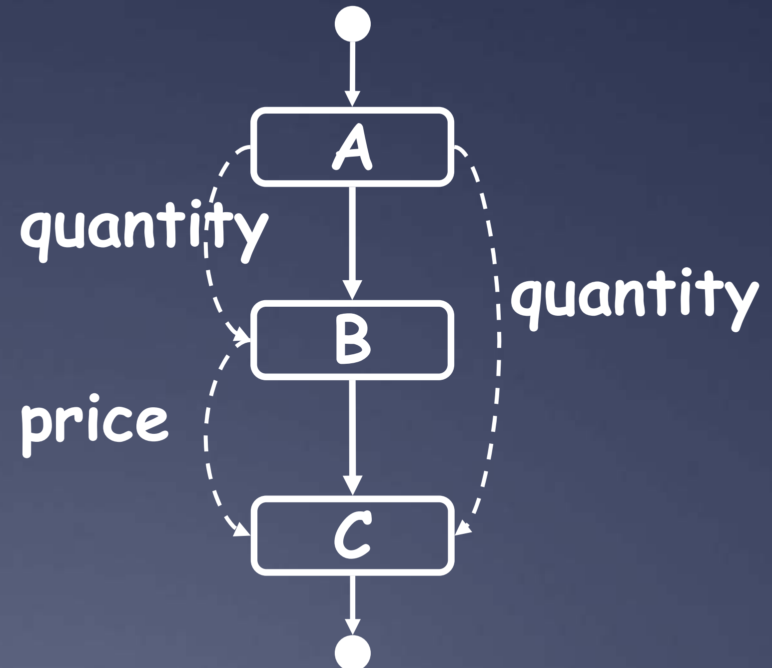
Rules

```
ON receive orderGoods
IF true
THEN invoke checkLocalStock;

ON complete(checkLocalStock)
IF (inStock==true)
THEN send confirmOrder;

ON complete(checkLocalStock)
IF (inStock==false)
THEN invoke checkShipAvailable;

ON complete(checkShipAvailable)
IF (shippingAvail ==true)
THEN send confirmOrder;

ON complete(checkShipAvailable)
IF (shippingAvail ==true)
THEN send cancelOrder;
```


copyright Springer-Verlag, 2004

Data and data access model

* Blackboard vs data flow
* "traditional" vs web services-typed
  * Xml, WSDL messages

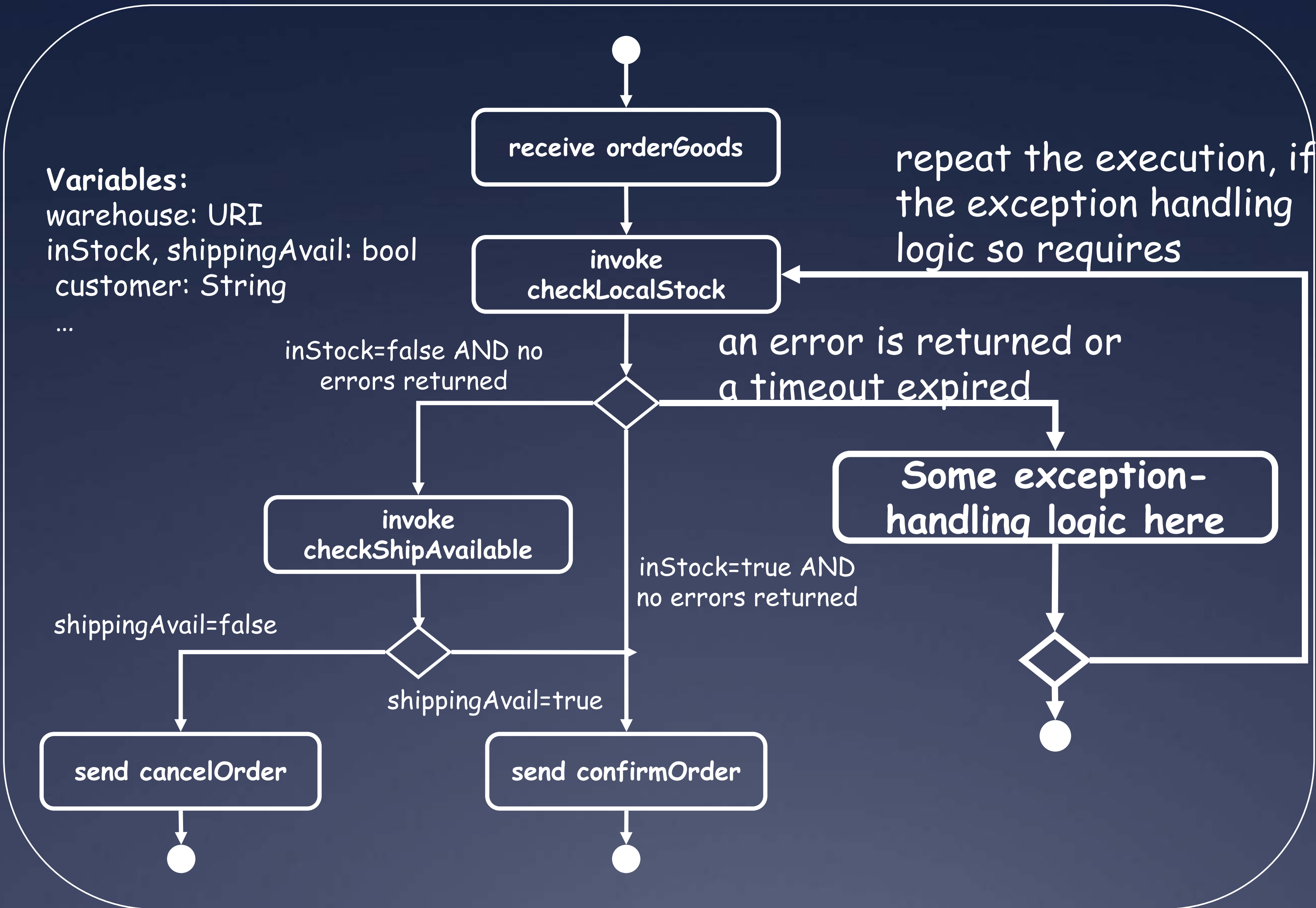


Service selection

* Static: hardcode URI
* Deployment time: map deployment parameters to URIs
* Dynamic:
  * Variable-based
  * Query/registry-based
  * Broker-based

Exceptions

* Two kinds: expected vs unexpected
  * Differ in how specific my handling can be
* ECA vs "throw and catch" vs flow
  * TC winning approach
  * Requires scopes



Copyright Springer Verlag Berlin Heidelberg 2004

Correlation

* Two main options:
  * Correlation sets
  * System-generated ID
* Correlation sets are generic, and allow to have a pub/sub combined with orchestration
* ID: Stay tuned for ws-addressing and variations

Summary: what did we learn?

* A programming language oriented to service composition. But, more importantly:
* The needs of a svc comp language, and the differences with conventional languages
* That parallelism is useful but tricky
* Compensations
* Correlations
* Late binding
* That the same behavior can be modeled in different ways. Each has its pros/cons, target users, target applications
* Simple, clear constructs are good for vendors of composition software and for programmers

References

* Alonso at al: Web services – Concepts, Architectures, and Applications. Springer Verlag. 2004
* Curbera et al - Web Services Platform Architecture. Prentice Hall
* Leymann and roller: Production workflows. Prentice Hall
* Papazoglou: Web services – Principles and Technologies. Prentice-Hall

WS-BPEL Resources

* OASIS Technical Committee
  http://www.oasis-open.org
* WS-BPEL 2.0 – oasis standard
  http://www.oasis-open.org/committees/document.php?document_id=14314&wg_abbrev=wsbpel
* Primer: http://www.oasis-open.org/committees/download.php/23964/wsbpel-v2.0-primer.htm
* Info aggregator sites
  * Wikipedia
    http://en.wikipedia.org/wiki/BPEL
  * BPEL Resource Guide
  http://bpelsource.com