Modeling Context-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

Copyright CC BY–NC-SA 4.0
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:

\[ f(x) = f_{\text{params}}(x) \]

in particular when used with implicit (hidden) parameters, supporting conventional implicit oo parameterization \( \text{this}.f(x) = f(x) \)
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: \texttt{amb \ exp \ in \ P}

where to achieve generality in the widest terms
- \texttt{exp} is any expression (term)
- \texttt{P} is any (already defined) ASM program

Intended behavior (see definition below):
- \textbf{Push} the \texttt{eval(exp, S, env, amb)} of the newly declared ambient expression to the current ambient
- execute \texttt{P} in the new ambient

Function classification is extended by ambient (in)dependent fcts.
AmbDependent\((f)\) (wrt Amb) iff
\[
\text{forsome } a, a' \in \text{Amb with } a \neq a' \text{ forsome } x \ f(a, x) \neq f(a', x)
\]
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\((f)\):**

  \[
  \text{eval}(f(t_1, \ldots, t_n), S, \text{env}, \text{amb}) = \\
  f_S(\text{amb}, \text{eval}(t_1, S, \text{env}, \text{amb}), \ldots, \text{eval}(t_n, S, \text{env}, \text{amb}))
  \]

  Here \(f_S\) is turned into a family of possibly different functions \(f_S, \text{amb}\).

- **Case AmbIndependent\((f)\) (unchanged interpretation of \(f\)):**

  \[
  \text{eval}(f(t_1, \ldots, t_n), S, \text{env}, \text{amb}) = \\
  f_S(\text{eval}(t_1, S, \text{env}, \text{amb}), \ldots, \text{eval}(t_n, S, \text{env}, \text{amb}))
  \]
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.

\[
Yields(amb \ exp \ in \ P, S, env, amb, U) \text{ if } Yields(P, S, env, \text{PUSH(eval(exp, S, env, amb), amb), U})
\]

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

\[
(\text{amb}_{\text{flat}} \ exp \ \text{in} \ P)^* = (\text{let} \ \text{curamb} = \ exp^* \ \text{in} \ P^*)
\]

- where \( exp^* \) is obtained from \( exp \) by replacing the names \( f \) of ambient dependent functions in \( exp \) by \( f_{\text{curamb}} \)
Inductive transformation of flat ambient ASMs

- **transformation of terms:**
  - for $\text{AmbIndependent}(f)$ 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 $\text{AmbDependent}(f)$ define:
    \[(f(t_1, \ldots, t_n))^* = f(\text{curamb}, t_1^*, \ldots, t_n^*)\]

- **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 $\text{ambflat}$):
    \[(\text{ambflat} \ exp \ in \ P)^* = (\text{let} \ \text{curamb} = \ exp^* \ in \ P^*)\]
  - for the other rules use induction on ASM programs, e.g.
    \[(\text{let} \ x = t \ in \ P)^* = (\text{let} \ x = t^* \ in \ P^*)\]
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 \textbf{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 \textit{acknowledgement} is requested,
- whether the communication action is \textit{blocking} (in case of reliable delivery), forcing the agent to wait for a response,
- whether the communication action \textit{fails},
- whether the communication action is \textit{repeated} (unreliable delivery case) until an acknowledgement arrives.

Therefore we have the following Send pattern types:

\[
\text{SendType} = \{ \text{noAck, Ack, AckAwait, UntilAck, UntilAckAwait} \}
\]

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

Copyright CC BY–NC-SA 4.0
SendPattern (using communication medium Send(m))

SendPattern(m) =

if ToSend(m) then -- trigger predicate at the sender
  if OkSend(m) then -- an open channel connects sender to receiver
    SEND(m)
    if AckRequested(m) then SETWAITCOND(m)
    if BlockingSend(m) then status := awaitAck(m)
  else HANDLESENDFAILURE(m, notOkSend)
Done(m)

where

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.
**SEND\textsubscript{noAck}** without acknowledgement

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

\[ \text{SEND}_{\text{noAck}}(m) = \text{SEND\textsc{Pattern}}(m) \]

**where**

- \( \text{AckRequested}(m) = \text{false} \)
- \( \text{BlockingSend}(m) = \text{false} \)
Non-blocking $\text{SEND}_{\text{Ack}}$ with acknowledgement

Requirements:

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

$\text{SEND}_{\text{Ack}}(m) = \text{SendPattern}(m)$

where

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

NB. In the non-blocking case, the sender continues its program execution and the $\text{status(sender)}$ does not change.
BecomeAwaitHandler in SendAck

\textbf{BecomeAwaitHandler}(a, m) =

\texttt{caller}(a) := \texttt{self} \quad -- \text{record callback data}

\texttt{callerpgm}(a) := \texttt{pgm}(\texttt{self}) \quad -- \text{record callback data}

\texttt{pgm}(a) := \texttt{HandleAwaitAck}(m)

\textbf{InitializeAwaitParams}(m)

\textbf{where}

\textbf{InitializeAwaitParams}(m) = \quad -- \text{placeholder for refinements}

\texttt{Set(waitParams}(m)) \quad -- \text{typically \texttt{deadline}, \texttt{resendtime}, \ldots}

NB. \texttt{HandleAwaitAck}(m) defined below terminates when an acknowledgement message is received.
If an ack msg for $m$ arrives, it triggers $\text{TerminateAwaitAck}(m)$ which
- does $\text{Unblock}(\text{status}(\text{caller}(\text{self})))$ 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 $\text{PerformOtherWaitActivities}(m)$

\[
\text{HandleAwaitAck}(m) = \\
\quad \text{if } \text{Received}(\text{Ack}(m)) \text{ then } \text{TerminateAwaitAck}(m) \\
\quad \text{else } \text{PerformOtherWaitActivities}(m)
\]
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
Priorities must be established should $\text{HANDLE\_Await\_Ack}$ 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:

$$\text{Perform\_Other\_Wait\_Activities}(m) =$$

```
\text{one of}

\text{if} \ \text{Timeout}(m, \text{waitingForAck}) \ \text{then}
\quad \text{HANDLE\_Timeout}(m, \text{waitingForAck})

\text{if} \ \text{Received}(\text{Failed}(\text{Ack}, m)) \ \text{then}
\quad \text{HANDLE\_Send\_Failure}(m, \text{Ack})

\text{OTHER\_WAIT\_RULES}(m) \quad -- \text{placeholder for extensions}
```
It suffices to refine $\text{SetWaitCond}(m)$ to let the sender itself 
$\text{BecomeAwaitHandler}(\text{self}, m)$

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

$\text{SendAckAwait}(m) = \text{SendPattern}(m)$

where

$\text{AckRequested}(m) = \text{true}$

$\text{BlockingSend}(m) = \text{true}$

$\text{SetWaitCond}(m) =$

$\text{BecomeAwaitHandler}(\text{self}, m)$

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

- $\text{BlockingSend}(m) = \text{true}$ implies that $\text{status}(\text{self}) := \text{awaitAck}(m)$
- $\text{AckRequested}(m) = \text{true}$ implies that $\text{SetWaitCond}(m)$ is called
Unreliable communication: include **RESEND**

Refine $\text{SEND}_{\text{Ack}}\ \text{resp.}\ \text{SEND}_{\text{AckAwait}}$ by including a $\text{RESEND}(m)$ rule into $\text{OTHERWAITRULES}$ of $\text{PERFORMOTHERWAITACTIVITIES}(m)$

$$\text{SEND}_{\text{UntilAck}}(m) = \text{SEND}_{\text{Ack}}(m)$$

-- non blocking version

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

-- blocking version

where

$$\text{PERFORMOTHERWAITACTIVITIES}(m) = \text{RESEND}(m)$$

$$\text{RESEND}(m) =$$

if $\text{Timeout}($resend$(m))$ then

$$\text{SEND}(\text{newVersion}(m, \text{now}))$$

-- copy and original may differ

$$\text{SETTIMER}($resend$(m))$$
Send Until Ack Await generalizes Alternating Bit protocol.
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 *ReceiveTypes* appear in two versions, without or with receive acknowledgement request:

*ReceiveType = \{nonBlocking, blocking, discard, buffer\} \cup \{nonBlockingAck, blockingAck, discardAck, bufferAck\}
The \textit{ReceiveType} parameters mean the following:

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

We skip the machine which performs the \texttt{FromBufferToMailbox} transfer.
\textbf{ReceivePattern} \( (m) = \)

\begin{align*}
\text{if } \text{Arriving}(m) \text{ then} & \\
\quad \text{if } \text{ReadyToReceive}(m) \text{ then} & \\
\quad \quad \text{Receive}(m) & \\
\quad \quad \text{AckReceive}(m) & \\
\text{else if } \text{ToBeDiscarded}(m) \text{ then} & \\
\quad \quad \text{Discard}(m) & \\
\quad \quad \text{AckDiscard}(m) & \\
\text{else if } \text{ToBeBuffered}(m) \text{ then} & \\
\quad \quad \text{Buffer}(m) & \\
\quad \quad \text{AckBuffer}(m) & \\
\end{align*}

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

- here the update \( \text{Arriving}(m) := \text{false} \)
Submachines of ReceivePattern

**AckReceive**($m$) =

\[
\text{if } \text{ToBeAcknowledged}(m, \text{receive}) \text{ then}
\]
\[
\text{Send}(\text{Ack}(m), \text{to sender}(m))
\]

**AckDiscard**($m$) =

\[
\text{if } \text{ToBeAcknowledged}(m, \text{discard}) \text{ then}
\]
\[
\text{Send}(\text{Ack}(m, \text{discarded}), \text{to sender}(m))
\]

**AckBuffer**($m$) =

\[
\text{if } \text{ToBeAcknowledged}(m, \text{buffer}) \text{ then}
\]
\[
\text{Send}(\text{Ack}(m, \text{buffered}), \text{to sender}(m))
\]
How to use \texttt{ReceivePattern} for \texttt{ReceiveType} patterns

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

$\text{Receive}_t$ can be obtained as instance of $\text{ReceivePattern}$
- by adding appropriate constraints on the predicates which appear in $\text{ReceivePattern}$

The versions with \textit{Acknowledgement} are obtained by adding to those without \textit{Acknowledgement} the appropriate condition

$$\text{ToBeAcknowledged}(m, \text{ofWhat}) = \text{true}$$
- where $\text{ofWhat} \in \text{receive, discard, buffer}$

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

\[
\text{Receive}_{\text{nonBlocking}}(m) = \text{ReceivePattern}(m)
\]

where

\[
\text{ReadyToReceive}(m) = \text{true}
\]
\[
\text{ToBeAcknowledged}(m, \text{receive}) = \text{false}
\]

\[
\text{Receive}_{\text{blocking}}(m) = \text{ReceivePattern}(m)
\]

where

\[
\text{ToBeDiscarded}(m) = \text{ToBeBuffered}(m) = \text{false}
\]

if \( \text{ReadyToReceive}(m) = \text{true} \) then

\[
\text{ToBeAcknowledged}(m, \text{receive}) = \text{false}
\]
\[
\text{Receive}_{\text{discard}}(m) = \text{ReceivePattern}(m)
\]
\[\text{where}\]
\[
\text{ToBeDiscarded}(m) = \text{not ReadyToReceive}(m)
\]
\[\text{if ToBeDiscarded}(m) \text{ then}\]
\[
\text{ToBeAcknowledged}(m, \text{discard}) = \text{false}
\]
\[
\text{Receive}_{\text{buffer}}(m) = \text{ReceivePattern}(m)
\]
\[\text{where}\]
\[
\text{ToBeDiscarded}(m) = \text{false}
\]
\[
\text{ToBeBuffered}(m) = \text{not ReadyToReceive}(m)
\]
\[\text{if ToBeBuffered}(m) \text{ then}\]
\[
\text{ToBeAcknowledged}(m, \text{buffer}) = \text{false}
\]
A variation of $\text{Receive}_{\text{nonBlocking}}$

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

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

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

where

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

or $\text{ToBeBuffered}(m) = \text{true}$

$\text{ToBeAcknowledged}(m, \text{receive}) = \text{ToBeAcknowledged}(m, \text{discard}) = \text{false}$
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 `SEND`s
- let the responder relate its `responseMsg` to a `reqMsg` by:
  \[ \text{responseTag(responseMsg)} := \text{reqMsg} \]
  – and include it into `responseMsg` as part of its refined `SENDResponse`
- check `responseTag(m) \in SentReqMsg` if `Arrived(m)`
Bilateral $\text{SendReceive}_{s,t}$ model

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

where

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

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

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

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

Copyright CC BY–NC-SA 4.0
Bilateral \( \text{ReceiveSend}_{t,s} \)

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

\[
\text{ReceiveSend}_{t,s}(m) = \text{one of } \{ \text{Receive}_t(m), \text{Send}_s(m) \}
\]

where

\[
\text{ToSend}(m) \iff \text{readyToSend}(m) = \text{true} \quad \text{and} \quad \text{forsome } \text{reqMsg} \in \text{ReqMsgToAnswer} \quad \text{IsAnswer}(m, \text{reqMsg})
\]

\( t \in \text{ReceiveType}, s \in \text{SendType} \)

NB. This pattern is used in the web service mediator (\text{VirtualProvider}) 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).
OneToManySend: Broadcast

OneToManySend_s(m) is a refinement of Send_s(m)
- i.e. of SendPattern(m) constrained by \( t \in \text{SendType} \)

refining the communication medium Send(m) to a Broadcast(m)
- requiring a (possibly dynamic) set Recipient(m) at the sender side
- msg content may depend on the receiver: ‘instantiate a template (i.e. payload) with data that varies from one party to another’

OneToManySend_{noAck}(m) = SendPattern(m)

where

\[ \text{AckRequested}(m) = \text{BlockingSend}(m) = \text{false} \]

Send(m) = Broadcast(m)

Broadcast(m) =

\[ \forall r \in \text{Recipient}(m) \text{ Send}(\text{payload}(m, r), r) \]

Analogously for other SendTypes than noAck.
Concept implies correlating arriving messages into correlation Groups, say corresponding to a message type.

Idea: refine in the ReceivePattern

- \textit{ReadyToReceive}(m) to whether the corresponding message group is Accepting messages of that type
- \textit{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)

\textbf{OneFromManyReceive} \textit{discard}(m) = \textit{Receive} \textit{discard}(m)

where

\begin{align*}
\textit{ReadyToReceive}(m) &= \text{Accepting}(\textit{group}(\text{type}(m))) \\
\textit{Receive}(m) &= \text{Insert}(m, \textit{group}(\text{type}(m)))
\end{align*}

Analogously for other ReceiveTypes than \textit{discard}. 
Combine `OneToManySend` with `OneFromManyReceive` with `SentReqMsg`, `responseTag` to relate Send/Receive actions – as defined for the corresponding bilateral case `SendReceive`

**OneToManySendReceive**

\[ \text{OneToManySendReceive}_{s,t}(m) = \text{one of} \]

\[ \{ \text{OneToManySend}_{s}(m), \text{OneFromManyReceive}_{t}(m) \} \]

where

\[ \text{Arriving}(m) \text{ iff } \text{Arrived}(m) \text{ and } \text{responseTag}(m) \in \text{SentReqMsg} \]

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.
responseMsg typically formed on the basis of a somehow Completed group of received reqMsgs

- Completed(g) expresses group consolidation
- In IsAnswer refine reqMsg by a group, so that:
  \[ \text{IsAnswer}(m, g) \text{ iff } m = \text{responseMsg}(g) \]

OneFromManyReceiveSend \(_{t,s}(m) = \)
OneFromManyReceive\(_t(m) \)
OneToManySend\(_s(m) \)

where \( \text{ToSend}(m) \text{ iff } \text{readyToSend}(m) = \text{true and} \)

forsome \( g \in \text{Group with } \text{Completed}(g) \text{ IsAnswer}(m, g) \)

Exl: *message routing through a network*

- forwarding (via OneToManySend) to *Recipients* msgs found in communicator’s mailbox (via OneFromManyReceive)
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 context-aware system models and further references

On communication patterns:

  – 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
It is permitted to (re-) use these slides under the CC-BY-NC-SA licence

https://creativecommons.org/licenses/by-nc-sa/4.0/

i.e. in particular under the condition that
- the original authors are mentioned
- modified slides are made available under the same licence
- the (re-) use is not commercial