Egon Börger (Pisa) & Alexander Raschke (Ulm)

Modeling Contex-Aware Behavior by Ambient ASMs

Communication Patterns Case Study

Università di Pisa, Dipartimento di Informatica, boerger@di.unipi.it
Universität Ulm, Abteilung Informatik, alexander.raschke@uni-ulm.de

See Ch. 4.4 of Modeling Companion Book
http://modelingbook.informatik.uni-ulm.de

Theme: Modeling context-aware system behavior

General question: How to gently model contexts and programs whose run behavior depends on the context in which the program is executed?

- NB. Often contexts are called environments, not to be confused with env understood as interpretation of free variables

General idea: *use parameterization* to model context dependency.

Here we illustrate modeling context-aware behavior by ambient ASMs via

- *communication patterns* for bilateral and multilateral interaction
- which can be composed and instantiated to a variety of *process interaction patterns* which
  - go beyond simple request-response sequences
  - may involve a dynamically evolving number of participants (see Barros/Börger 2005) [1]

We recapitulate motivation and definition of ambient ASMs

---

[1] all figures in this lecture are © 2018 Springer-Verlag and reused with permission

The role of parameterization for ASMs

Parameterizing ASM functions and rules permits to directly model:

- partitioning/isolating of states and distributed computations
  - of agents which concurrently execute in heterogeneous contexts
- various forms of information hiding (encapsulation of memory)
  - statically: scope, module, package, library, etc.
  - dynamically: executing agents, threads, process instances, etc.
- patterns of programming, of communication and of control flow
- mobility (environments where agents can move)
- modularity (of specs and property verifications)

Exploit *simplicity of semantical foundation* of parameterization:

$$f(x) = f_{params}(x)$$

in particular when used with implicit (hidden) parameters, supporting conventional implicit oo parameterization $this.f(x) = f(x)$

Ambient ASMs to explicitly support env-sensitive actions

Idea: enrich ASMs by an abstract ambient parameter

- with respect to which the terms involved in a step are evaluated
- for ambients (environments) at whatever level of abstraction
- which can be created, modified, deleted, also at run time

Syntactical construct: **amb** exp **in** P

where to achieve generality in the widest terms

- exp is any expression (term)
- P is any (already defined) ASM program

Intended behavior (see definition below):

- PUSH the $eval(exp, S, env, amb)$ of the newly declared ambient expression to the current ambient
- execute P in the new ambient

Function classification is extended by ambient (in)dependent fcts.

Ambient dependent functions and their evaluation

$AmbDependent(f)$ (wrt Amb) **iff**

forsome $a, a' \in Amb$ **with** $a \neq a'$ **forsome** x $f(a, x) \neq f(a', x)$

Otherwise f is called $AmbIndependent$. We also say *environment (in)dependent*, hoping that Ambients will not be confused with Environments (interpretation of free variables in a state).

- Case $AmbDependent(f)$:

  $eval(f(t_1, \ldots, t_n), S, env, amb) =$
  $f_S(amb, eval(t_1, S, env, amb), \ldots, eval(t_n, S, env, amb))$

  Here f_S is turned into a family of possibly different functions $f_{S,amb}$.
- Case $AmbIndependent(f)$ (unchanged interpretation of f):

  $eval(f(t_1, \ldots, t_n), S, env, amb) =$
  $f_S(eval(t_1, S, env, amb), \ldots, eval(t_n, S, env, amb))$

Semantics of ambient ASMs

To avoid a signature blow up by dynamic ambient nesting, we adopt Simone Zenzaro's idea (PhD Thesis, Pisa 2016) to

treat amb as a stack

where new ambient expressions are pushed (passed by value)

- so that for each $f^{(n)}$ one extension $f^{(n+1)}$ suffices which offers an additional argument position for the interpretation of $f^{(n)}$ in a given ambient.

The body of **amb** exp **in** P is then executed with the new stack value.

$Yields(\textbf{amb}\ exp\ \textbf{in}\ P, S, env, amb, U)$ **if**

$$Yields(P, S, env, \textsc{push}(eval(exp, S, env, amb), amb), U)$$

- NB. Often the execution of P (read: the interpretation $f_{S,amb}$) depends only on the top of the stack, i.e. on $eval(exp, S, env, amb)$ (*flat ambient ASMs*, see JCSS 2012).

Special case: flat ambient ASMs

Only the last declared (most recent) ambient is kept

- instead of an ambient stack (of nested ambients) to which each newly declared environment is PUSHed
  - This was the original definition of ambient ASMs in JCSS 2012, which is generalized by the stack interpretation of ambients.

For flat ambients it suffices to bind the value of the expression in the ambient declaration

- which is computed in the current state (in the current ambient)

to a logical variable, say $curamb$:

$$(\textbf{amb}_{\textbf{flat}}\ exp\ \textbf{in}\ P)^* = (\textbf{let}\ curamb = exp^*\ \textbf{in}\ P^*)$$

- where exp^* is obtained from exp by replacing the names f of ambient dependent functions in exp by f_{curamb}

Inductive transformation of flat ambient ASMs

- *transformation of terms*:
  - for $AmbIndependent(f)$ define:

    $$(f(t_1, \ldots, t_n))^* = f(t_1^*, \ldots, t_n^*)$$

    - logical variables and names of parameterized rules are classified as ambient independent
  - for $AmbDependent(f)$ define:

    $$(f(t_1, \ldots, t_n))^* = f(curamb, t_1^*, \ldots, t_n^*)$$
- *transformation of ambient ASM rules* (to standard ASM programs):
  - for update rules define:

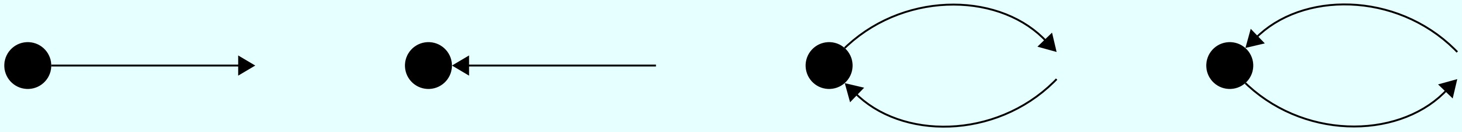
    $$(f(t_1, \ldots, t_n) = t)^* = (f(t_1, \ldots, t_n)^* := t^*)$$
  - for ambient rules define (eliminating $\textbf{amb}_{\textbf{flat}}$):

    $$(\textbf{amb}_{\textbf{flat}}\ exp\ \textbf{in}\ P)^* = \ (\textbf{let}\ curamb = exp^*\ \textbf{in}\ P^*)$$
  - for the other rules use induction on ASM programs, e.g.

    $$(\textbf{let}\ x = t\ \textbf{in}\ P)^* = \ (\textbf{let}\ x = t^*\ \textbf{in}\ P^*)$$

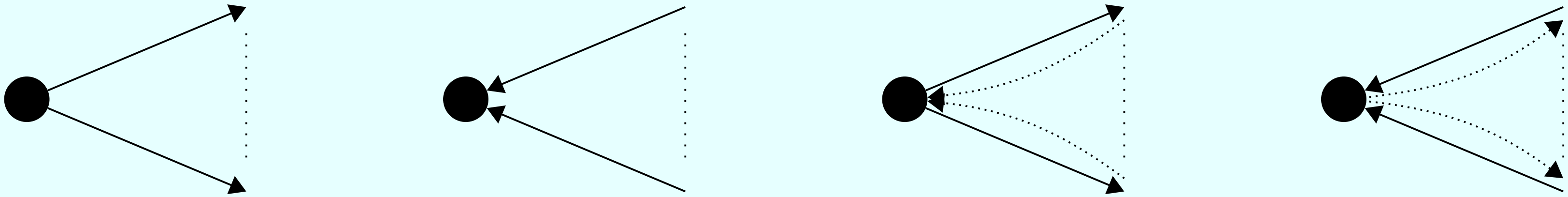
Eight Basic Communication Patterns

Basic bilateral patterns (sender/receiver agent view):



a) Send b) Receive c) Send/Receive d) Receive/Send

Basic multilateral patterns (sender/receiver agent view):



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

From these basic patterns one can compose any bilateral/multiple-parties communication patterns (see Barros/Börger 2005).

Parameters for bilateral communication

Let SEND be the Send action of the communication medium which is used by the Send patterns but left abstract.

The following parameters are considered for bilateral communication:

- whether an *acknowledgement* is requested,
- whether the communication action is *blocking* (in case of reliable delivery), forcing the agent to wait for a response,
- whether the communication action *fails*,
- whether the communication action is *repeated* (unreliable delivery case) until an acknowledgement arrives.

Therefore we have the following Send pattern types:

$$SendType = \{noAck, Ack, AckAwait, UntilAck, UntilAckAwait\}$$

- $noAck$ (resp. Ack) does not (resp. does) expect an acknowledgement
- Ack (resp. $AckAwait$) is not (resp. is) blocking
- $UntilAck$, $UntilAckAwait$ include resending

SendPattern (using communication medium $\textsc{Send}(m)$)

```
$\textsc{SendPattern}(m) =$
  if $ToSend(m)$ then                                  -- trigger predicate at the sender
    if $OkSend(m)$ then  -- an open channel connects sender to receiver
      $\textsc{Send}(m)$
      if $AckRequested(m)$ then $\textsc{SetWaitCond}(m)$
      if $BlockingSend(m)$ then $status := awaitAck(m)$
    else $\textsc{HandleSendFailure}(m, notOkSend)$
    $\textsc{Done}(m)$
where
  $\textsc{Done}(m) = (readyToSend(m) := false)$
  $ToSend(m)$ iff $readyToSend(m) = true$
```

Variations by parameter contraints and component refinements yield the 4 basic bilateral communication patterns above.

$\textsc{Send}_{noAck}$ without acknowledgement

Appropriate for reliable communication medium where messages are neither lost nor corrupted:

$\textsc{Send}_{noAck}(m) = \textsc{SendPattern}(m)$

where

$$AckRequested(m) = false$$

$$BlockingSend(m) = false$$

Non-blocking $\textsc{Send}_{Ack}$ with acknowledgement

Requirements:

- sender must $\textsc{SetWaitCond}$ition for m
- depending on whether sender should be blocked it means that either itself or some other agent should $\textsc{BecomeAwaitHandler}$ for m

```
$\textsc{Send}_{Ack}(m) = \textsc{SendPattern}(m)$
  where
    $AckRequested(m) = true$                      -- constrain $AckRequested$
    $BlockingSend(m) = false$                     -- constrain $BlockingSend$
    $\textsc{SetWaitCond}(m) =$                   -- refine $\textsc{SetWaitCond}$
      let $a = new(Agent)$                        -- create a handler to wait for an Ack
        $\textsc{BecomeAwaitHandler}(a, m)$
```

NB. In the non-blocking case, the $sender$ continues its program execution and the $status(sender)$ does not change.

BecomeAwaitHandler **in** $\textsc{Send}_{Ack}$

```
BecomeAwaitHandler$(a, m)$ =
  $caller(a) :=$ self                                  -- record callback data
  $callerpgm(a) := pgm($self$)$                        -- record callback data
  $pgm(a) :=$ HandleAwaitAck$(m)$
  InitializeAwaitParams$(m)$
where
  InitializeAwaitParams$(m)$ =                         -- placeholder for refinements
    Set$(waitParams(m))$                  -- typically $deadline, resendtime, \ldots$
```

NB. HandleAwaitAck(m) defined below terminates when an acknowledgement message is received.

HandleAwaitAck submachine of BecomeAwaitHandler

- If an ack msg for m arrives, it triggers $\textsc{TerminateAwaitAck}(m)$ which
  - does $\textsc{Unblock}(\mathit{status}(\mathit{caller}(\textbf{self})))$ where needed and terminates the wait action (by a call back or by deleting the handler)
  - possibly performs some more action we keep here as a placeholder for future refinements
- Otherwise we forsee that the handler which must wait for an acknowledement may $\textsc{PerformOtherWaitActivities}(m)$

```
$\textsc{HandleAwaitAck}(m) =$
  if $\mathit{Received}(\mathit{Ack}(m))$ then $\textsc{TerminateAwaitAck}(m)$
    else $\textsc{PerformOtherWaitActivities}(m)$
```

TERMINATEAWAITACK **submachine of** HANDLEAWAITACK

```
TERMINATEAWAITACK($m$) =
  PERFORMACTION($m$)                                      -- to be specified
               -- including UNBLOCK($status(caller($**self**$))$) where needed
  if self = $caller$(self)        -- if sender itself did HANDLEAWAITACK
    then $pgm$(self) := $callerpgm$(self) -- switch to original sender pgm
    else EXIT
  where
    EXIT = DELETE(self, $Agent$)                            -- kill the agent
```

PERFORMOTHERWAITACTIVITIES

Priorities must be established should HANDLEAWAITACK be required to also perform some other wait activities

- triggered by events that could happen simultaneously, like timeouts, failure notice, etc.

We leave them open using the **choose** operator:

```
PERFORMOTHERWAITACTIVITIES($m$) =
  one of
    if $Timeout(m, waitingForAck)$ then
      HANDLETIMEOUT($m, waitingForAck$)
    if $Received(Failed(Ack, m))$ then
      HANDLESENDFAILURE($m, Ack$)
    OTHERWAITRULES($m$)                -- placeholder for extensions
```

Blocking variant $\text{SEND}_{AckAwait}$ of SEND_{Ack} of

It suffices to refine $\text{SETWAITCOND}(m)$ to let the sender itself $\text{BECOMEAWAITHANDLER}(\textbf{self}, m)$

- Therefore the sender is ‘blocked’ (must interrupt the execution of its current program) by switching to $status = awaitAck(m)$

$$\text{SEND}_{AckAwait}(m) = \ \text{SENDPATTERN}(m)$$

where

$$AckRequested(m) = true$$

$$BlockingSend(m) = true$$

$$\text{SETWAITCOND}(m) = \\ \quad \text{BECOMEAWAITHANDLER}(\textbf{self}, m)$$

NB. By the definition of $\text{SENDPATTERN}(m)$:

- $BlockingSend(m) = true$ implies that $status(\textbf{self}) := awaitAck(m)$
- $AckRequested(m) = true$ implies that $\text{SETWAITCOND}(m)$ is called

Unreliable communication: include $\textsc{Resend}$

Refine $\textsc{Send}_{Ack}$ resp. $\textsc{Send}_{AckAwait}$ by including a $\textsc{Resend}(m)$ rule into $\textsc{OtherWaitRules}$ of $\textsc{PerformOtherWaitActivities}(m)$

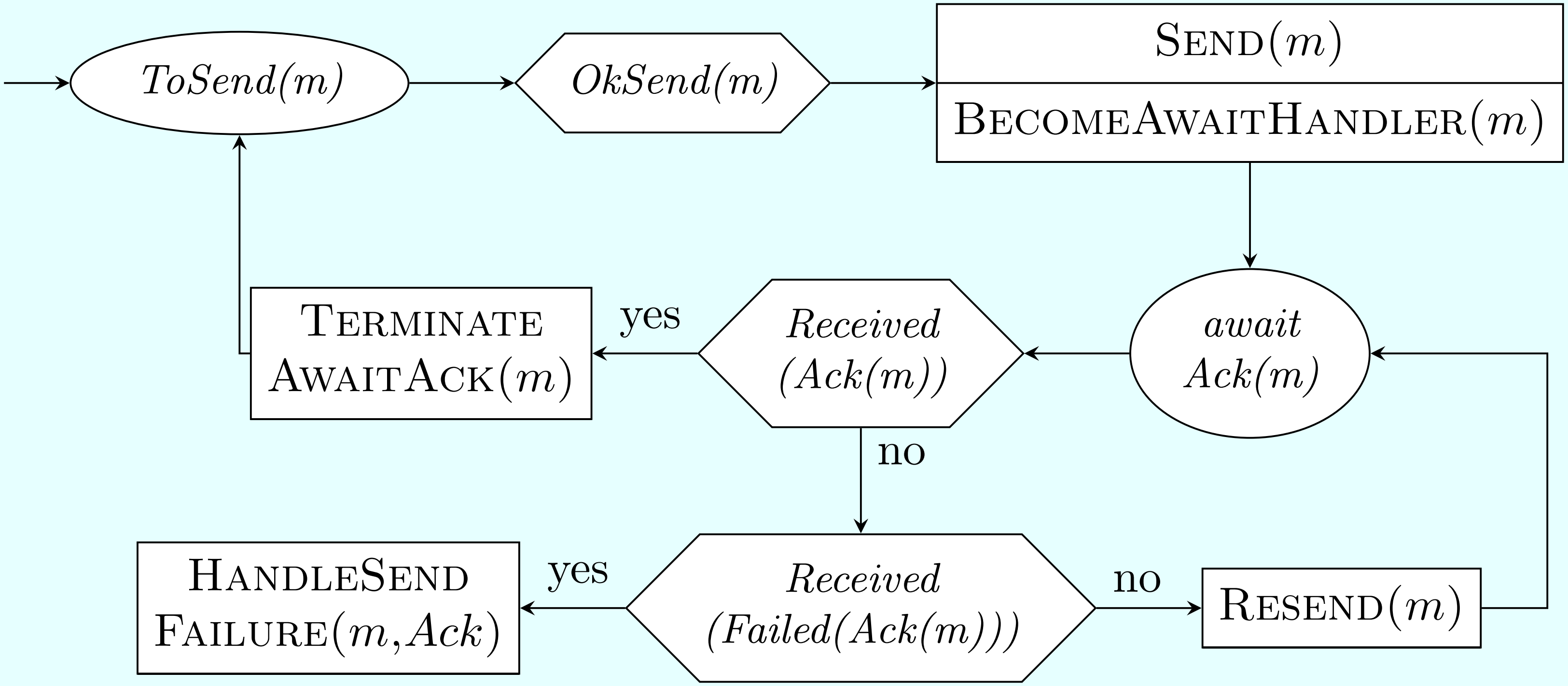
$\textsc{Send}_{UntilAck}(m) = \textsc{Send}_{Ack}(m)$ -- non blocking version

$\textsc{Send}_{UntilAckAwait}(m) = \textsc{Send}_{AckAwait}(m)$ -- blocking version

where

```
  $\textsc{PerformOtherWaitActivities}(m) = \textsc{Resend}(m)$
  $\textsc{Resend}(m) =$
    if $Timeout(resend(m))$ then
      $\textsc{Send}(newVersion(m, now))$        -- copy and original may differ
      $\textsc{SetTimer}(resend(m))$
```

Send$_{UntilAckAwait}$ generalizes Alternating Bit protocol



Mailbox handler requirements

Goal: a scheme for *what a mailbox handler does when messages arrive*.

- The mailbox handler uses a RECEIVE machine
  - kept abstract with the intended interpretation to insert $Arriving$ msgs into the destination agent's $mailbox$.
- We do not treat how the destination agent interacts with its $mailbox$.

For an $Arriving(msg)$, depending on

- whether the mailbox handler is $ReadyToReceive(msg)$
- or if not whether $ToBeDiscarded(msg)$ holds
- or if not whether $ToBeBuffered(msg)$ holds

four $ReceiveType$s appear in two versions, without or with receive acknowledgement request:

$$ReceiveType = \{nonBlocking, blocking, discard, buffer\}$$
$$\cup\{nonBlockingAck, blockingAck, discardAck, bufferAck\}$$

Mailbox handler requirements (Cont'd)

The $ReceiveType$ parameters mean the following:

- $nonBlocking$ requires that $ReadyToReceive(msg)$ is true
- $blocking$ requires that in case the mailbox handler is not $ReadyToReceive(msg)$, the msg should neither be discarded nor buffered so that the machine must wait until $ReadyToReceive(msg)$ becomes true
- $discard$ requires that a msg the mailbox handler is not $ReadyToReceive(msg)$ should be discarded
- $buffer$ requires that a msg the mailbox handler is not $ReadyToReceive(msg)$ should not be discarded but buffered

We skip the machine which performs the $\textsc{FromBufferToMailbox}$ transfer.

RECEIVEPATTERN

```
$\textsc{ReceivePattern}(m) =$
  if $Arriving(m)$ then
    if $ReadyToReceive(m)$ then
      $\textsc{Receive}(m)$
      $\textsc{AckReceive}(m)$
    else if $ToBeDiscarded(m)$ then
      $\textsc{Discard}(m)$
      $\textsc{AckDiscard}(m)$
    else if $ToBeBuffered(m)$ then
      $\textsc{Buffer}(m)$
      $\textsc{AckBuffer}(m)$
```

We keep $\textsc{Receive}(m)$, $\textsc{Discard}(m)$ and $\textsc{Buffer}(m)$ abstract but assume that they include a $\textsc{Consume}(m)$ action

- here the update $Arriving(m) := false$

Submachines of ReceivePattern

```
$\textsc{AckReceive}(m) =$
  if $ToBeAcknowledged(m, receive)$ then
    $\textsc{Send}(Ack(m),$ to $sender(m))$
$\textsc{AckDiscard}(m) =$
  if $ToBeAcknowledged(m, discard)$ then
    $\textsc{Send}(Ack(m, discarded),$ to $sender(m))$
$\textsc{AckBuffer}(m) =$
  if $ToBeAcknowledged(m, buffer)$ then
    $\textsc{Send}(Ack(m, buffered),$ to $sender(m))$
```

How to use $\textsc{ReceivePattern}$ for $\mathit{ReceiveType}$ patterns

Goal: Define for each $t \in \mathit{ReceiveType}$ a $\textsc{Receive}_t$ pattern

$\textsc{Receive}_t$ can be obtained as instance of $\textsc{ReceivePattern}$

- by adding appropriate constraints on the predicates which appear in $\textsc{ReceivePattern}$

The versions with Acknowledgement are obtained by adding to those without Acknowledgement the appropriate condition

$$\mathit{ToBeAcknowledged}(m, \mathit{ofWhat}) = \mathit{true}$$

- where $\mathit{ofWhat} \in \mathit{receive}, \mathit{discard}, \mathit{buffer}$

We therefore define now the four receive patterns without Acknowledgement (where $\mathit{ToBeAcknowledged}(m, \mathit{ofWhat}) = \mathit{false}$).

Instantiating $\textsc{ReceivePattern}$

$$\textsc{Receive}_{nonBlocking}(m) = \textsc{ReceivePattern}(m)$$

where

$$ReadyToReceive(m) = true$$

$$ToBeAcknowledged(m, receive) = false$$

$$\textsc{Receive}_{blocking}(m) = \textsc{ReceivePattern}(m)$$

where

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

if $ReadyToReceive(m) = true$ **then**

$$ToBeAcknowledged(m, receive) = false$$

Instantiating $\textsc{ReceivePattern}$ (Cont'd)

$\textsc{Receive}_{discard}(m) = \textsc{ReceivePattern}(m)$

where

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

$\quad$ **if** $ToBeDiscarded(m)$ **then**

$\qquad ToBeAcknowledged(m, discard) = false$

$\textsc{Receive}_{buffer}(m) = \textsc{ReceivePattern}(m)$

where

$\quad ToBeDiscarded(m) = false$

$\quad ToBeBuffered(m) =$ **not** $ReadyToReceive(m)$

$\quad$ **if** $ToBeBuffered(m)$ **then**

$\qquad ToBeAcknowledged(m, buffer) = false$

A variation of $\textsc{Receive}_{nonBlocking}$

$\textsc{Receive}_{nonBlocking}$ as defined above equals $\textsc{Receive}$.

A variation would be to consider a msg as Received if the mailbox handler is $ReadyToReceive(m)$ or else if it is $ToBeDiscarded(m)$ or else $ToBeBuffered(m)$:

$$\textsc{Receive}_{nonBlocking}(m) = \textsc{ReceivePattern}(m)$$

where

$$ReadyToReceive(m) = true \textbf{ or } ToBeDiscarded(m) = true$$

$$\quad \textbf{or } ToBeBuffered(m) = true$$

$$ToBeAcknowledged(m, receive) =$$

$$\quad ToBeAcknowledged(m, discard) =$$

$$\qquad ToBeAcknowledged(m, buffer) = false$$

Bilateral $\textsc{SendReceive}_{s,t}$ requirements

To constrain agents to receive only responses to a previously sent request, the *responseMsg* and the *reqMsg* must be related

- by a common item of information in the request and the response that allows these two messages to be unequivocally related to one another

One way to achieve this without explicit sequentialization of sender steps is as follows:

- let $SentReqMsg(sender)$ be a set where reqMsgs are recorded
  - as part of a refined $\textsc{Send}_s$
- let the responder relate its $responseMsg$ to a $reqMsg$ by:
  $responseTag(responseMsg) := reqMsg$
  - and include it into $responseMsg$ as part of its refined $\textsc{SendResponse}$
- check $responseTag(m) \in SentReqMsg$ if $Arrived(m)$

Bilateral $\textsc{SendReceive}_{s,t}$ model

$\textsc{SendReceive}_{s,t}(m) =$ **one of** $(\{\textsc{Send}_s(m), \textsc{Receive}_t(m)\})$

where

$Arriving(m)$ iff $Arrived(m)$ **and** $responseTag(m) \in SentReqMsg$

$s \in SendType, t \in ReceiveType$

NB. By Lamport's ordering of communication events (see Comm. ACM 21.7 (1978))

- every sender's $\textsc{Send}(m')$ precedes the receiver's $\textsc{Receive}(m')$
- which precedes the receiver's $\textsc{Send}(m)$ tagged m'
- which precedes the sender's $\textsc{Receive}(m)$.

Bilateral $\text{R}\textsc{eceive}\text{S}\textsc{end}_{t,s}$

- let $\mathit{ReqMsgToAnswer}(\mathit{sender})$ be a set where reqMsgs which require sending an answer are recorded
  - as part of a refined $\text{R}\textsc{eceive}_t$
- let $\mathit{IsAnswer}(m, \mathit{reqMsg})$ express that the message m is the answer to the received reqMsg recorded in $\mathit{ReqMsgToAnswer}$

$$\text{R}\textsc{eceive}\text{S}\textsc{end}_{t,s}(m) = \textbf{one of } (\{\text{R}\textsc{eceive}_t(m), \text{S}\textsc{end}_s(m)\})$$

where

$$\mathit{ToSend}(m) \textbf{ iff}$$

$$\quad \mathit{readyToSend}(m) = \mathit{true} \textbf{ and}$$

$$\quad\quad \textbf{forsome } \mathit{reqMsg} \in \mathit{ReqMsgToAnswer} \;\; \mathit{IsAnswer}(m, \mathit{reqMsg})$$

$$t \in \mathit{ReceiveType}, s \in \mathit{SendType}$$

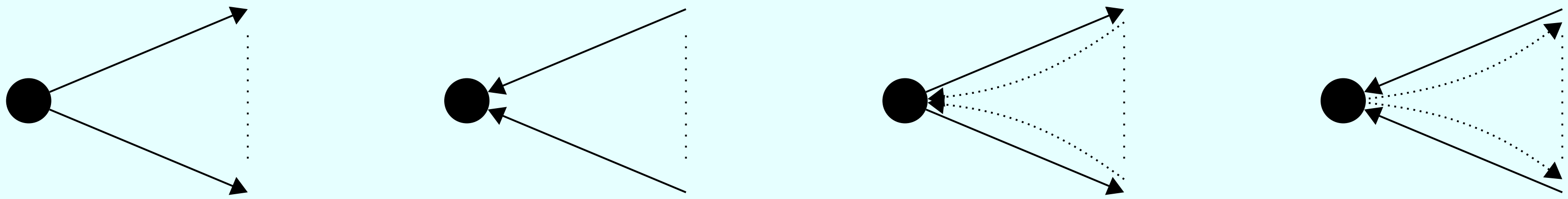
NB. This pattern is used in the web service mediator ($\text{V}\textsc{irtual}\text{P}\textsc{rovider}$) in Ch.5.1 of the Modeling Companion Book.

Basic multilateral communication patterns

Goal: formulate schemes for communication among multiple parties

- where requests are sent to, and responses received from, multiple parties instead of a pair of one sender and one receiver

Considering multiple senders/receivers turns the basic bilateral communication patterns into the following four basic multi-lateral communication patterns:



a) One-to-many send

b) One-from-many receive

c) One-from-many send-receive

d) One-from-many receive-send

From these basic patterns one can compose any communication interaction among multiple parties (see Barros/Börger 2005).

$\text{O}\textsc{ne}\text{T}\textsc{o}\text{M}\textsc{any}\text{S}\textsc{end}$: **Broadcast**

$\text{O}\textsc{ne}\text{T}\textsc{o}\text{M}\textsc{any}\text{S}\textsc{end}_s(m)$ is a refinement of $\text{S}\textsc{end}_s(m)$

- i.e. of $\text{S}\textsc{end}\text{P}\textsc{attern}(m)$ constrained by $t \in \mathit{SendType}$

refining the communication medium $\text{S}\textsc{end}(m)$ *to a* $\text{B}\textsc{roadcast}(m)$

- requiring a (possibly dynamic) set $\mathit{Recipient}(m)$ at the sender side
- msg content may depend on the receiver: ‘instantiate a template (i.e. $\mathit{payload}$) with data that varies from one party to another’

$$\text{O}\textsc{ne}\text{T}\textsc{o}\text{M}\textsc{any}\text{S}\textsc{end}_{noAck}(m) = \text{S}\textsc{end}\text{P}\textsc{attern}(m)$$

where

$$\mathit{AckRequested}(m) = \mathit{BlockingSend}(m) = \mathit{false} \qquad \text{-- } \text{S}\textsc{end}_{noAck}$$

$$\text{S}\textsc{end}(m) = \text{B}\textsc{roadcast}(m)$$

$$\text{B}\textsc{roadcast}(m) =$$

$$\quad \textbf{forall } r \in \mathit{Recipient}(m)\ \text{S}\textsc{end}(\mathit{payload}(m, r), r)$$

Analogously for other $\mathit{SendType}$s than noAck.

OneFromManyReceive

Concept implies correlating arriving messages into correlation $Group$s, say corresponding to a message $type$.

Idea: refine in the ReceivePattern

- $ReadyToReceive(m)$ to whether the corresponding message group is $Accepting$ messages of that type
- Receive action as msg insertion into correlated group

abstracting from further refinable internal group management (group creation, consolidation, closure) and ack requirements (see Barros/Börger 2005)

$$\textsc{OneFromManyReceive}_{discard}(m) = \textsc{Receive}_{discard}(m)$$

where

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

$$\textsc{Receive}(m) = \textsc{Insert}(m, group(type(m)))$$

Analogously for other $ReceiveType$s than $discard$.

OneToManySendReceive

Combine OneToManySend with OneFromManyReceive

- with $\mathit{SentReqMsg}, \mathit{responseTag}$ to relate Send/Receive actions
  - as defined for the corresponding bilateral case SendReceive

$$\textsc{OneToManySendReceive}_{s,t}(m) = \textbf{one of}$$

$$\{\textsc{OneToManySend}_s(m), \textsc{OneFromManyReceive}_t(m)\}$$

where

$$\mathit{Arriving}(m) \text{ iff } \mathit{Arrived}(m) \textbf{ and } \mathit{responseTag}(m) \in \mathit{SentReqMsg}$$

- Internal group management decides when a group of collected responses is sufficient to be taken for further operation as an answer to the broadcasted request (exl: VirtualProvider).
- Timing requirements, for example that responses are expected within a given time frame, concern the internal group management.

This pattern is used in the web service mediator (VirtualProvider) in Ch.5.1 of the Modeling Companion Book.

$\textsc{OneFromManyReceiveSend}$

$responseMsg$ typically formed on the basis of a somehow $Completed$ group of received $reqMsg$s

- $Completed(g)$ expresses group consolidation
- In $IsAnswer$ refine $reqMsg$ by a group, so that:

$$IsAnswer(m, g) \textbf{ iff } m = responseMsg(g)$$

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

$\quad \textsc{OneFromManyReceive}_t(m)$

$\quad \textsc{OneToManySend}_s(m)$

where $ToSend(m)$ **iff** $readyToSend(m) = true$ **and**

$\quad$ **forsome** $g \in Group$ **with** $Completed(g)$ $IsAnswer(m, g)$

Exl: *message routing through a network*

- forwarding (via $\textsc{OneToManySend}$) to $Recipient$s msgs found in $communicator$'s mailbox (via $\textsc{OneFromManyReceive}$)
  - Glässer/Gurevich/Veanes: IEEE Trans. Sw Engg 30 (7) 2004

References

- E. Börger and A. Raschke: Modeling Companion for Software Practitioners. Springer 2018
  http://modelingbook.informatik.uni-ulm.de
  – Ch.4 contains other applications of ambient ASMs for contex-aware system models and further references

On communication patterns:

- A. Barros and E. Börger: A Compositional Framework for Service Interaction Patterns and Communication Flows.
  – LNCS 3785 (2005) 5-35
- U. Glässer and Y. Gurevich and M. Veanes: Abstract communication model for distributed systems.
  – IEEE Trans. Sw Engg 30 (7) 2004

Copyright Notice