Egon Börger (Pisa)

Service Interaction Patterns and Interaction Flows

An ASM-Based Compositional Framework

Università di Pisa, Dipartimento di Informatica, I-56127 Pisa, Italy
boerger@di.unipi.it

Joint work with Alistair Barros, SAP Research, Brisbane, Australia

Goal

Support software-engineered business process management in multi-party collaborative environments

We define

- *ground models* for fundamental service interaction patterns
- *composition schemes* to build complex service-based business process interconnections and interaction flows

providing a *rigorous basis for*

- execution-platform-independent *analysis* (e.g. benchmarking of web services functionalities)
- *implementations by refinements* of standard specifications (e.g. to BPEL programs)

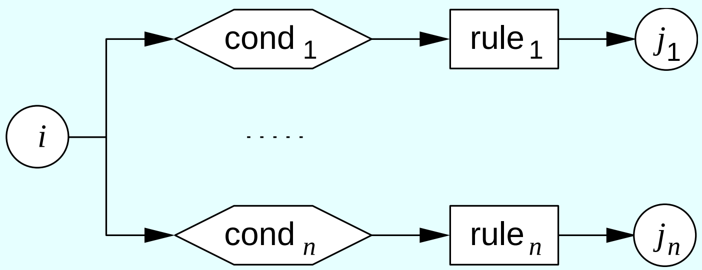
Technical Results

- *precise high-level basic models* for eight fundamental service interaction patterns
  - four basic bilateral business process interaction patterns
  - refinements to four basic multilateral interaction patterns
- *combinations and refinements* of fundamental patterns defining arbitrarily complex interaction patterns of distributed service-based business processes that
  - go beyond simple request-response sequences
  - may involve a dynamically evolving number of participants

Method:

- construction of *ASM ground models* for basic patterns
- application of *ASM refinements* for instantiation and combination of basic patterns to complex schemes

Abstract State Machine = FSM with Generalized State



if $ctl\_state = i$ **then**
  if $cond_1$ **then**
    $rule_1$
    $ctl\_state := j_1$
    $\ldots$
  if $cond_n$ **then**
    $rule_n$
    $ctl\_state := j_n$

instructions $\textsc{Fsm}(i, \textbf{if } cond_\nu \textbf{ then } rule_\nu, j_\nu)$ updating

- a single internal ctl_state assuming values $i, j_1, \ldots, j_n$ in a not furthermore structured finite set
- in/output locations in, out assuming values in finite alphabets

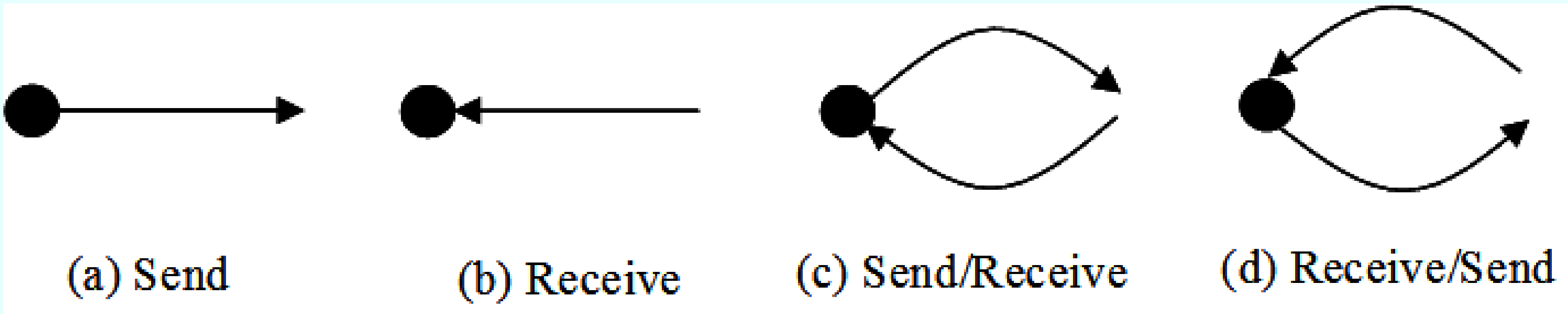
are extended by allowing

- a *set of parameterized locations holding values of whatever types*
- simultaneous updates of arbitrary many locations via *multiple assignments* $loc(x_1, \ldots, x_n) := val$

resulting in rules of form $\textbf{if } cond \textbf{ then } assignments$ with

- non-determinism replaced by synchronous parallelism

4 Basic Components of Bilateral Interaction Patterns



Each pattern describes one side of an interaction, resulting in a mono-agent ASM (or module) defined below.

Refinements of those 4 basic patterns suffice to compose any other bilateral interaction pattern, of whatever structural complexity.

Requirements for Send Pattern

Variations depending on whether

- delivery is reliable (guaranteed) or not
- action is blocking or non-blocking (in case of reliable delivery)
- sending may result in a fault message in response
- periodic resending of a message is performed

'*Counter-party* may or may not be known at design time' reflected by possibly dynamic function

$$recipient\colon Message \to Recipient$$

$$recipient\colon Message \times Param \to Recipient$$

Unspecified message delivery system reflected by abstract submachines

$$\textsc{BasicSend}(m), \textsc{BasicSend}(m, r)$$
$$\quad \textbf{where } r = recipient(m, param)$$

Requirements for Send Pattern (Cont'd)

Regular behavior by $\textsc{FirstSend}(m)$ without further resending, triggered by a monitored guard $SendMode(m)$ typically assuming:

- $SendMode(m)$ **and not** $OkSend(m)$ (read: there is no open channel connecting sender to recipient) implies $SendFaultMode(m) = true$

Faulty behavior

- originating at the sender's side, during an attempt to send m: triggers an abstract $\textsc{SendFaultHandler}$ submachine guarded by $SendFaultMode(m)$
- originating at the receiver's side as result of sending m: reflected by an abstract monitored predicate $Faulty(m)$

incorporated into an abstract machine $\textsc{HandleSendFault}(m)$.

Sender Scheme

$\textsc{Send\&Check} =$

$\quad \textsc{FirstSend}(m)$

$\quad \textsc{HandleSendFault}(m)$

where

$\quad \textsc{FirstSend}(m) = \textbf{if } \mathit{SendMode}(m) \textbf{ then}$

$\quad\quad \textbf{if } \mathit{OkSend}(m) \textbf{ then}$

$\quad\quad\quad \textsc{BasicSend}(m)$

$\quad\quad\quad \textbf{if } \mathit{AckRequested}(m) \textbf{ then } \textsc{SetWaitCondition}(m)$

$\quad\quad\quad \textbf{if } \mathit{BlockingSend}(m) \textbf{ then } \mathit{status} := \mathit{blocked}(m)$

$\quad \textsc{HandleSendFault}(m) =$

$\quad\quad \textbf{if } \mathit{SendFaultMode}(m) \textbf{ then } \textsc{SendFaultHandler}(m)$

- assuming preemptive rule firing (for notational convenience)
- $\textsc{SetWaitCondition}$ typically $\textsc{Initialize}$s a shared predicate $\mathit{WaitingFor}(m)$ an acknowledgement

Instantiating the Sender ASM

- *Send Without Guaranteed Delivery*

  $MODULE\ \ \textsc{Send}_{noAck} = \textsc{Send\&Check}$

  where forall m

  $AckRequested(m) = BlockingSend(m) = \mathit{false}$

- *Guaranteed Non-Blocking Send*

  $MODULE\ \ \textsc{Send}_{ackNonBlocking} = \textsc{Send\&Check}$ **where**

  forall m

  $AckRequested(m) = \mathit{true}$ **and** $BlockingSend(m) = \mathit{false}$

  $\textsc{SetWaitCondition}(m) =$

  $\textsc{Initialize}(WaitingFor(m))$

  $\textsc{Set}(deadline(m), sendTime(m), frequency(m), \ldots)$

- $WaitingFor(m)$ reset to false typically by $recipient(m)$
- typically $Timeout(m) = (now - sendTime(m) > deadline(m))$
- frequent scheduler requirement: $SendFaultMode(m)$ implied by a (often monitored) predicate $Timeout(m)$

Instantiation to Guaranteed Blocking Send

$$MODULE \ \textsc{Send}_{ackBlocking} =$$
$$\textsc{Send\&Check} \cup \{\textsc{UnblockSend}(m)\} \ \textbf{where}$$

$$\textbf{forall } m \ AckRequested(m) = BlockingSend(m) = true$$

$$SendMode(m) \ = \ (status = readyToSend)$$

$$\textsc{SetWaitCondition}(m) =$$
$$\quad \textsc{SetWaitCondition}_{\textsc{Send}_{ackNonBlocking}}(m)$$

$$\textsc{UnblockSend}(m) = \textbf{if } UnblockMode(m) \textbf{ then}$$
$$\quad \{\textsc{Unblock}(status), \textsc{PerformAction}(m)\}$$

$$UnblockMode(m) \ =$$
$$\quad status = blocked(m) \textbf{ and not } WaitingFor(m)$$

$$SendFaultMode(m) \ = \ Faulty(m) \textbf{ and}$$
$$\quad status = blocked(m) \textbf{ and } WaitingFor(m)$$

Adding Resending to $\textsc{Send}$

Add new machine $\textsc{ReSend}(m)$, triggered periodically, at $ResendTime(m)$

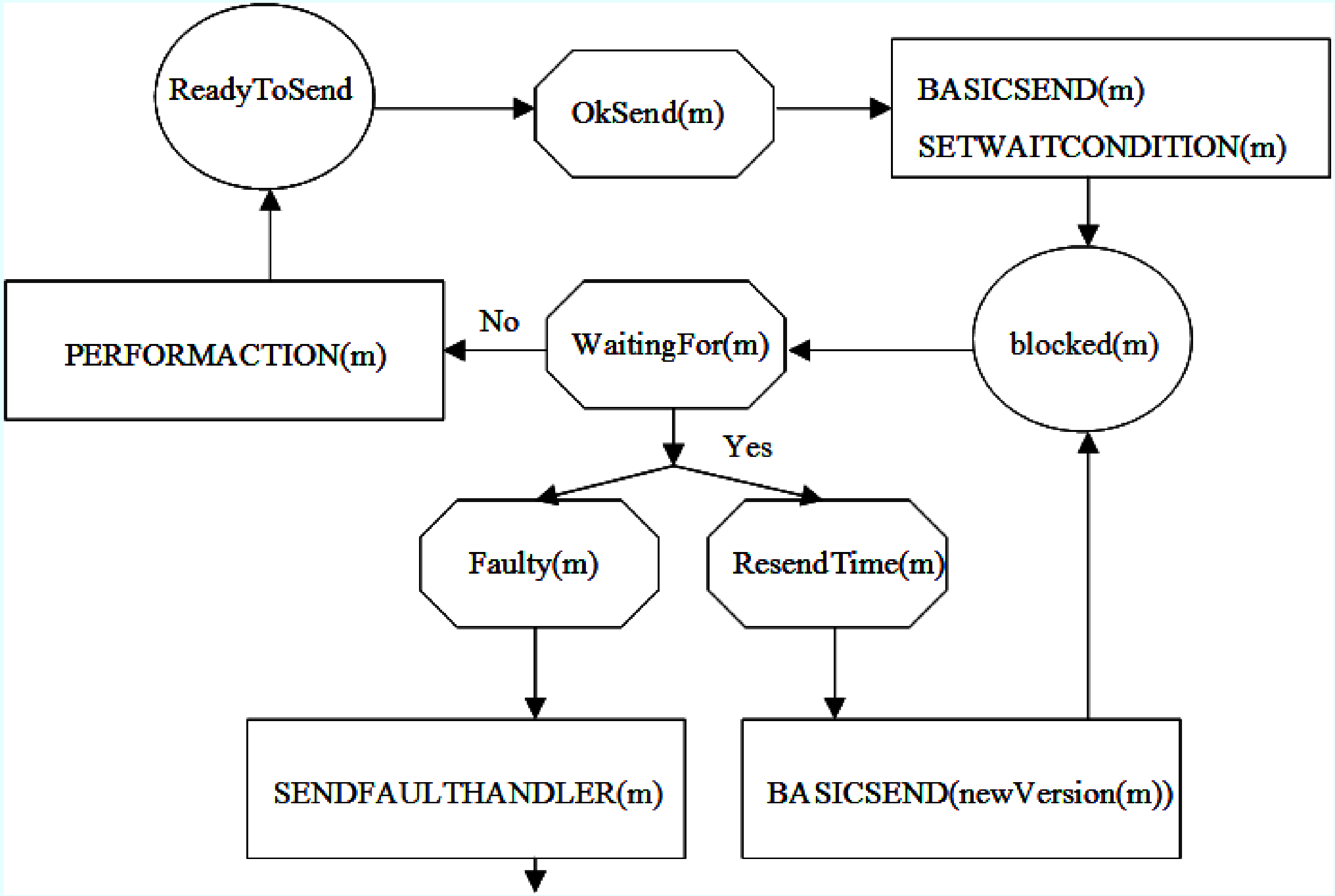
- until $WaitingFor(m) = false$ or a $Faulty(m)$ event triggers $\textsc{HandleSendFault}(m)$, typically stopping $\textsc{ReSend}$ing
- $ResendTime(m)$ typically depends on $lastSendTime(m), now$
- message copies $newVersion(m, now)$ may vary in time

$$MODULE\ \ \textsc{Send}_{tResend} = \textsc{Send}_t \cup \{\textsc{ReSend}(m)\}$$

where

$$\textsc{ReSend}(m) = \textbf{if } ResendMode(m) \textbf{ then}$$
$$\quad \textsc{BasicSend}(newVersion(m, now))$$
$$\quad lastSendTime(m) := now$$
$$ResendMode(m) \ = \ ResendTime(m) \textbf{ and } WaitingFor(m)$$

Blocking Send with Acknowledgement and Resend



- Generalizes the Alternating Bit Sender

Receive Pattern Requirements

Variations depending on whether

- receive action is *blocking* or non-blocking,
- messages that upon arrival cannot be received are *buffered* for further consumption or discarded
- an *acknowledgement* is required or not
- receive action may result in a *fault message* or not

reflected using abstract predicates and related submachines

- $Arriving(m)$ (read: m in message channel or buffer)
- $ReadyToReceive(m)$
- $ToBeDiscarded(m)$
- $ToBeBuffered(m)$
- $ToBeAcknowledged(m)$

Receive Pattern ASM

$\textsc{Receive}(m) = \textbf{if } Arriving(m) \textbf{ then}$

$\quad \textbf{if } ReadyToReceive(m) \textbf{ then}$

$\quad\quad \textsc{Consume}(m)$

$\quad\quad \textbf{if } ToBeAcknowledged(m) \textbf{ then}$

$\quad\quad\quad \textsc{BasicSend}(Ack(m), sender(m))$

$\quad \textbf{elseif } ToBeDiscarded(m) \textbf{ then}$

$\quad\quad \textsc{Discard}(m)$

$\quad \textbf{else } \textsc{Buffer}(m)$

$\textbf{where } \textsc{Buffer}(m) =$

$\quad \textbf{if } ToBeBuffered(m) \textbf{ then}$

$\quad\quad \textsc{Enqueue}(m)$

$\quad\quad enqueueTime(m) := now$

$\quad \textbf{if } DequeueTime \textbf{ then } \textsc{Dequeue}$

Instantiating Variations of $\textsc{Receive}$

$\textsc{Receive}_{blocking} = \textsc{Receive}$

where forall m

$$ToBeDiscarded(m) = false = ToBeBuffered(m)$$

$$DequeueTime = false$$

- no message is discarded or buffered
- therefore an $Arriving(m)$ upon **not** $ReadyToReceive(m)$ blocks the machine until $ReadyToReceive(m)$

$\textsc{Receive}_{discard} = \textsc{Receive}$ **where forall** m

$$ReadyToReceive(m) = false \Rightarrow ToBeDiscarded(m) = true$$

$\textsc{Receive}_{buffer} = \textsc{Receive}$ **where forall** m

$$ReadyToReceive(m) = false \Rightarrow$$

$$ToBeDiscarded = false$$

$$ToBeBuffered(m) = true$$

Pattern Send/Receive: combining Send and Receive

- receiving a response to a previously sent request
- "a common item of information in the request and the response that allows these two messages to be unequivocally related to one another": captured by dynamic sets $RequestMsg$, $ResponseMsg$ with functions $requestMsg \colon ResponseMsg \rightarrow RequestMsg$ identifying $requestMsg(m)$ to which m is the $responseMsg$

$$MODULE \ \ \textsc{SendReceive}_{s,t} = \textsc{Send}_s \cup \{\textsc{Receive}_t(m)\}$$

where

$$Arriving(m) = Arrived(m) \textbf{ and } m \in ResponseMsg$$

$$ResponseMsg = \{m \mid m = responseMsg(requestMsg(m))\}$$

- typical assumption: after having $\textsc{Initialize}$d $WaitingFor(m)$ through $\textsc{FirstSend}(m)$, $WaitingFor(m)$ is set at the $recipient(m)$ to $false$ when $responseMsg(m)$ is defined—so that $\textsc{Receive}$ and $\textsc{UnblockSend}$ can be called for $responseMsg(m)$

Pattern Receive/Send: combining Receive and Send

- symmetric to Send/Receive but letting receiving a request precede sending the answer
- refining $SendMode(m)$ by adding the condition $m \in ResponseMsg$ to guarantee that sending out an answer message is preceded by having received a corresponding request message

$$MODULE \;\; \textsc{ReceiveSend}_{t,s} = \{\textsc{Receive}_t(m)\} \cup \textsc{Send}_s$$

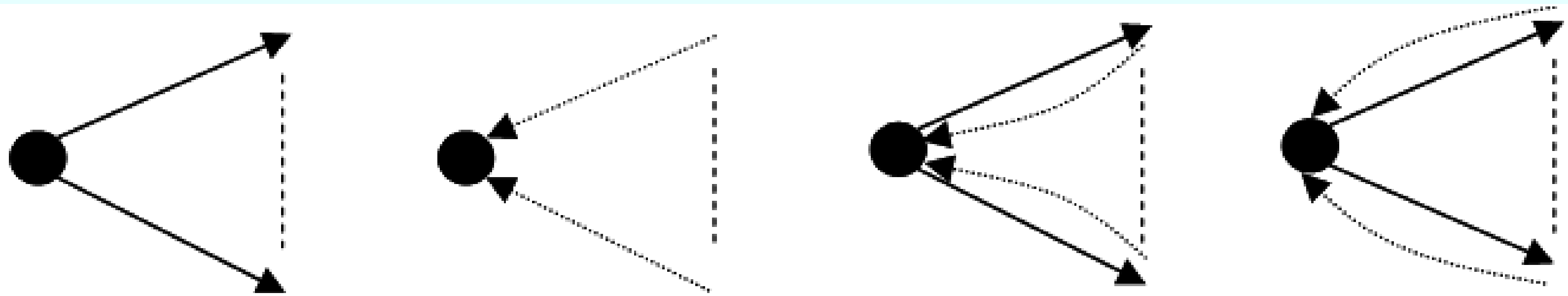
where

$$SendMode(m) = SendMode_t(m) \textbf{ and } m \in ResponseMsg$$

$$ResponseMsg = \{responseMsg(m) \mid ReceivedMsg(m)\}$$

- An example appears in the web service mediator model by Altenhofen-Boerger-Lemcke (ICFEM 2005)

4 Basic Components of Multilateral Interaction Patterns



(a) One-to-many send (b) One-from-many receive (c) One-from-many send-receive (d) One-from-many receive-send

- allowing multiple recipients or senders in each basic bilateral interaction pattern
- again each pattern describes one side of an interaction, resulting in a mono-agent ASM (or module)

Refinements of those 4 basic patterns suffice to compose any other multilateral interaction pattern, of whatever structural complexity.

One-to-many Send Pattern: refining $\textsc{BasicSend}$

- *broadcast action*: one agent sends messages to several recipients
- "*number of parties* to whom a message is sent may or may not be known at design time": captured by a dynamic set $Recipient$
- *message contents* may differ from one recipient to another, "instantiating a template with data that varies from one party to another":

  $$msgContent\colon MsgTemplate \times Recipient \rightarrow Message$$
- *variations* by refining the abstract predicates like $FaultMode(m)$ or $SendMode(m)$ accordingly

$$\begin{array}{l}\textsc{OneToManySend}_s = \textsc{Send}_s \textbf{ where} \\ \quad \textsc{BasicSend}(m) = \textbf{forall } r \in Recipient(m) \\ \qquad \textsc{AtomicSend}_{type(m,r)}(msgContent(m,r))\end{array}$$

One-from-many Receive Pattern

- *correlate messages*, received from autonomous multiple parties, into groups of given types, whose consolidation may complete successfully, e.g. into a single logical request, or trigger a failure process

$MODULE$ $\text{OneFromManyReceive}_t =$

$\{\text{Receive}_t\} \cup \text{GroupRules}$ **where**

$ReadyToReceive(m) = Accepting(type(m))$

determining which incoming mssgs should be grouped together

permits one open correlation group per mssg type

$\text{Consume}(m) = \textbf{let } t = type(m) \textbf{ in}$

if $Accepting(currGroup(t))$ *stop condition: number of mssgs to be received not necessarily known in advance*

then insert m into $currGroup(t)$

else $\text{InitializeInsert}(m, new(Group(t)))$

$ToBeDiscarded(m) = \textbf{not } Accepting(type(m))$ *no buffering*

Group Rules: Creation and Initialization

$\textsc{GroupRules} = \{\textsc{CreateGroup}(type), \textsc{Consolidate}(group),$
$\quad \textsc{CloseCurrGroup}(type), \textsc{CloseGroup}(type)\}$

$\textsc{CreateGroup}(type) =$ *Requirement: can occur at any time*
$\quad \textbf{if } GroupCreationEvent(type) \textbf{ then}$
$\qquad \textbf{let } g = new(Group(type)) \textbf{ in } \textsc{InitializeGroup}(g)$

$\textsc{InitializeGroup}(g) =$
$\quad Accepting(g) := true$
$\quad currGroup(type(g)) := g$
$\quad timer(g) := now$

$\textsc{InitializeInsert}(m, g) =$
$\quad \textsc{InitializeGroup}(g)$
$\quad$ insert m into g

Group Rules: Consolidation and Closure

$\textsc{Consolidate}(group) = \textbf{if } Completed(group) \textbf{ then}$

$\quad \textbf{if } Success(group)$ *correlation may complete successfully or not depending on the set of messages gathered*

$\qquad \textbf{then } \textsc{ProcessSuccess}(group)$

$\qquad \textbf{else } \textsc{ProcessFailure}(group)$

$\textsc{CloseCurrGroup}(type) =$

$\quad \textbf{if } Timeout(currGroup(type)) \textbf{ or } Completed(currGroup(type))$

Requirement: The arrival of mssgs needs to be timely enough for their correlation as a single logical request

$\qquad \textbf{then } Accepting(currGroup(type)) := false$

$\textsc{CloseGroup}(type) =$

$\quad \textbf{if } ClosureEvent(type) \textbf{ then } Accepting(type) := false$

One-to-many Send/Receive Pattern

- receiving *responses from multiple recipients* to a previously sent request: composing $\textsc{OneToManySend}$ and $\textsc{OneFromManyReceive}$
- responses are expected *within a given timeframe*: include $\textsc{SetWaitCondition}$ update $sendTime(m) := now$ into Send machine to determine the $Accepting$ predicate in $\textsc{OneFromManyReceive}$
- *some parties may not respond*, either not at all or not in time

$$
\begin{array}{l}
MODULE\ \ \textsc{OneToManySendReceive}_{s,t} = \\
\quad \textsc{OneToManySend}_s \cup \textsc{OneFromManyReceive}_t \\
\qquad \mathbf{where} \\
\qquad\quad Arriving(m) = Arrived(m)\ \mathbf{and}\ m \in ResponseMsg
\end{array}
$$

An instance appears in the web service mediator model by Altenhofen-Boerger-Lemcke (ICFEM 2005)

One-from-many Receive/Send Pattern

Composing $\textsc{OneFromManyReceive}/\textsc{OneToManySend}$

- $SendMode$ refined to guarantee that in any round, sent messages are responses to completed groups of received requests
- $responseMsg$ is defined not on received messages, but on their correlation groups formed by $\textsc{OneFromManyReceive}$

$$
\begin{array}{l}
MODULE\ \ \textsc{OneFromManyReceiveSend}_{t,s} = \\
\quad \textsc{OneFromManyReceive}_t \cup \textsc{OneToManySend}_s \\
\qquad \mathbf{where}\ SendMode(m)\ = \\
\qquad\quad SendMode(m)_s\ \mathbf{and}\ m = responseMsg(g) \\
\qquad\qquad \text{for some } g \in Group\ \mathbf{with}\ Completed(g)
\end{array}
$$

Generalizes abstract communication model for distributed systems proposed by Glässer et al.(IEEE Trans.SwEngg 2004): communicators route messages through a network by forwarding into the mailboxes of the $Recipient$s the mssgs found in the communicator's mailbox

Composition of basic interaction patterns

Two ways to define complex business process interaction patterns, whether

- mono-agent (bilateral and multilateral) patterns or
- asynchronous multi-agent patterns

from the eight basic interaction pattern ASMs:

- *refining the interaction rules* to tailor them to the needs of particular interaction steps
- *investigating* the order and timing of single interaction steps in (typically longer lasting) *runs of interacting agents*
  – leading to the analysis of runs of async ASMs (Co-Design FSMs with generalized state), extending classical workflow analysis by studying the effect of allowing some agents to START or SUSPEND or RESUME or STOP such collaborations (thread handling analysis)

Competing Receive Pattern: Requirements

- racing between incoming messages of various types
- exactly one among possibly multiple received messages will be chosen for a $\textsc{Continuation}$
- simultaneously one should also $\textsc{ProcessRemainingResp}$onses
- an $\textsc{EscalationProcedure}$ should be triggered in case of a $Timeout$
- no buffering is foreseen in this pattern

Competing Receive ASM: Refining $\textsc{Consume}$ in $\textsc{Receive}$

$\textsc{CompetingReceive} = \textsc{Receive}$ **where**

$\quad ReadyToReceive(m) =$ (*independent of* m)

$\qquad waitingForResponse(Type)$ **and not** $Timeout$

$\quad Arriving(m) = true$ (*independent of* m)

$\quad \textsc{Consume} =$

$\qquad$ **let** $ReceivedResponse(Type) =$

$\qquad \{r \mid Received(r)$ **and** $Response(r, t)$ **forsome** $t \in Type\}$

$\qquad\quad$ **if** $ReceivedResponse(Type) \neq \emptyset$ **then**

$\qquad\qquad$ **let** $resp = select(ReceivedResponse(Type))$

$\qquad\qquad\quad \textsc{Continuation}(resp)$

$\qquad\qquad\quad \textsc{ProcessRemainingResp}$

$\qquad\qquad\qquad (ReceivedResponse(Type) \setminus \{resp\})$

$\qquad\qquad waitingForResponse(Type) := false$

Competing Receive (Cont'd): Refining $\textsc{Discard}$ing

$$
\begin{array}{l}
\mathit{ToBeDiscarded}(m) = \\
\quad \mathit{Timeout}\ \mathbf{or\ not}\ \mathit{waitingForResponse}(\mathit{Type}) \\
\textsc{Discard} = \\
\quad \mathbf{if\ not}\ \mathit{waitingForResponse}(\mathit{Type})\ \mathbf{then} \\
\qquad \textsc{ProcessLateResponses} \\
\quad \mathbf{if}\ \mathit{Timeout}\ \mathbf{then}\ \textsc{EscalationProcedure}
\end{array}
$$

Multi-response Pattern

- A multi-transmission instance of SendReceive where the requester may receive multiple responses from the recipient "until no further responses are required"
- no further responses r for a request m will be accepted (and presumably discarded) for any of the following reasons:
  - a response informing that no $FurtherResponseExpected(m)$ (predicate to be set during the initialization rule in $\text{SetWaitCondition}(m)$)
  - expiry of the request $deadline(m)$: time elapsed since the $sendTime(m)$, set in $\text{SetWaitCondition}(m)$ when the request m was sent
  - expiry of the $lastResponseDeadline(m)$: time that elapsed since the last response to request message m has been received. To define this it suffices to refine $\text{Consume}(m)$ by setting $lastResponseTime(requestMsg(m)) := now$

Multi-response ASM: Refining $\textsc{SendReceive}$

$MODULE$ $\textsc{MultiResponse}_{s,t} = \textsc{SendReceive}_{s,t}$

where

$\textsc{SetWaitCondition}(m) =$

$\textsc{SetWaitCondition}_{\textsc{SendReceive}_{s,t}}(m)$

addRule $FurtherResponseExpected(m) := true$

$ReadyToReceive(m) =$

$FurtherResponseExpected(requestMsg(m))$ **and**

not $Timeout(requestMsg(m))$

$\textsc{Consume}(m) = \textsc{Consume}_{\textsc{SendReceive}_{s,t}}(m)$

addRule $lastResponseTime(requestMsg(m)) := now$

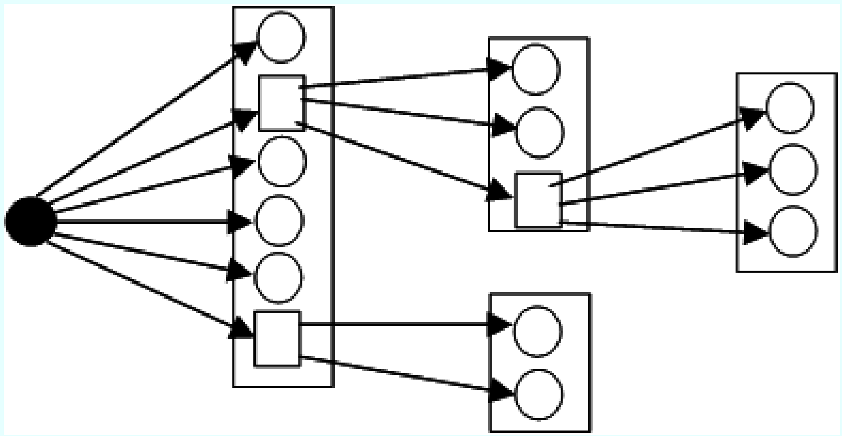
$ToBeDiscarded(m) =$ **not** $ReadyToReceive(m)$

$Timeout(m) = Expired(deadline(m))$ **or**

$Expired(lastResponseDeadline(m))$

Transactional Multicast Notify (Generalized Master-Slave)

- notification $Recipient(m)$s are arranged in groups, allowing in groups also further groups as members (arbitrary nesting)
- within each group g a certain number of members, typically between a minimum $acceptMin(m, g)$ and a maximum $acceptMax(m, g)$ (initialized in $\textsc{SetWaitcondition}$), are expected (*by the master*) to "accept" the request m within a certain timeframe $Timeout(m)$



NB. Nested group structure represented as a $recipientTree(m)$.
Leaves are pictorially represented by circles, groups by rectangles

Transactional Multicast Notify: Acceptance notion

- in the $recipientTree(m)$
  - nodes (except the root) stand for groups or recipients
  - $children(n)$ represents a group for each non-leaf n. Since each inner node has a unique such group, every corresponding $currGroup(t)$ is kept open by defining it as $Accepting$.
  - $Leaves(recipientTree(m))$ defines the set $Recipient(m)$
- $Accept$ance computed by master as part of $\textsc{PerformAction}(m)$, specified as derived predicate by a recursion on $recipientTree(m)$, abstracting from the underlying tree walk algorithm:

  $Accept(n) \Leftrightarrow \mid \{c \in children(n) \mid Accept(c)\} \mid \geq acceptMin(m, children(n))$

  $Accept(leaf) \Leftrightarrow$ from $leaf$ some $r \in ResponseMsg(m)$ was received such that $AcceptMsg(r)$
- By defining $type(r) = r$ for any response mssg r, $currGroup(r)$ collects all the $AcceptMsg$s received from brothers of $sender(r)$

Transactional Multicast Notification: priority driven selection

- at $Timeout(m)$ more than $acceptMax(m, g)$ accepting messages may have arrived
- a "priority" function $chooseAccChildren$ selects an appropriate set of accepting children among the elements of $children(n)$

$$chooseAccChildren(n) =$$
$$\begin{cases} \emptyset \ \ \textbf{if} \mid AcceptChildren(n) \mid < acceptMin(m, children(n)) \\ \subseteq_{min,max} AcceptChildren(n) \ \ \textbf{else} \end{cases}$$

where

$$AcceptChildren(n) = \{c \in children(n) \mid Accept(n)\}$$
$$min = acceptMin(m, children(n))$$
$$max = acceptMax(m, children(n))$$
$$A \subseteq_{l,h} B \Leftrightarrow A \subseteq B \ \textbf{and} \ l \leq \mid A \mid \leq h$$

Transactional Multicast Notification: $chosenAccParty(root)$

- the elements of all the selected sets constitute the $chosenAccParty(root)$ of recipients, defined as *derived set* by recursion on $recipientTree(m)$ as follows:

$$chosenAccParty(leaf) = \begin{cases} \{n\} & \textbf{if } Accept(n) \\ \emptyset & \textbf{else} \end{cases}$$

$$chosenAccParty(n) = \bigcup_{c \in chooseAccChildren(n)} chosenAccParty(c)$$

$\textsc{TransactionalMulticastNotify}$ Master ASM

$MODULE$ $\textsc{TransactionalMulticastNotify}_t =$
$\textsc{OneToManySendReceive}_{ackBlocking,t}$ **where**

$WaitingFor(m) \; = \textbf{not} \; Timeout(m)$

status=blocked(m) to receive AcceptMsges from Recipients

$\textsc{SetWaitCondition}(m) = \textsc{SetWaitcondition}_{OTMSR}(m)$

addRule

$\textsc{InitializeMinMax}(m)$

$\textsc{InitializeCurrGroup}(m)$ **where**

$\textsc{InitializeMinMax}(m) =$

$\textbf{forall } g = children(n) \in recipientTree(m)$

$\textsc{Initialize}(acceptMin(m,g), acceptMax(m,g))$

$\textsc{InitializeCurrGroup}(m) = \textbf{forall } r \in Recipient(m)$

$currGroup(r) := \emptyset$

$\textsc{TransactionalMulticastNotify}$ Master ASM (Cont'd)

$$type(response) = response$$

$$Accepting(response) = response \in AcceptMsg \textbf{ and}$$
$$\quad \textbf{not } Timeout(requestMsg(response))$$

$$currGroup(response) = currGroup(sender(response))$$

$currGroup(recipient) =$ *derived set, depending on Accept(leaf)*

$$\quad brothers\&sisters(recipient) \cap \{leaf \mid Accept(leaf)\}$$

$Accepting(currGroup(r)) =$ true

$$\textsc{PerformAction}(m) =$$
$$\quad \textbf{if } Accept(root(recipientTree(m))) \textbf{ then let}$$
$$\qquad accParty = chosenAccParty(root(recipientTree(m)))$$
$$\qquad others = Leaves(recipientTree(m)) \setminus accParty \textbf{ in}$$
$$\qquad\quad \textsc{Process}(fullRequest(m), accParty, others)$$
$$\quad \textbf{else } \textsc{RejectProcess}(m)$$

Multi-round One-to-many Send/Receive: Requirements (1)

- multiple one-to-many sends, each followed by one-from-many receives
- no a priori bound on number of receiving/responding parties: dynamic set $Recipient$
- no a priori bound on number of previously sent requests
  - dynamic set $ReqHistory$ where $currReq$ has to be stored when a new $currRequest$ is sent out
  - $WaitingFor$, $sendTime$ and $blocked$ may depend on both the message template m and the recipient r
- each response message is assumed to be a response to (exactly) one of the sent requests: define $type(m)$ for $m \in ResponseMsg$ as the $requestMsg(m)$ that triggered the response m

Multi-round One-to-many Send/Receive: Requirements (2)

- every request r is allowed to trigger more than one response m from each recipient (apparently without limit)
  - generalize $responseMsg$ to a relation $responseMsg(m, r)$
  - therefore $currGroup(request)$ represents the current collection of responses received to $request$
- "the latest response ... overrides the latest status of the data ... provided, although previous states are also maintained"
  - abstract derived set $ResponseSoFar$
  - additional machine $\textsc{MaintainDataStatus}$ keeping track of the $dataStatus$ of previous states for any request (submachine of $\textsc{Consume}$)
  - $ResponseSoFar =$
$$\bigcup\{Group(m) \mid m \in ReqHistory\} \cup \{currGroup(currReq)\}$$

$\textsc{MultiRoundOneToManySendReceive}$ **ASM**

$$
\begin{array}{l}
\mathit{MODULE}\ \textsc{MultiRoundOneToManySendReceive} = \\
\quad \textsc{OneToManySendReceive}\ \mathbf{where} \\
\mathit{SendMode}(m) = \mathit{SendMode}(m)_{\mathit{OTMSR}}\ \mathbf{and} \\
\quad \mathbf{forall}\ r \in \mathit{Recipient}(m)\ \mathit{ReadyToSendTo}(m, r) \\
\textsc{SetWaitCondition}(m) = \\
\quad \mathbf{forall}\ r \in \mathit{Recipient}(m) \begin{cases} \textsc{Initialize}(\mathit{WaitingFor}(m, r)) \\ \mathit{sendTime}(m, r) := \mathit{now} \\ \mathit{status} := \mathit{blocked}(m, r) \end{cases} \\
\quad \mathsf{insert}\ \mathit{currRequest}\ \mathsf{into}\ \mathit{ReqHistory} \\
\quad \mathit{currRequest} := m \\
\mathit{type}(m) = \mathit{requestMsg}(m) \\
\textsc{Consume}(m) = \textsc{Consume}(m)_{\mathit{OTMSR}} \\
\quad \mathbf{addRule}\ \textsc{MaintainDataStatus}(\mathit{Group}(\mathit{requestMsg}(m)))
\end{array}
$$

Request With Referral: a 2-agent Pattern

- a *sender* of requests, apparently without any reliability assumption: $\textsc{Send}_{noAck}$ module
- a *receiver* (called referral agent) from where "any follow-up response should be sent to a number of other parties ..."
  - refine $\textsc{Consume}$ submachine of $\textsc{Receive}$ to contain $\textsc{OneToManySend}$ for the set $Recipient(m)$ encoded as set of $followUpResponseAddr$essees extracted from m
  - since the follow-up response parties (read: $Recipient(m)$) may be chosen depending on the evaluation of certain conditions, $followUpResponseAddr$ can be thought of as a set of pairs of form $(cond, adr)$ where $cond$ enters the definition of $SendMode(m)$
- faults "could alternatively be sent to another nominated party or in fact to the sender"
  - $followUpResponseAddr$ may be split into disjoint subsets $failureAddr$ and $normalAddr$

2-Agent ASM $\textsc{RequestReferral}$

2-Agent ASM $\textsc{RequestReferral} =$

Sender agent with module $\textsc{Send}_{noAck}$

Referral agent with module $\textsc{Receive}$

where

$\textsc{Consume}(m) = \textsc{OneToManySend}(Recipient(m))$

$Recipient(m) = followUpResponseAddr(m)$

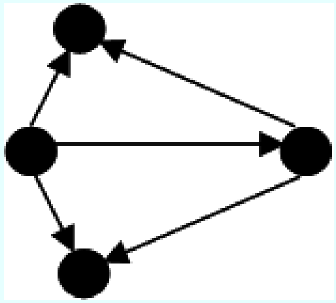
Enriching RequestReferral by Advanced Notification

- an advanced notification should be sent by the original sender to the other parties informing them that the request will be serviced by the original receiver
- sender may first send his request m to the receiver and only later inform the receiver (and the to-be-notified other parties) about $Recipient(m)$

Refine in RequestReferral

- $\textsc{Send}_{noAck}$ by a machine with blocking acknowledgment, where $WaitingFor(m)$ means that $Recipient(m)$ is not yet known and that $Timeout(m)$ has not yet happened
- $\textsc{PerformAction}(m)$ as a $\textsc{OneToManySend}$ of the notification, guarded by $known(Recipient(m))$

2-Agent ASM $\textsc{NotifiedRequestReferral}$



Sender module $\textsc{Send}_{ackBlocking} \cup \{\textsc{OneToManySend}\}$ **where**

$$
\begin{aligned}
&\quad WaitingFor(m) = \\
&\qquad \textbf{not } known(Recipient(m)) \textbf{ and not } Timeout(m) \\
&\quad \textsc{PerformAction}(m) = \\
&\qquad \textbf{if not } known(Recipient(m)) \textbf{ then } \textsc{SendFailure}(m) \\
&\qquad\quad \textbf{else } \textsc{OneToManySend}(advancedNotif(m))
\end{aligned}
$$

Referral module $\textsc{Receive}$ as in $\textsc{RequestReferral}$

Relayed Request: Another Refinement of RequestReferral

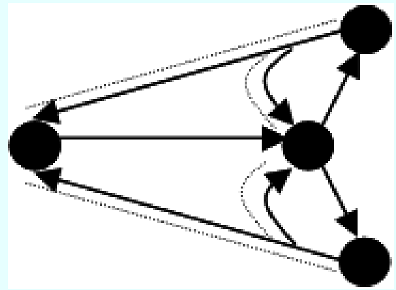
Additional requirements:

- the other parties continue interacting with the original sender
- original receiver "observes a 'view' of the interactions including faults"
- interacting parties are aware of this 'view'

Refinements to capture the new requirements:

- equip the sender also with a machine to Receive messages from third parties
- introducing a set *Server* of third party agents, each equipped with
  - Receive
  - Send&Audit refining Send by the required observer mechanism

Multiple Agent ASM $\textsc{RelayedRequest}$



n+2-Agent ASM $\textsc{RelayedRequest} =$

2-Agent ASM $\textsc{RequestReferral}$

$\quad$ **where** $module(Sender) =$

$\quad\quad module_{\textsc{RequestReferral}}(Sender) \cup \{\textsc{Receive}\}$

n Server agents with module $\{\textsc{Receive}, \textsc{Send\&Audit}\}$

$\quad$ **where**

$\quad \textsc{Send\&Audit} = \textsc{Send}_s$ **with**

$\quad\quad \textsc{BasicSend} = \textsc{BasicSend}_s \cup$

$\quad\quad\quad \{\textbf{if } AuditCondition(m) \textbf{ then } \textsc{BasicSend}_s(filtered(m))\}$

Dynamic Routing: Requirements

- a first party "sends out requests to other parties"
  - an instance of $\textsc{OneToManySend}$ to $Recipient(sender)$
- "these parties receive the request in a certain order encoded in the request. When a party finishes processing its part of the overall request, it sends it to a number of other parties depending on the 'routing slip' attached or contained in the request. This routing slip can incorporate dynamic conditions based on data contained in the original request or obtained in one of the 'intermediate steps'."
  - third party agents $\textsc{Receive}$ request mssgs m with $routingSlip(m)$, $\textsc{Consume}$ requests by $\textsc{Process}$ing them, forward a $furtherRequest(m, currState(router))$ possibly depending on data in $currState(router)$
- "The set of parties through which the request should circulate might not be known in advance. Moreover, these parties may not know each other at design/build time": dynamic set $RoutingAgent$s

Multiple Agent ASM $\textsc{DynamicRouting}$

Multi-Agent ASM $\textsc{DynamicRouting} =$

Agent *sender* with module $\textsc{OneToManySend}(Recipient(sender))$

Agents $router \in RouterAgent$ each with module $\textsc{Receive}$

where

$$\textsc{Consume}(m) =$$

$$\textsc{Process}(m) \ \mathbf{seq}$$
$$\textsc{OneToManySend}(furtherRequest(m, currState(router)))$$
$$(Recipient(router, routingSlip(m, currState(router))))$$

- *Recipient* set depends on the *router* agent and on the *routingSlip* information
- use of the ASM **seq** operator reflects an intrinsically sequential behavior

What we would like to see to be done

- a *provably correct implementation* of the pattern ASMs, e.g. by BPEL programs, using the ASM model defined by Farahbod-Glaesser-Vajihollahi for the semantics of BPEL
- using the pattern ASMs for *benchmarking* existing implementations
- *defining rigorous ASM ground models for other interaction patterns* by combining refinements of basic bilateral and multilateral service interaction pattern ASMs
- *mathematical study of conversation patterns* (business process interaction flows), viewed as runs of asynchronous multi-agent interaction pattern ASMs

References

- A. Barros and M. Dumas and A.H.M. ter Hofstede: Service Interaction Patterns: Towards a reference framework for service-based business process interconnection. FIT-TR-2005-02 Queensland University of Technology, Brisbane (Australia)
- A. Barros and E. Börger: A Compositional Framework for Service Interaction Patterns and Communication Flows. Proc. ICFEM 2005, Springer LNCS 3785 (2005) 5-35
- *AsmBook*
  E. Börger and R. F. Stärk: Abstract State Machines. A Method for High-Level System Design and Analysis
  Springer-Verlag. 2003
- *ASM Tutorial*
  E. Börger: The ASM Method for System Design and Analysis. A Tutorial Introduction
  Springer LNAI 3717 (2005), 264-283

