A Compositional Framework for Service Interaction Patterns and Interaction Flows

Alistair Barros[1] and Egon Börger[2]

[1] SAP Research Centre Brisbane, Australia alistair.barros@sap.com
[2] Università di Pisa, Dipartimento di Informatica, I-56125 Pisa, Italy boerger@di.unipi.it

Abstract. We provide precise high-level models for eight fundamental service interaction patterns, together with schemes for their composition into complex service-based business process interconnections and interaction flows, supporting software-engineered business process management in multi-party collaborative environments. The mathematical nature of our models provides a basis for a rigorous execution-platform-independent analysis, in particular for benchmarking web services functionality. The models can also serve as accurate standard specifications, subject to further design leading by stepwise refinement to implementations.

We begin by defining succinct rigorous models to mathematically capture the behavioral meaning of four basic bilateral business process interaction patterns (Sect. 1), together with their refinements to four basic multilateral interaction patterns (Sect. 2). We then illustrate with characteristic examples how by appropriate combinations and refinements of these eight fundamental patterns one can define arbitrarily complex interaction patterns of distributed service-based business processes that go beyond simple request-response sequences and may involve a dynamically evolving number of participants. This leads to a definition of the concept of process interaction flow or conversation, namely via multi-agent distributed interaction pattern runs (Sect. 3). We point to various examples in the literature on web-service-oriented business process management, which illustrate the models and concepts defined here.

We start from the informal business process interaction pattern descriptions in [2][1], streamlining, generalizing or unifying them where the formalization suggested to do so. Our models provide for them an accurate high-level view one can consider as *ground model* (blueprint) definition, in the sense defined in [4], clarifying open issues and apt to direct the further detailing, by stepwise refinement in the sense defined in [5], to an executable version as for example BPEL code. Since for the semantics of the forthcoming BPEL standard a formal model has been provided in [15,24,13,14], such refinements can be mathematically investigated to *prove* their correctness with respect to the ground model, thus preparing for the application of verifying compiler techniques [17,10].

For the modeling we use the Abstract State Machines (ASMs) method with its ASM ground model and ASM refinement techniques, extending the pattern description scheme outlined in [8,9]. The support the ASM notion offers to express the dynamics of abstract state changes allows us to provide a high-level state-based view of service inter-*action* patterns, where the behavioral interface is defined through pattern actions performed by submachines, which remain largely abstract due to the intention to leave the design space open for further refinements to concrete pattern instantiations. Most of what we use below to model service interaction patterns by ASMs is self-explanatory, given the semantically well-founded pseudo-code character of ASMs, an extension of Finite State Machines (FSMs) by a general notion of state. For the sake of completeness we sketch in the appendix (Section 5) what is needed for a correct understanding: the simple semantics of ASMs as extension of FSMs by generalized memory locations together with the ASM classification of locations and functions that supports modularity in high-level system descriptions. A recent tutorial introduction into the ASM method for high-level system design and analysis is available in [7]. For a more detailed textbook presentation of the method see the AsmBook [12].

[1] All the not furthermore qualified quotes in this paper are from there.

1 Basic components of bilateral interaction patterns

The basic bilateral (one-to-one) interaction patterns we have identified are characterized by four component type ASMs, refinements of which suffice to compose any other bilateral interaction pattern of whatever structural complexity: *Send* and *Receive* and their sequential combinations *SendReceive* (for sending a request followed by receiving a response) and *ReceiveSend* (for receiving a request followed by sending a response). Each of these pattern components describes one side of an interaction, as illustrated in Figure 1, so that all the basic bilateral interaction pattern ASMs we define in this section are mono-agent machines or modules.

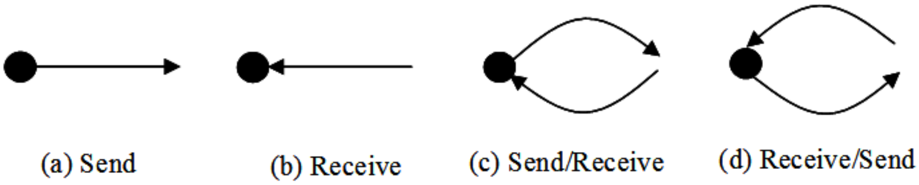


Fig. 1. Basic Bilateral Interaction Pattern Types

1.1 Pattern Send

Different versions for sending are considered, depending on whether the delivery is reliable (guaranteed) or not and whether, in case of reliable delivery, the action is blocking or non-blocking. Also the possibility is contemplated that the send action may result in a fault message in response or that a periodic resending of a message is performed.

For each version it is required that the counter-party may or may not be known at design time. This is reflected by the following possibly dynamic function, associating a recipient to each message. In case the recipient depends on further parameters, one has to refine $recipient(m)$ by adding those parameters to the function to determine $recipient(m, param)$.

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

All considered types of the send pattern use an abstract machine $\textsc{BasicSend}(m)$ with the intended interpretation that message m is sent to $recipient(m)$, or to $recipient(m, param)$ in case some additional recipient *param*eters are given by the context. Some patterns also use the notation $\textsc{BasicSend}(m, r)$ where $r = recipient(m, param)$. This abstraction will be refined below, e.g. to capture broadcasting instead of bilateral sending. It also reflects that the underlying message delivery system is deliberately left unspecified.

To indicate that a faulty behavior has happened at the receiver's side as result of sending message m, we use an abstract monitored predicate $Faulty(m)$ with the intended interpretation that a fault message in response has arrived. Possible faults originating at the sender's side, during an attempt to send a message m, are captured by a $\textsc{SendFaultHandler}$, typically triggered by a condition **not** $OkSend(m)$. A typical refinement of $OkSend(m)$ would be that there exists a channel, connecting the sender to the recipient, which is open to send m.

We therefore have two abstract methods, a machine which does a $\textsc{FirstSend}(m)$ without further resending and a machine to $\textsc{HandleSendFault}(m)$. These two machines use, as guards for being triggered, abstract monitored predicates $SendMode(m)$ respectively $SendFaultMode(m)$. A typical assumption on the underlying scheduler for calling these machines will be that for each m, $SendFaultMode(m)$ can become true only after $SendMode(m)$ has been true.

To formalize sending messages whose delivery is requested to be guaranteed by an acknowledgement, a machine SetWaitCondition will typically Initialize a shared predicate $\mathit{WaitingFor}(m)$, whose role is to record that the sender is still waiting for an acknowledgement which informs that m has been delivered. In case of a $\mathit{BlockingSend}(m)$, the *blocking* effect is formalized by setting $\mathit{status} := \mathit{blocked}(m)$. Here *status* itself is not parameterized by m given that its role is to possibly block the Send machine from further sending out other messages (see below the discussion of the blocking case).

```
FirstSend($m$) = if $\mathit{SendMode}(m)$ then [2]
  if $\mathit{OkSend}(m)$ then
    BasicSend($m$)
    if $\mathit{AckRequested}(m)$ then SetWaitCondition($m$)
    if $\mathit{BlockingSend}(m)$ then $\mathit{status} := \mathit{blocked}(m)$
```

```
HandleSendFault($m$) = if $\mathit{SendFaultMode}(m)$ then SendFaultHandler($m$)[3]
```

As typical assumption $\mathit{SendMode}(m)$ **and not** $\mathit{OkSend}(m)$ implies $\mathit{SendFaultMode}(m) = \mathit{true}$.

Send&Check = {FirstSend(m), HandleSendFault(m)}[4]

Send Without Guaranteed Delivery. For the instantiation of Send&Check to Send_{noAck} it suffices to require *AckRequested* and *BlockingSend* to be always false.

```
MODULE  Send$_{noAck}$ = Send&Check
  where [5]  forall $m$ $\mathit{AckRequested}(m) = \mathit{BlockingSend}(m) = \mathit{false}$
```

Guaranteed Non-Blocking Send. For the instantiation of Send&Check to $\text{Send}_{ackNonBlocking}$ with guaranteed delivery, but without blocking effect, it suffices to require *AckRequested* to be always true resp. *BlockingSend* to be always false and to further detail the abstract submachine SetWaitCondition(m). This machine has to Set various deadlines and to Initialize the predicate $\mathit{WaitingFor}(m)$,[6] which is reset to *false* typically through an action of the $\mathit{recipient}(m)$ upon receipt of m, e.g. by sending an acknowledgement message under a reliable messaging protocol[7]. Among the various deadlines occurring in different patterns we mention here $\mathit{deadline}(m)$ and $\mathit{sendTime}(m)$, which are typically used to define $\mathit{Timeout}(m)$ by $(\mathit{now} - \mathit{sendTime}(m) > \mathit{deadline}(m))$, using a system time function *now* to which $\mathit{sendTime}(m)$ is set in Set. We also mention a function $\mathit{frequency}(m)$ which will help to define the frequency of events expected by the sender, e.g. the arrival of response messages or the $\mathit{ResendTime}(m)$ at which m has to be resent periodically (see below). A frequent requirement on the scheduler is that $\mathit{SendFaultMode}(m)$ is implied by a $\mathit{Timeout}(m)$, although some patterns come with Timeout concepts which are not related to faulty message sending and therefore trigger other machines than TimeoutFaultHandlers.

[2] For notational succinctness we assume the firing of this rule to be preemptive. This means that when the rule is applied because $\mathit{SendMode}(m)$ became true, $\mathit{SendMode}(m)$ becomes false as a result of this application. Usually such an preemptiveness assumption is automatically guaranteed through further refinement steps.

[3] As for FirstSend(m) we assume also the firing of HandleSendFault(m) to be preemptive.

[4] By this notation we indicate that Send&Check consists of the two methods FirstSend and HandleSendFault, callable for any legal call parameter m, whatever may be the parameter passing mechanism.

[5] Notationally we use unrestricted quantifiers, assuming that their underlying range is clear from the context.

[6] This predicate reflects the not furthermore specified message delivery system. In some cases below it will be refined by providing further details for its definition.

[7] The limit case is possible that Initialize($\mathit{WaitingFor}(m)$) sets $\mathit{WaitingFor}(m) := \mathit{false}$ in case of immediate and safe delivery.

```
$MODULE\ \ \textsc{Send}_{ackNonBlocking} = \textsc{Send\&Check}$
where
  forall $m$ $AckRequested(m) = true$ and $BlockingSend(m) = false$
  $\textsc{SetWaitCondition}(m) =$
    $\textsc{Initialize}(WaitingFor(m))$
    $\textsc{Set}(deadline(m), sendTime(m), frequency(m), \ldots)$
```

Guaranteed Blocking Send. For the instantiation of $\textsc{Send\&Check}$ to $\textsc{Send}_{ackBlocking}$ with guaranteed delivery and *blocking* effect, we require both *AckRequested* and *BlockingSend* to be always true, refine $SendMode(m)$ to $status = readyToSend$ and add a submachine $\textsc{UnblockSend}(m)$. Its role is to switch back from $blocked(m)$ to an unblocked status, typically *readyToSend*, and to $\textsc{PerformAction}(m)$ to be taken upon the end of the waiting period.[8]

For a succinct formulation of the refinement of $\textsc{SetWaitcondition}$ we use the following notation introduced in [6]: M **addRule** R denotes the parallel composition of M and R.

```
$M$ addRule $R =$
  $M$
  $R$
```

To avoid confusion among different machines, which occur as submachine of machines N, N' but within those machines carry the same name M, we use indexing and write M_N respectively $M_{N'}$.

```
$MODULE\ \ \textsc{Send}_{ackBlocking} = \textsc{Send\&Check} \cup \{\textsc{UnblockSend}(m)\}$
where
  forall $m$ $AckRequested(m) = BlockingSend(m) = true$
  $SendMode(m) \ = \ (status = readyToSend)$
  $\textsc{SetWaitCondition}(m) = \textsc{SetWaitCondition}_{\textsc{Send}_{ackNonBlocking}}(m)$
    addRule $status := blocked(m)$
  $\textsc{UnblockSend}(m) =$ if $UnblockMode(m)$ then
    $\textsc{Unblock}(status)$
    $\textsc{PerformAction}(m)$
  $UnblockMode(m) \ = \ (status = blocked(m)$ and not $WaitingFor(m))$
  $SendFaultMode(m) \ = \ (Faulty(m)$ and $status = blocked(m)$ and $WaitingFor(m))$
```

Send with Resending. In case one wants a still not delivered message to be resent from time to time, it suffices to periodically trigger an additional machine $\textsc{ReSend}(m)$ [9] until the $WaitingFor(m)$ value has changed to *false*—or a $Faulty(m)$ event triggers $\textsc{HandleSendFault}(m)$, which typically is assumed to stop $\textsc{ReSend}$ing. We write the pattern for any *SendType* considered above, namely $t \in \{noAck, ackNonBlocking, ackBlocking\}$. We foresee that message copies are variations $newVersion(m, now)$ of the original m, where the variation may depend on the current time *now*.

```
$MODULE\ \ \textsc{Send}_{tResend} = \textsc{Send}_t \cup \{\textsc{ReSend}(m)\}$
where
  $\textsc{ReSend}(m) =$ if $ResendMode(m)$ then
    $\textsc{BasicSend}(newVersion(m, now))$
    $lastSendTime(m) := now$
  $ResendMode(m) \ = \ ResendTime(m)$ and $WaitingFor(m)$
```

[8] We use here an abstract machine $\textsc{Unblock}(status)$ instead of $status := readyToSend$ to be prepared for further refinements of $\textsc{PerformAction}$ which could include *status* updates, in coordination with a corresponding refinement of $\textsc{Unblock}(status)$.

[9] The period is determined by a predicate $ResendTime(m)$, which typically is defined in terms of the function $lastSendTime(m)$ and the monitored system time *now*.

For a pictorial representation of $\textsc{Send}_{ackBlockingResend}$ see Figure 2. It generalizes the Alternating Bit Sender control state ASM diagram in [p.243][12].

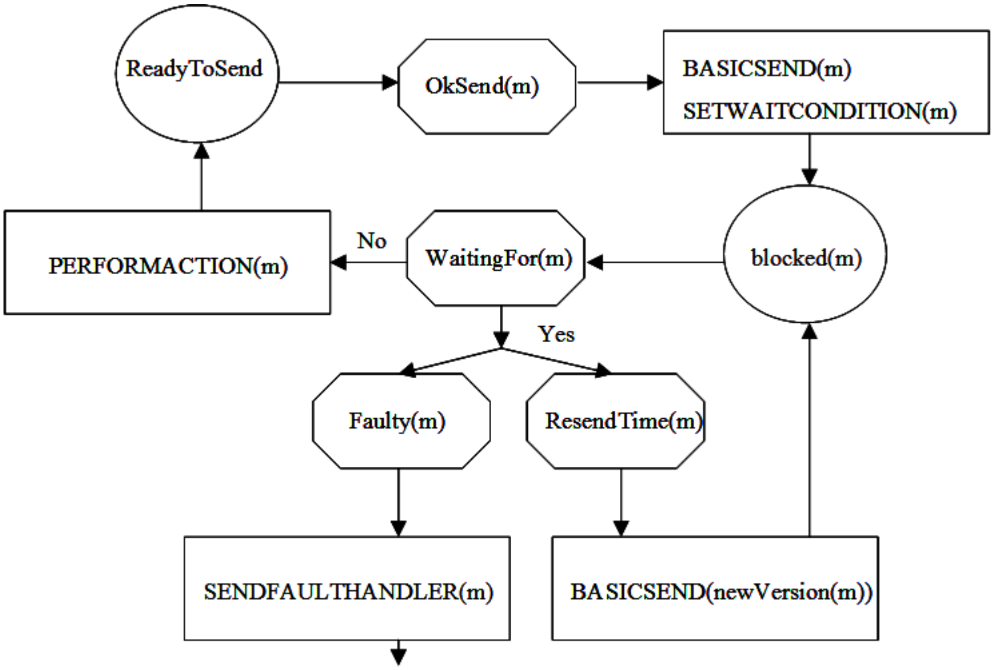


Fig. 2. Blocking Send with Acknowledgement and Resend

We reassume here the definition of the set of *SendTypes* considered in this paper:

$$\begin{aligned} &SendType = \{noAck, ackNonBlocking, ackBlocking\} \\ &\quad \cup \{noAckResend, ackNonBlockingResend, ackBlockingResend\} \end{aligned}$$

1.2 Pattern Receive

We formalize a general form for receiving messages, which can be instantiated to the different versions discussed in [2], depending on whether the action is blocking or non-blocking, whether messages which upon arrival cannot be received are buffered for further consumption or discarded and whether an acknowledgement is required or not. Also the possibility is contemplated that the receive action may result in a fault message.

For each version it is required that the party from which the message will be received may or may not be known at design time. This is reflected by the following possibly dynamic function, associating a sender to each message.

$$sender\colon Message \rightarrow Sender$$

We use abstract predicates checking whether a message m is *Arriving*[10] and *ToBeAcknowledged* and whether our machine is $ReadyToReceive(m)$ and in case it is not whether the message is *ToBeDiscarded* or *ToBeBuffered*, in which cases the action is described by abstract machines to $\textsc{Consume}(m)$, $\textsc{Discard}(m)$, $\textsc{Buffer}(m)$ or to send an $Ack(m)$ message or a $faultMsg(m)$ to the $sender(m)$. For the buffering submachine we also forsee the possibility that upon *DequeueTime*,

[10] The intended interpretation of $Arriving(m)$ is that m is in the message channel or in the message buffer.

which is typically defined in terms of the *enqueueTime* of messages, a Dequeue action is required. We leave it as part of the here not furthermore specified submachines $\textsc{Discard}(m)$ and $\textsc{Enqueue}(m)$ to acknowledge by $\textsc{BasicSend}(\mathit{discardOrBufferMsg}(m), \mathit{sender}(m))$ a received but discarded or buffered message m where required. Similarly one may consider sending a further acknowledgement as part of the Dequeue submachine.

```
$\textsc{Receive}(m) =$ if $\mathit{Arriving}(m)$ then
  if $\mathit{ReadyToReceive}(m)$ then
    $\textsc{Consume}(m)$
    if $\mathit{ToBeAcknowledged}(m)$ then $\textsc{BasicSend}(\mathit{Ack}(m), \mathit{sender}(m))$
  elseif $\mathit{ToBeDiscarded}(m)$ then
    $\textsc{Discard}(m)$
  else $\textsc{Buffer}(m)$
where $\textsc{Buffer}(m) =$
  if $\mathit{ToBeBuffered}(m)$ then
    $\textsc{Enqueue}(m)$
    $\mathit{enqueueTime}(m) := \mathit{now}$
  if $\mathit{DequeueTime}$ then $\textsc{Dequeue}$
```

Remark. Note that Consume and Discard are typically realized at the application (e.g. BPEL) level, whereas $\textsc{Buffer}(m)$ belongs to the system level and is usually realized through lower level middleware functionality.

Instances of $\textsc{Receive}(m)$. It is now easy to define special versions of Receive by restricting some of the abstract guards.
$\textsc{Receive}_{blocking}$ can be defined as Receive where no message is discarded or buffered, so that for an $\mathit{Arriving}(m)$ that is not $\mathit{ReadyToReceive}(m)$, the machine is 'blocked' (read: cannot fire its rule for this m) until it becomes $\mathit{ReadyToReceive}(m)$, formally speaking where there is no *DequeueTime* and for each message m holds:

$$\mathit{ToBeDiscarded}(m) = \mathit{false} = \mathit{ToBeBuffered}(m).$$

$\textsc{Receive}_{discard}$ can be defined similarly as Receive where arriving messages, if they cannot be received at arrival time, are discarded, so that for each message m holds:

$$\mathit{ReadyToReceive}(m) = \mathit{false} \Rightarrow \mathit{ToBeDiscarded}(m) = \mathit{true}.$$

$\textsc{Receive}_{buffer}$ can be defined as Receive where arriving messages, if they cannot be received at arrival time, are buffered, formally where for each message m holds:

$$\mathit{ReadyToReceive}(m) = \mathit{false} \Rightarrow \\ \mathit{ToBeDiscarded}(m) = \mathit{falseToBeBuffered}(m) = \mathit{true}.$$

For each of the preceding Receive instances one can define a version $\textsc{Receive}_{ack}$ with acknowledgement, namely by requiring that for each message m it holds that $\mathit{ToBeAcknowledged}(m) = \mathit{true}$; analogously for a version $\textsc{Receive}_{noAck}$ without acknowledgement, where it is required that $\mathit{ToBeAcknowledged}(m) = \mathit{false}$ holds for each message m.

We reassume here the definition of the set of *ReceiveType*s considered above, where we add the distinction among those with and those without acknowledgement, depending on whether $\mathit{ToBeAcknowledged}(m)$ is true or not for every m:

$$\mathit{ReceiveType} = \{\mathit{blocking}, \mathit{buffer}, \mathit{discard}\} \cup \\ \{\mathit{noAckBlocking}, \mathit{noAckBuffer}, \mathit{ackBlocking}, \mathit{ackBuffer}\}$$

1.3 Pattern Send/Receive

This pattern is about receiving a response to a previously sent request. One can define this pattern as a combination of the machines for the send and the receive pattern. The requirement of "a common item of information in the request and the response that allows these two messages to be unequivocally related to one another" is captured by two dynamic predicates[11] *RequestMsg* and *ResponseMsg* with a function *requestMsg*, which identifies for every $m \in \mathit{ResponseMsg}$ the $\mathit{requestMsg}(m) \in \mathit{RequestMsg}$ to which m is the *responseMsg*.[12]

For the non-blocking version of the pattern, sending a request message is made to precede the call of Receive for the response message m by refining the Receive-guard $\mathit{Arriving}(m)$ through the condition that the message which *Arrived* is a *ResponseMsg*. If no acknowledgement of the response is requested, it suffices to require $\mathit{ToBeAcknowledged}(m) = \mathit{false}$ for each m. Another natural assumption is that after having Initialized $\mathit{WaitingFor}(m)$ through $\text{FirstSend}(m)$, $\mathit{WaitingFor}(m)$ is set at the $\mathit{recipient}(m)$ to *false* in the moment $\mathit{responseMsg}(m)$ is defined. For the blocking version this assumption guarantees that both Receive and UnblockSend can be called for the $\mathit{responseMsg}(m)$. We formulate the Send/Receive pattern for any pair (s, t) of *SendType* and *ReceiveType*.

$$\mathit{MODULE}\ \text{SendReceive}_{s,t} = \text{Send}_s \cup \{\text{Receive}_t(m)\}$$
where
$$\mathit{Arriving}(m) = \mathit{Arrived}(m)\ \textbf{and}\ m \in \mathit{ResponseMsg}$$
$$\mathit{ResponseMsg} = \{m \mid m = \mathit{responseMsg}(\mathit{requestMsg}(m))\}$$

1.4 Pattern Receive/Send

This pattern is a dual to the Send/Receive pattern and in fact can be composed out of the same constituents, but with different refinements for some of the abstract predicates to let receiving a request precede sending the answer. The different versions of the pattern are reflected by the different versions for the constituent machines for the Receive or Send pattern. The refinement of $\mathit{SendMode}(m)$ by adding the condition $m \in \mathit{ResponseMsg}$ guarantees that sending out an answer message is preceded by having received a corresponding request message, a condition which is represented by a predicate *ReceivedMsg*.

$$\mathit{MODULE}\ \text{ReceiveSend}_{t,s} = \{\text{Receive}_t(m)\} \cup \text{Send}_s$$
where
$$\mathit{SendMode}(m) = \mathit{SendMode}_t(m)\ \textbf{and}\ m \in \mathit{ResponseMsg}$$
$$\mathit{ResponseMsg} = \{\mathit{responseMsg}(m) \mid \mathit{ReceivedMsg}(m)\}$$

An example of this bilateral service interaction pattern appears in the web service mediator model defined in [1], namely in the pair of machines to ReceiveREQuests and to SendAnsWers.

2 Basic multilateral interaction patterns (Composition of basic bilateral interaction patterns)

In this section four basic multi-party interaction patterns are identified for each of the four basic bilateral interaction pattern ASMs of the previous section, namely by allowing multiple recipients or senders: *Send* messages (representing requests or responses) to multiple recipients, *Receive* responses or requests from multiple senders, *SendReceive* to send requests to multiple interaction partners followed by receiving responses from them, similarly *ReceiveSend* to receive requests from multiple interaction partners followed by sending responses to them.

[11] We identify sets with unary predicates.

[12] This view of *ResponseMsg* can be turned into a refined view by distinguishing at each agent *ReceivedResponseMsg*es from *ToBeSentResponseMsg*es.

Each of these patterns describes one side of the interaction, as illustrated in Figure 3, so that all the components we define in this section are mono-agent ASM machines or modules. Refinements of the identified basic multilateral interaction pattern ASMs suffice to compose any other multilateral interaction pattern of whatever structural complexity.

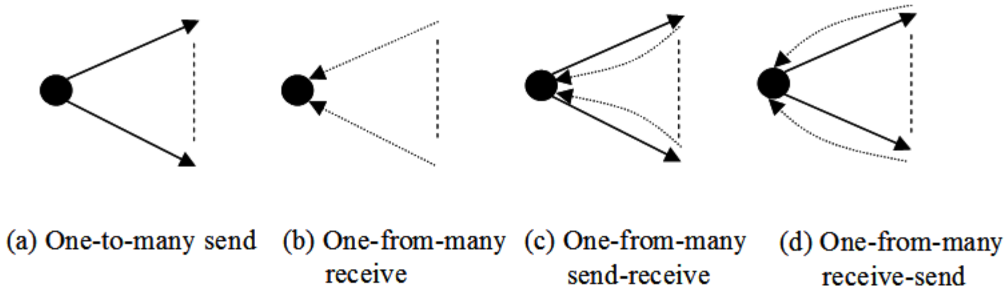

(a) One-to-many send (b) One-from-many receive (c) One-from-many send-receive (d) One-from-many receive-send

Fig. 3. Basic Multilateral Interaction Pattern Types

2.1 One-to-many Send Pattern

This pattern describes a broadcast action where one agent sends messages to several recipients. The requirement that the number of parties to whom a message is sent may or may not be known at design time is reflected by having a dynamic set *Recipient*. The condition that the message contents may differ from one recipient to another can be captured by a dynamic function *msgContent* for "instantiating a template with data that varies from one party to another".

$$\mathit{msgContent}\colon \mathit{MsgTemplate} \times \mathit{Recipient} \to \mathit{Message}$$

Variations of the pattern can be captured by refining the abstract predicates like $\mathit{FaultMode}(m)$ or $\mathit{SendMode}(m)$ accordingly[13] and refining the $\textsc{BasicSend}$ component by **forall** $r \in \mathit{Recipient}$ **do** $\textsc{AtomicSend}_{\mathit{type}(m,r)}(\mathit{msgContent}(m,r))$, where the new abstract machine $\textsc{AtomicSend}$ plays the role of the atomic sending mechanism for broadcasting messages to multiple recipients. We forsee that it can be of a *type* which depends on messages m and their recipients r.

```
$\textsc{OneToManySend}_s = \textsc{Send}_s$
**where**
  $\textsc{BasicSend}(m) = $ **forall** $r \in \mathit{Recipient}(m)$ **do** $\textsc{AtomicSend}_{\mathit{type}(m,r)}(\mathit{msgContent}(m,r))$
```

2.2 One-from-many Receive Pattern

This pattern describes correlating 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. "The arrival of messages needs to be *timely* enough for their correlation as a single logical request." The pattern can be defined by adding to a refinement of $\textsc{Receive}$ a new module $\textsc{GroupRules}$ whose components manipulate groups by creating, consolidating and closing them.

[13] For example $\mathit{SendMode}(m)$ could be refined for the guaranteed blocking send by stipulating $\mathit{SendMode}(m) \equiv \mathit{status} = $ **forall** $r \in \mathit{Recipient}\ \mathit{ReadyToSendTo}(m,r)$ and refining $\mathit{WaitingFor}(m)$ to **forall** $r \in \mathit{RecipientExpectedToAnswer}(m)\ \mathit{WaitingFor}(m,r)$ or to **forsome** $r \in \mathit{RecipientExpectedToAnswer}(m)\ \mathit{WaitingFor}(m,r)$.

The refinement of Receive consists first of all in adapting the predicate $\mathit{ReadyToReceive}(m)$ to mean that the current state is *Accepting* the type of arriving message. The $\mathit{type}(m)$ serves to "determine which incoming messages should be grouped together". The machine Consume(m) is detailed to mean that m is accepted by and put into its current (possibly newly created) correlation group *currGroup*, which depends on the $\mathit{type}(m)$. By the constraint that a message, to be inserted into its correlation group, has to be accepted by its *currGroup*, we reflect a stop condition, which is needed because in the pattern "the number of messages to be received is not necessarily known in advance". The definition of $\mathit{ToBeDiscarded}(m)$ as the negation of $\mathit{Accepting}(m)$ reflects that no buffering is foreseen in this scheme.

The machine CreateGroup(type) to create a correlation group of the given *type* reflects that following the pattern requirements, groups can be created not only upon receipt of a first message (namely as part of Consume), but "this can occur at any time". To InitializeGroup for a newly created element $g \in \mathit{Group}(t)$ comes up to make it the *currGroup* of $\mathit{type}(g) = t$, *Accepting* and with the $\mathit{timer}(g)$ set to the current system time *now*.

The group closure machines CloseCurrGroup and CloseGroup reset *Accepting* for their argument (namely *currGroup* or a group type) to *false* in case a *Timeout*, imposed on the correlation process, or a group completion event respectively a *ClosureEvent* for a correlation type does occur.

To Consolidate a group g upon its completion into a single result, the two cases are forseen that the correlation "may complete successfully or not depending on the set of messages gathered" in g, wherefore we use abstract machines to ProcessSuccess(g) or ProcessFailure(g).

```
MODULE OneFromManyReceive_t = {Receive_t} ∪ GroupRules
where
$\mathit{ReadyToReceive}(m) = \mathit{Accepting}(\mathit{type}(m))$
Consume$(m)$ =
  let $t = \mathit{type}(m)$ in
    if $\mathit{Accepting}(\mathit{currGroup}(t))$
      then insert $m$ into $\mathit{currGroup}(t)$
      else InitInsert$(m, \mathit{new}(\mathit{Group}(t)))$
InitInsert$(m, g)$ =
  InitializeGroup$(g)$
  insert $m$ into $g$
$\mathit{ToBeDiscarded}(m) =$ not $\mathit{Accepting}(\mathit{type}(m))$
GroupRules = {CreateGroup$(\mathit{type})$, Consolidate$(\mathit{group})$,
  CloseCurrGroup$(\mathit{type})$, CloseGroup$(\mathit{type})$}

CreateGroup$(\mathit{type})$ = if $\mathit{GroupCreationEvent}(\mathit{type})$ then
  let $g = \mathit{new}(\mathit{Group}(\mathit{type}))$ in InitializeGroup$(g)$
InitializeGroup$(g)$ =
  $\mathit{Accepting}(g) := \mathit{true}$
  $\mathit{currGroup}(\mathit{type}(g)) := g$
  $\mathit{timer}(g) := \mathit{now}$

Consolidate$(\mathit{group})$ = if $\mathit{Completed}(\mathit{group})$ then
  if $\mathit{Success}(\mathit{group})$
    then ProcessSuccess$(\mathit{group})$
    else ProcessFailure$(\mathit{group})$

CloseCurrGroup$(\mathit{type})$ =
  if $\mathit{Timeout}(\mathit{currGroup}(\mathit{type}))$ or $\mathit{Completed}(\mathit{currGroup}(\mathit{type}))$
    then $\mathit{Accepting}(\mathit{currGroup}(\mathit{type})) := \mathit{false}$
CloseGroup$(\mathit{type})$ =
  if $\mathit{ClosureEvent}(\mathit{type})$ then $\mathit{Accepting}(\mathit{type}) := \mathit{false}$
```

This formalization permits to have at each moment more than one correlation group open, namely one *currGroup* per message type. It also permits to have at each moment messages of different types to arrive simultaneously. It assumes however that per message type at each moment only one message is arriving. If this assumption cannot be guaranteed, one has to refine the Consume machine to consider completing a group by say m_1 of m simultaneously arriving messages and to create a new group for the remaining $m - m_1$ ones (unless the completion of a group triggers the closure of the group type).

2.3 One-to-many Send/Receive Pattern

This pattern is about sending a message to multiple recipients from where responses are expected within a given timeframe. Some parties may not respond at all or may not respond in time. The pattern can be composed out of the machine OneToManySend and the module OneFromManyReceive, similarly to the composition of the SendReceive modules out of Send and Receive. For this purpose the sending machine used in OneToManySend is assumed to contain the SetWaitCondition submachine to initialize $sendTime(m) := now$. This value is needed to determine the *Accepting* predicate in the module OneFromManyReceive to reflect that "responses are expected within a given timeframe". Remember that the refinement of the Receive-guard $Arriving(m)$ guarantees that OneToManySend has been called for $requestMsg(m)$ before OneFromManyReceive is called to Receive(m).

```
MODULE OneToManySendReceive$_{s,t}$ =
  OneToManySend$_s$ $\cup$ OneFromManyReceive$_t$
    where
      $Arriving(m) = Arrived(m)$ and $m \in ResponseMsg$
```

An instance of this multilateral service interaction pattern appears in the web service mediator model defined in [1], realized by the machine pair FeedSendReq and ReceiveAnsw. The former is an instance of OneToManySend, the latter is used as a OneFromManyReceive until all expected answers have been received.

2.4 One-from-many Receive/Send Pattern

This pattern is symmetric to OneToManySendReceive and can be similarly composed out of OneToManySend and OneFromManyReceive but with a different refinement, namely of the *SendMode* predicate, which guarantees that in any round, sent messages are responses to completed groups of received requests. Since several received messages are correlated into a single response message, which is then sent to multiple recipients, *responseMsg* is defined not on received messages, but on correlation groups of such, formed by OneFromManyReceive.

```
MODULE OneFromManyReceiveSend$_{t,s}$ =
  OneFromManyReceive$_t$ $\cup$ OneToManySend$_s$
    where $SendMode(m)$ =
      $SendMode(m)_s$ and $m = responseMsg(g)$ for some $g \in Group$ with $Completed(g)$
```

This pattern generalizes the abstract communication model for distributed systems proposed in [16] as a description of how communicators route messages through a network, namely by forwarding into the mailboxes of the *Recipients* (read: via OneToManySend) the messages found in the communicator's mailbox (read: via OneFromManyReceive). The core of this communicator model is the ASM defined in [16, Sect.3.4], which exhibits "the common part of all message-based communication networks" and is reported to have been applied to model several distributed communication architectures.

3 Composition of basic interaction patterns

In this section we illustrate how to build complex business process interaction patterns, both mono-agent (bilateral and multilateral) and asynchronous multi-agent patterns, from the eight basic interaction pattern ASMs defined in the preceding sections. There are two ways to define such patterns: one approach focusses on refinements of the interaction rules to tailor them to the needs of particular interaction steps; the other approach investigates the order and timing of single interaction steps in (typically longer lasting) runs of interacting agents.

To illustrate the possibilities for refinements of basic interaction pattern ASMs, which exploit the power of the general ASM refinement notion [5], we define two bilateral mono-agent patterns, namely an instance CompetingReceive of Receive and a refinement MultiResponse of the bilateral SendReceive. We define TransactionalMulticastNotification, a mono-agent multilateral instance of the multilateral OneToManySendReceive and a generalization of the well-known Master-Slave network protocol. We define MultiRoundOneToManySendReceive, an iterated version of OneToManySendReceive. As examples for asynchronous multi-agent patterns we define four patterns: Request with Referral, Request with Referral and Notification, Relayed Request, Dynamic Routing.

The list can be extended as needed to include any complex or specialized monoagent or multi-agent interaction pattern, by defining combinations of refinements of the eight basic bilateral and multilateral interaction pattern ASMs identified in the preceding sections.

At the end of this section we shortly discuss the investigation of interaction pattern ASM runs and link the management of such runs to the study of current thread handling disciplines.

3.1 Competing Receive Pattern

This pattern describes a racing between incoming messages of various types, where exactly one among multiple received messages will be chosen for a special Continuation of the underlying process. The normal pattern action is guarded by *waitingForResponse*s contained in expected messages of different types, belonging to a set *Type*; otherwise an abstract submachine will be called to ProcessLateResponses. The Continuation submachine is called for only one *ReceivedResponse*(*Type*), i.e. one *resp*onse of some type $t \in \mathit{Type}$, and is executed in parallel with another submachine to ProcessRemainingResponses. The interaction is closed by updating *waitingForResponse* to *false*. The choice among the competing types of received response events is expressed by a possibly dynamic and here not furthermore specified *select*ion function to select one received *resp*onse of some type. An abstract machine EscalationProcedure is foreseen in case of a *Timeout* (which appears here as a monitored predicate). It is natural to assume that EscalationProcedure changes *waitingForResponse*(*Type*) from *true* to *false*. Apparently no buffering is foreseen in this pattern, so that *ToBeDiscarded*(m) is defined as negation of *ReadyToReceive*(m).

An ASM with this behavior can be defined as a refinement of Receive(m) as follows. We define *ReadyToReceive*(m) to mean that up to now *waitingForResponse*(*Type*) holds and no *Timeout* occurred. This notion of *ReadyToReceive*(m) describes a guard for executing Receive which does not depend on m. In fact we have to make CompetingReceive work independently of whether a message arrived or not, in particular upon a *Timeout*. Therefore also *Arriving*(m) has to be adapted not to depend on m, e.g. by defining it to be always true.[14] The submachine Consume is refined to formalize the normal pattern action described above (which may be parameterized by m or not), whereas *ToBeDiscarded*(m) describes that either a *Timeout* happened or the system is not *waitingForResponse*(*Type*) any more, in which case Discard formalizes the abnormal pattern behavior invoking EscalationProcedure or ProcessLateResponses.

```
CompetingReceive = Receive
where
```

[14] These stipulations mean that the scheduler will call CompetingReceive independently of any message parameter, which is a consequence of the above stated requirements.

```
$Arriving(m) = true$
$ReadyToReceive(m) =$
  $waitingForResponse(Type)$ **and not** $Timeout$
Consume $=$
  **let** $ReceivedResponse(Type) = \{r \mid Received(r)$ **and** $Response(r, t)$ **forsome** $t \in Type\}$
    **if** $ReceivedResponse(Type) \neq \emptyset$ **then**
      **let** $resp = select(ReceivedResponse(Type))$
        Continuation$(resp)$
        ProcessRemainingResp$(ReceivedResponse(Type) \setminus \{resp\})$
      $waitingForResponse(Type) := false$
$ToBeDiscarded(m) = Timeout$ **or not** $waitingForResponse(Type)$
Discard $=$
  **if not** $waitingForResponse(Type)$ **then** ProcessLateResponses
  **if** $Timeout$ **then** EscalationProcedure
```

3.2 Contingent Request

This pattern has multiple interpretations. It can be defined as an instance of SendReceive, where the Send(m) comes with the ReSend submachine to resend m (maybe to a new recipient) if no response is received from the previous recipient within a given timeframe. I can also be defined as a combination of CompetingReceive with ReSend. In both cases the function *newVersion* reflects that the recipient of that version may be different from the recipient of the original.

3.3 Multi-response Pattern

This pattern is a multi-transmission instance of the SendReceive pattern, where the requester may receive multiple responses from the recipient "until no further responses are required". It suffices to refine the Receive-guard *readyToReceive* according to the requirements for what may cause that no further responses r will be accepted (and presumably discarded) for the request m, namely (a response) triggering to reset $FurtherResponseExpected(m)$ to *false* or a $Timeout(m)$ due to the expiry of either the request $deadline(m)$ (time elapsed since the $sendTime(m)$, set in SetWaitCondition(m) when the request m was sent) or a $lastResponseDeadline(m)$ (time that elapsed since the last response to request message m has been received).

To make this work, SetWaitCondition(m) has to be refined by adding the initialization rule $FurtherResponseExpected(m) := true$. To Receive response messages m, Consume(m) has to be refined by adding the rule $lastResponseTime(requestMsg(m)) := now$, so that the timeout predicate $Expired(lastResponseDeadline(m))$ can be defined with the help of $lastResponseTime(m)$. For the refinement of SetWaitCondition and Consume we use the notation M **addRule** R from [6] to denote the parallel composition of M and R.

```
$MODULE$ MultiResponse$_{s,t}$ $=$ SendReceive$_{s,t}$
**where**
  SetWaitCondition$(m) =$ SetWaitCondition$_{\textsc{SendReceive}_{s,t}}(m)$
    **addRule** $FurtherResponseExpected(m) := true$
  $ReadyToReceive(m) = FurtherResponseExpected(requestMsg(m))$ **and**
    **not** $Timeout(requestMsg(m))$
  Consume$(m) =$ Consume$_{\textsc{SendReceive}_{s,t}}(m)$
    **addRule** $lastResponseTime(requestMsg(m)) := now$
  $ToBeDiscarded(m) =$ **not** $ReadyToReceive(m)$
  $Timeout(m) = Expired(deadline(m))$ **or** $Expired(lastResponseDeadline(m))$
```

3.4 Transactional Multicast Notification

This pattern generalizes the well-known Master-Slave network protocol investigated in [19,12]. In each round a notification m is sent to each recipient in a possibly dynamic set $\mathit{Recipient}(m)$. The elements of $\mathit{Recipient}(m)$ are arranged in groups, allowing in groups also further groups as members, yielding a possibly arbitrary nesting of groups. Within each group g a certain number of members, typically between a minimum $\mathit{acceptMin}(m, g)$ and a maximum $\mathit{acceptMax}(m, g)$ number, are expected to "accept" the request m within a certain timeframe $\mathit{Timeout}(m)$. Recipients "accept" m by sending back an $\mathit{AcceptMsg}$ to the master.

The pattern description given below defines the master program in the master-slave protocol, whereas in applications it is typically assumed that each slave uses a blocking $\textsc{Send}$ machine. The master ASM can be defined as a refinement of the $\textsc{OneToManySendReceive}$ pattern with blocking $\textsc{Send}$. $\mathit{WaitingFor}(m)$ is refined to **not** $\mathit{Timeout}(m)$, so that the blocking condition $\mathit{status} = \mathit{blocked}(m)$ appears as waiting mode for receiving $\mathit{AcceptMsg}$es from $\mathit{Recipient}$s in response to the notification m. The nested group structure is represented as a $\mathit{recipientTree}(m)$ whose nodes (except the root) stand for groups or recipients. The set $\mathit{Leaves}(\mathit{recipientTree}(m))$ of its leaves defines the set $\mathit{Recipient}(m)$; for each tree node n which is not a leaf the set $\mathit{children}(n)$ represents a group. Since for each inner node there is only one such group, we keep every corresponding $\mathit{currGroup}(t)$ open by defining it as $\mathit{Accepting}$. We can define $\mathit{type}(r) = r$ so that $\mathit{currGroup}(r)$ for any response message r collects all the $\mathit{AcceptMsg}$s which are received from brothers of $\mathit{sender}(r)$.

$\textsc{SetWaitCondition}(m)$ is extended by a machine to $\textsc{InitializeMinMax}(m)$ and a machine to $\textsc{InitializeCurrGroup}(m)$ for each group. The Acceptance notion for tree nodes has to be computed by the master itself, namely as part of $\textsc{PerformAction}(m)$. We abstract from the underlying tree walk algorithm by defining Accept as a derived predicate, namely by recursion on the $\mathit{recipientTree}(m)$ as follows, starting from the $\mathit{AcceptMsg}$ concept of response messages accepting the notification m. By $\mid X \mid$ we denote the cardinality of the set X.

$$\begin{array}{l}
\mathit{Accept}(n) \Leftrightarrow \mathit{acceptMin}(m, \mathit{children}(n)) \leq \mid \{c \in \mathit{children}(n) \mid \mathit{Accept}(c)\} \mid \\
\mathit{Accept}(\mathit{leaf}) \Leftrightarrow \text{ some } r \in \mathit{ResponseMsg}(m) \text{ with } \mathit{AcceptMsg}(r) \text{ was received from } \mathit{leaf}
\end{array}$$

It may happen that at $\mathit{Timeout}(m)$ more than $\mathit{acceptMax}(m, g)$ accepting messages did arrive. Then a "priority" function $\mathit{chooseAccChildren}$ is used to choose an appropriate set of accepting children among the elements of $g = \mathit{children}(n)$. The elements of all these chosen sets constitute the set $\mathit{chosenAccParty}(\mathit{root})$ of recipients chosen among those who accepted the notification m. Both functions are defined as derived functions by a recursion on the $\mathit{recipientTree}(m)$ as follows:

$$\begin{array}{l}
\mathit{chooseAccChildren}(n) = \\
\quad \begin{cases} \emptyset & \textbf{if} \mid \mathit{AcceptChildren}(n) \mid < \mathit{acceptMin}(m, \mathit{children}(n)) \\ \subseteq_{\mathit{min},\mathit{max}} \mathit{AcceptChildren}(n) & \textbf{else} \end{cases} \\
\textbf{where} \\
\quad \mathit{AcceptChildren}(n) = \{c \in \mathit{children}(n) \mid \mathit{Accept}(n)\} \\
\quad \mathit{min} = \mathit{acceptMin}(m, \mathit{children}(n)) \\
\quad \mathit{max} = \mathit{acceptMax}(m, \mathit{children}(n)) \\
\quad A \subseteq_{l,h} B \Leftrightarrow A \subseteq B \textbf{ and } l \leq \mid A \mid \leq h
\end{array}$$

$$\begin{array}{l}
\mathit{chosenAccParty}(\mathit{leaf}) = \begin{cases} \{n\} & \textbf{if } \mathit{Accept}(n) \\ \emptyset & \textbf{else} \end{cases} \\
\mathit{chosenAccParty}(n) = \bigcup_{c \in \mathit{chooseAccChildren}(n)} \mathit{chosenAccParty}(c)
\end{array}$$

The submachine $\textsc{PerformAction}(m)$, which in case that $\mathit{Accept}(\mathit{root}(\mathit{recipientTree}(m)))$ holds is executed in $\mathit{UnblockMode}(m)$, $\textsc{Process}$es the $\mathit{fullRequest}(m)$ for the $\mathit{chosenAccParty}$ at the $\mathit{root}(\mathit{recipientTree}(m))$, taking into account also the other recipients. Otherwise a $\textsc{RejectProcess}$ is called.

To formulate the refinement of $\textsc{SetWaitcondition}$ we use again the notation M **addRule** R from [6] to denote the parallel composition of M and R. OTMSR stands as abbreviation for $\textsc{OneToManySendReceive}_{\mathit{ackBlocking},t}$.

```
$\mathit{MODULE}$ $\textsc{TransactionalMulticastNotify}_t = \textsc{OneToManySendReceive}_{ackBlocking,t}$
where [15]
$WaitingFor(m) =$ not $Timeout(m)$
$\textsc{SetWaitCondition}(m) = \textsc{SetWaitcondition}_{OTMSR}(m)$
  addRule
    $\textsc{InitializeMinMax}(m)$
    $\textsc{InitializeCurrGroup}(m)$
      where
      $\textsc{InitializeMinMax}(m) =$ forall $g = children(n) \in recipientTree(m)$
        $\textsc{Initialize}(acceptMin(m, g), acceptMax(m, g))$
      $\textsc{InitializeCurrGroup}(m) =$ forall $r \in Recipient(m)$
        $currGroup(r) := \emptyset$
$type(response) = response$
$Accepting(response) = response \in AcceptMsg$ and not $Timeout(requestMsg(response))$
$currGroup(response) = currGroup(sender(response))$
$currGroup(recipient) = brothers\&sisters(recipient) \cap \{leaf \mid Accept(leaf)\}$
$Accepting(currGroup(r)) = \mathrm{true}$
$\textsc{PerformAction}(m) =$
  if $Accept(root(recipientTree(m)))$ then
    let
    $accParty = chosenAccParty(root(recipientTree(m)))$
    $others = Leaves(recipientTree(m)) \setminus accParty$ in
      $\textsc{Process}(fullRequest(m), accParty, others)$
  else $\textsc{RejectProcess}(m)$
```

For a pictorial representation of TransactionalMulticastNotify (without arrows for the responses) see Figure 4, where the leaves are represented by circles and the groups by rectangles.

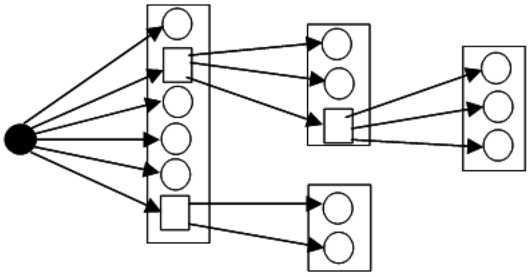


Fig. 4. Transactional Multicast Notify

3.5 Multi-round One-to-many Send/ReceivePattern

This pattern can be described as an iteration of (a refinement of) the OneToManySendReceive components OneToManySend and OneFromManyReceive. The number of one-to-many sends followed by one-from-many receives is left unspecified; to make it configurable one can introduce a corresponding *roundNumber* counter (into the *SendMode* guard and the SetWaitCondition).

The dynamic set *Recipient* guarantees that the number of parties where successive requests are sent to, and from where multiple responses to the current or any previous request may be received

[15] One could probably delete the GroupRules since they are not used by this pattern.

by the sender, may be bounded or unbounded. There is also no a priori bound on the number of previous requests, which are collected into a dynamic set *ReqHistory*.

The main refinement on sending concerns the submachine SetWaitCondition, which has to be adapted to the fact that *WaitingFor*, *sendTime* and *blocked* may depend on both the message template m and the recipient r. Furthermore this submachine has to record the message template as new *currRequest* and to store the old *currReq* into the *ReqHistory*, since incoming responses may be responses to previous request versions. The pattern description speaks about responses to "the request" for each request version, so that we use the request template m to define *currRequest* (and *Group* types below)[16]. Also the guard $SendMode(m)$ is refined to express (in addition to the possible *status* condition) that **forall** $r \in Recipient(m)$ a predicate $ReadyToSendTo(m, r)$ holds, where this predicate is intended to depend on the responses returned so far (defined below as a derived set *ResponseSoFar*).

The main refinement on receiving concerns the definition of $type(m)$ for any $m \in ResponseMsg$ as the $requestMsg(m)$ that triggered the response m. This reflects that each response message is assumed to be a response to (exactly) one of the sent requests. However, every request r is allowed to trigger more than one response m from each recipient (apparently without limit), so that the function *responseMsg* is generalized to a relation $responseMsg(m, r)$. Therefore $currGroup(request)$ represents the current collection of responses received to the *request*. It remains to reflect the condition that "the latest response ... overrides the latest status of the data ... provided, although previous states are also maintained". Since the pattern description contains no further requirements on the underlying state notion, we formulate the condition by the derived set *ResponseSoFar* defined below and by adding to Consume an abstract machine MaintainDataStatus to allow one to keep track of the *dataStatus* of previous states, for any request. *OTMSR* stands as abbreviation for OneToManySendReceive.

```
$MODULE$ MultiRoundOneToManySendReceive = OneToManySendReceive
where
  $SendMode(m) = SendMode(m)_{OTMSR}$ and
    forall $r \in Recipient(m)$ $ReadyToSendTo(m, r)$
  SetWaitCondition$(m)$ =
    forall $r \in Recipient(m)$
      Initialize$(WaitingFor(m, r))$
      $sendTime(m, r) := now$
      $status := blocked(m, r)$
    insert $currRequest$ into $ReqHistory$
    $currRequest := m$
  $type(m) = requestMsg(m)$
  Consume$(m)$ = Consume$(m)_{OTMSR}$
    addRule MaintainDataStatus$(Group(requestMsg(m)))$
```

$$ResponseSoFar = \bigcup\{Group(m) \mid m \in ReqHistory\} \cup \{currGroup(currReq)\}$$

3.6 Request With Referral

This pattern involves two agents, namely a sender of requests and a receiver from where "any follow-up response should be sent to a number of other parties ...", in particular faults, which however "could alternatively be sent to another nominated party or in fact to the sender". Apparently sending is understood without any reliability assumption, so that the sender is simply formalized by the module Send$_{noAck}$. For referring sent requests, the appropriate version of the Receive machine is used, with the submachine Consume refined to contain OneToManySend for the set $Recipient(m)$ encoded as set of *followUpResponseAddr*essees extracted from m. As stated in the requirement for the pattern, *followUpResponseAddr* may be split into disjoint subsets *failureAddr* and *normalAddr*. Since the follow-up response parties (read: $Recipient(m)$) may be

[16] Otherwise one could define $currReq(r) := msgContent(m, r)$ for each recipient r.

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 $SendMode(m)$.

```
2-Agent ASM  RequestReferral =
  Sender agent with module $\textsc{Send}_{noAck}$
  Referral agent with module Receive
    where
    $\textsc{Consume}(m) = \textsc{OneToManySend}(Recipient(m))$
    $Recipient(m) = followUpResponseAddr(m)$
```

For a pictorial representation of RequestReferral see Figure 5.

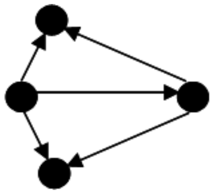


Fig. 5. Request with Referral and Advanced Notification

A refinement of RequestReferral has the additional requirement of an advanced notification, sent by the original sender to the other parties and informing them that the request will be serviced by the original receiver. This requirement comes with the further requirement that the sender may first send his request m to the receiver and only later inform the receiver (and the to-be-notified other parties) about $Recipient(m)$. These two additional requirements can be obtained by refining in RequestReferral the $\textsc{Send}_{noAck}$ 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—and the $\textsc{PerformAction}(m)$ submachine as a OneToManySend of the notification guarded by $known(Recipient(m)$.

```
2-Agent ASM  NotifiedRequestReferral =
  Sender agent with module $\textsc{Send}_{ackBlocking} \cup \{\textsc{OneToManySend}\}$
    where
      $WaitingFor(m) = \textbf{not } known(Recipient(m)) \textbf{ and not } Timeout(m)$
      $\textsc{PerformAction}(m) =$
        if not $known(Recipient(m))$ then $\textsc{SendFailure}(m)$
          else $\textsc{OneToManySend}(advancedNotif(m))$
    Referral agent with module Receive
      where
        $\textsc{Consume}(m) = \textsc{OneToManySend}(Recipient(m))$
        $Recipient(m) = followUpResponseAddr(m)$
```

3.7 Relayed Request Pattern

The RelayedRequest pattern extends Request Referral by the additional requirement that the other parties continue interacting with the original sender and that the original receiver "observes a "view" of the interactions including faults" and that the interacting parties are aware of this "view". To capture this we refine RequestReferral by equipping the sender also with a machine to Receive messages from third parties and by introducing a set *Server* of third party agents, each of with is equipped with two machines Receive and Send&Audit where the latter is a refinement of Send by the required observer mechanism.

For a pictorial representation of RelayedRequest see Figure 6.

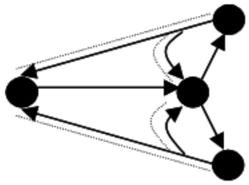


Fig. 6. Relayed Request

```
n+2-Agent ASM RelayedRequest =
  2-Agent ASM RequestReferral
    where $module(Sender)$ =
      $module_{\textsc{RequestReferral}}(Sender) \cup \{\textsc{Receive}\}$
  n Server agents with module $\{\textsc{Receive}, \textsc{Send\&Audit}\}$
    where
    Send&Audit = $\textsc{Send}_s$ with
      BasicSend = $\textsc{BasicSend}_s \cup$
        {if $AuditCondition(m)$ then $\textsc{BasicSend}_s(filtered(m))$}
```

Using as subcomponent NotifiedRequestReferral yields NotifiedRelayedRequest.

3.8 Dynamic Routing

This pattern comes with a dynamic set of agents: a first party which "sends out requests to other parties" (an instance of OneToManySend) but with the additional requirement that "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"."

In this way the third parties become additional pattern agents to receive requests, process them and forward them to the next set of recipients. Furthermore, this set of agents is dynamic: "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."

We therefore have a first *sender* agent with module OneToManySend concerning its set *Recipient*(*sender*). We then have a dynamic set of *RoutingAgent*s which can Receive request messages m with *routingSlip*(m) and Consume requests by first Processing them and then forwarding a *furtherRequest*(m, *currState*(*router*)), which may depend not only on the received (and thereby without loss of generality of the original) request message, but also on data in the router state *currState*(*router*) after message processing. Thus also the *routingSlip*(m, *currState*(*router*)) may depend on the original request and router data. The *Recipient* set depends on the *router* agent and on the *routingSlip* information. For the intrinsically sequential behavior we make use of the **seq** operator defined for ASMs in [11] (see also [12]).

```
Multi-Agent ASM DynamicRouting =
  Agent $sender$ with module $\textsc{OneToManySend}(Recipient(sender))$
  Agents $router \in RouterAgent$ each with module Receive
  where
  $\textsc{Consume}(m)$ =
    $\textsc{Process}(m)$ seq $\textsc{OneToManySend}(furtherRequest(m, currState(router)))$
      $(Recipient(router, routingSlip(m, currState(router))))$
```

3.9 Defining Interaction Flows (Conversations)

In the preceding section the focus for the composition of basic interaction patterns into more complex ones was on combination and refinement of basic interaction pattern ASMs, i.e. on how the interaction rules (read: the programs) executed by the communicating agents can be adapted to the patterns under consideration. An equally important different view of interaction patterns is focussed instead on the run scenarios, that is to say on when and in which order the participating agents perform interaction steps by applying their interaction rules. This view is particularly important for the study of interaction structures which occur in long running processes, whose collaborations involve complex combinations of basic bilateral or multilateral interaction pattern ASM moves.

As an example for an interaction structure in a long running process consider a one-to-many-send-receive to short-list candidate service providers, which may be followed by a transactional multi-cast to issue service requests to selected providers, where finally individual providers may use relayed requests for outsourcing the work.

An elementary example is the well-known coroutining pattern, which is characterized by two agents a_1 and a_2 each equipped with a $\textsc{Send}_{noAck}$ and a $\textsc{Receive}$ module. The typical scenario is a distributed run where an application of $\textsc{Send}_{noAck}$ by a_1 precedes firing $\textsc{Receive}$ by a_2, which (as consequence of the execution of $\textsc{Consume}$ at a_2) is followed by an application of $\textsc{Send}_{noAck}$ by a_2 and eventually triggers an execution of $\textsc{Receive}$ at a_1.

Such interactions structures resemble *conversations* or *interaction flows* between collaborating parties. This concept is captured by the notion of asynchronous runs of multi-agent service interaction pattern ASMs, i.e. ASMs whose rules consist of some of the basic or composed service interaction pattern ASMs defined in the preceding sections. Such runs, also called distributed runs, are partial orders of moves of the participating agents, each of which is (read: executes) a sequential ASM, constrained by a natural condition which guarantees that independent moves can be put into an arbitrary execution order without changing the semantical effect. [17] We therefore define a *conversation* or *interaction flow* to be a run of an asynchronous service interaction pattern ASM, where such an ASM is formally defined by a set of agents each of which is equipped with some (basic or complex, bilateral or multilateral) service interaction pattern modules defined in the previous sections.

A theory of such interaction flow patterns is needed, which builds upon the knowledge of classical workflow analysis [25]. A satisfactory theory should also provide possibilities to study the effect of allowing some agents to $\textsc{Start}$ or $\textsc{Suspend}$ or $\textsc{Resume}$ or $\textsc{Stop}$ such collaborations (or parts of them), the effect such conversation management actions have for example on security policies, etc. This naturally leads to investigate the impact of current thread handling methods (see for example [26,23,22]) on business process interaction management.

4 Conclusion and Outlook

We would like to see the ASM models provided here (or modified versions thereof) be implemented in a provably correct way, e.g. by BPEL programs, and be used as benchmarks for existing implementations. We would also like to see other interaction patterns be defined as combinations of refinements of the eight basic bilateral and multilateral service interaction pattern ASMs defined here. In particular we suggest the study of conversation patterns (business process interaction flows), viewed as runs of asynchronous multi-agent interaction pattern ASMs.

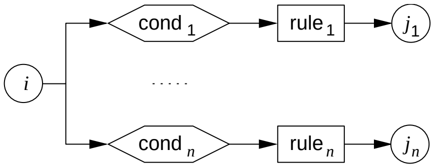
5 Appendix: The Ingredients of the ASM Method

The ASM method for high-level system design and analysis (see the AsmBook [12]) comes with a simple mathematical foundation for its three constituents: the notion of *ASM*, the concept of *ASM ground model* and the notion of *ASM refinement*. For an understanding of this paper only

[17] Details on this definition of partial order ASM run can be found in [12, pg.208].

the concept of ASM and that of ASM refinement have to be grasped,[18] whose definitions support the intuitive understanding of the involved concepts. We use here the definitions first presented in [7,3] and [5].

5.1 ASMs = FSMs with arbitrary locations



```
if $ctl\_state = i$ then
  if $cond_1$ then
    $rule_1$
    $ctl\_state := j_1$
      ...
  if $cond_n$ then
    $rule_n$
    $ctl\_state := j_n$
```

Fig. 7. Viewing FSM instructions as control state ASM rules

The instructions of an FSM program are pictorially depicted in Fig. 7, where $i, j_1, \ldots, j_n$ are internal (control) states, $cond_\nu$ (for $1 \leq \nu \leq n$) represents the input condition $in = a_\nu$ (reading input a_ν) and $rule_\nu$ the output action $out := b_\nu$ (yielding output b_ν), which goes together with the ctl_state update to j_ν. Control state ASMs have the same form of programs and the same notion of run, but the underlying notion of state is extended from the following three locations:

- a single internal *ctl_state* that assumes values in a not furthermore structured finite set
- two input and output locations *in*, *out* that assume values in a finite alphabet

to a *set of possibly parameterized locations holding values of whatever types*. Any desired level of abstraction can be achieved by permitting to hold values of arbitrary complexity, whether atomic or structured: objects, sets, lists, tables, trees, graphs, whatever comes natural at the considered level of abstraction. As a consequence an FSM step, consisting of the simultaneous update of the *ctl_state* and of the *out*put location, is turned into an ASM step consisting of the simultaneous update of a set of locations, namely via multiple assignments of the form $loc(x_1, \ldots, x_n) := val$, yielding a new ASM state.

This simple change of view of what a state is yields machines whose states can be arbitrary *multisorted structures*, i.e. domains of whatever objects coming with predicates (attributes) and functions defined on them, structures programmers nowadays are used to from object-oriented programming. In fact such a memory structure is easily obtained from the flat location view of abstract machine memory by grouping subsets of data into tables (arrays), via an association of a value to each table entry $(f, (a_1, \ldots, a_n))$. Here f plays the role of the name of the table, the sequence $(a_1, \ldots, a_n)$ the role of a table entry, $f(a_1, \ldots, a_n)$ denotes the value currently contained in the location $(f, (a_1, \ldots, a_n))$. Such a table represents an array variable f of dimension n, which can be viewed as the current interpretation of an n-ary "dynamic" function or predicate (boolean-valued function). This allows one to structure an ASM state as a set of tables and thus as a multisorted structure in the sense of mathematics.

In accordance with the extension of unstructured FSM control states to ASM states representing arbitrarily rich structures, the FSM-input *cond*ition is extended to arbitrary ASM-state expressions, namely formulae in the signature of the ASM states. They are called *guards* since they determine whether the updates they are guarding are executed.[19] In addition, the usual nondeterministic interpretation, in case more than one FSM-instruction can be executed, is replaced by the parallel interpretation that in each ASM state, the machine executes simultaneously all the

[18] For the concept of ASM ground model (read: mathematical system blueprint) see [4].

[19] For the special role of *in*/*out*put locations see below the classification of locations.

updates which are guarded by a condition that is true in this state. This *synchronous parallelism*, which yields a clear concept of *locally described global state change*, helps to abstract for high-level modeling from irrelevant sequentiality (read: an ordering of actions that are independent of each other in the intended design) and supports refinements to parallel or distributed implementations.

Including in Fig. 7 $ctl\_state = i$ into the guard and $ctl\_state := j$ into the multiple assignments of the rules, we obtain the definition of a *basic ASM* as a set of instructions of the following form, called ASM *rules* to stress the distinction between the parallel execution model for basic ASMs and the sequential single-instruction-execution model for traditional programs:

```
if $cond$ then $Updates$
```

where *Updates* stands for a set of *function updates* $f(t_1, \ldots, f_n) := t$ built from expressions t_i, t and an n-ary function symbol f. The notion of run is the same as for FSMs and for transition systems in general, taking into account the synchronous parallel interpretation.[20] Extending the notion of mono-agent sequential runs to asynchronous (also called partially ordered) multi-agent runs turns FSMs into globally asynchronous, locally synchronous Codesign-FSMs [18] and similarly basic ASMs into *asynchronous ASMs* (see [12, Ch.6.1] for a detailed definition).

The synchronous parallelism (over a finite number of rules each with a finite number of to-be-updated locations of basic ASMs) is often further extended by a synchronization over arbitrary many objects in a given *Set*, which satisfy a certain (possibly runtime) *Property*:

```
forall $x \in Set$ with $Property(x)$ do
  $rule(x)$
```

standing for the execution of *rule* for every object x, which is element of *Set* and satisfies *Property*. Sometimes we omit the key word **do**. The parts $\in Set$ and **with** $Property(x)$ are optional.

ASM Modules Standard module concepts can be adopted to syntactically structure large ASMs, where the module interface for the communication with other modules names the ASMs which are imported from other modules or exported to other modules. We limit ourselves here to consider an ASM module as a pair consisting of *Header* and *Body*. A module header consists of the name of the module, its (possibly empty) import and export clauses, and its signature. As explained above, the signature of a module determines its notion of state and thus contains all the basic functions occurring in the module and all the functions which appear in the parameters of any of the imported modules. The body of an ASM module consists of declarations (definitions) of functions and rules. An ASM is then a module together with an optional characterization of the class of initial states and with a compulsory additional (the main) rule. Executing an ASM means executing its main rule. When the context is clear enough to avoid any confusion, we sometimes speak of an ASM when what is really meant is an ASM module, a collection of named rules, without a main rule.

ASM Classification of Locations and Functions The ASM method imposes no a priori restriction neither on the abstraction level nor on the complexity nor on the means of definition of the functions used to compute the arguments and the new value denoted by t_i, t in function updates. In support of the principles of separation of concerns, information hiding, data abstraction, modularization and stepwise refinement, the ASM method exploits, however, the following distinctions reflecting the different roles these functions (and more generally locations) can assume in a given machine, as illustrated by Figure 8 and extending the different roles of *in*, *out*, *ctl_state* in FSMs.

[20] More precisely: to execute one step of an ASM in a given state S determine all the fireable rules in S (s.t. *cond* is true in S), compute all expressions t_i, t in S occuring in the updates $f(t_1, \ldots, t_n) := t$ of those rules and then perform simultaneously all these location updates if they are consistent. In the case of inconsistency, the run is considered as interrupted if no other stipulation is made, like calling an exception handling procedure or choosing a compatible update set.

A function f is classified as being of a given type if in every state, every location $(f, (a_1, \ldots, a_n))$ consisting of the function name f and an argument $(a_1, \ldots, a_n)$ is of this type, for every argument $(a_1, \ldots, a_n)$ the function f can take in this state.

Semantically speaking, the major distinction is between static and dynamic locations. Static locations are locations whose values do not depend on the dynamics of states and can be determined by any form of satisfactory state-independent (e.g. equational or axiomatic) definitions. The further classification of dynamic locations with respect to a given machine M supports to distinguish between the roles different 'agents' (e.g. the system and its environment) play in using (providing or updating the values of) dynamic locations. It is defined as follows:

- *controlled* locations are readable and writable by M,
- *monitored* locations are for M only readable, but they may be writable by some other machine,
- *output* locations are by M only writable, but they may be readable by some other machine,
- *shared* locations are readable/writable by M as well as by some other machine, so that a protocol will be needed to guarantee the consistency of writing.

Monitored and shared locations represent an abstract mechanism to specify communication types between different agents, each executing a basic ASM. *Derived* locations are those whose definition in terms of locations declared as basic is fixed and may be given separately, e.g. in some other part ("module" or "class") of the system to be built. The distinction of derived from basic locations implies that a derived location can in particular not be updated by any rule of the considered machine. It represents the input-output behavior performed by an independent computation. For details see the AsmBook [12, Ch.2.2.3] from where Figure 8 is taken.

A particularly important class of monitored locations are selection locations, which are frequently used to abstractly describe scheduling mechanisms. The following notation makes the inherent non-determinism explicit in case one does not want to commit to a particular selection scheme.

```
choose $x \in Set$ with $Property(x)$ do
  $rule(x)$
```

This stands for the ASM executing $rule(x)$ for some element x, which is arbitrarily chosen among those which are element of *Set* and satisfy the selection criterion *Property*. Sometimes we omit the key word **do**. The parts $\in$ *Set* and **with** $Property(x)$ are optional.

We freely use common notations like **let** $x = t$ **in** R, **if** *cond* **then** R **else** S, etc. When refining machines by adding new rules, we use the following notation introduced in [6]: M **addRule** R denotes the parallel composition of M and R. Similarly M **minusRule** R denotes N for $M = N, R$.

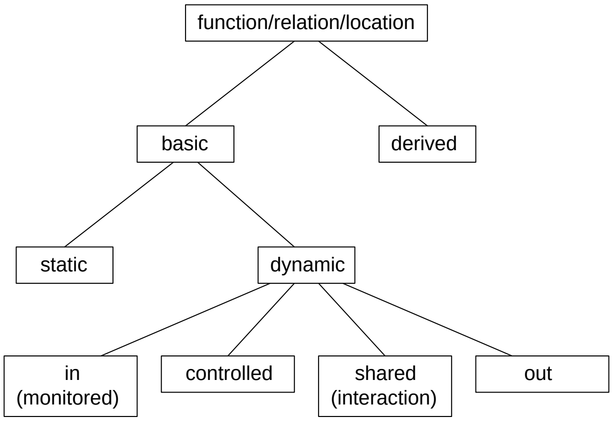


Fig. 8. Classification of ASM functions, relations, locations

To avoid confusion among different machines, which occur as submachine of machines N, N' but within those machines carry the same name M, we use indexing and write M_N respectively $M_{N'}$.

Non-determinism, Selection and Scheduling Functions It is adequate to use the **choose** construct of ASMs if one wants to leave it completely unspecified who is performing the choice and based upon which selection criterion. The only thing the semantics of this operator guarantees is that each time one element of the set of objects to choose from will be chosen. Different instances of a selection, even for the same set in the same state, may provide the same element or maybe not. If one wants to further analyze variations of the type of choices and of who is performing them, one better declares a *Select*ion function, to select an element from the underlying set of *Cand*idates, and writes instead of **choose** $c \in \mathit{Cand}$ **do** $R(c)$ as follows, where R is any ASM rule:

let $c = \mathit{Select}(\mathit{Cand})$ **in** $R(c)$

The functionality of *Select* guarantees that exactly one element is chosen. The **let** construct guarantees that the choice is fixed in the binding range of the **let**. Declaring such a function as dynamic guarantees that the selection function applied to the same set in different states may return different elements. Declaring such a function as controlled or monitored provides different ownership schemes. Naming these selection functions allows the designer in particular to analyze and play with variations of the selection mechanisms due to different interpretations of the functions.

5.2 ASM Refinement Concept

The ASM refinement concept is a generalization of the familiar commutative diagram for refinement steps, as illustrated in Figure 9.

$$\begin{tikzcd}
\text{State } S \arrow[r, "\overbrace{\tau_1 \ \cdots \ \tau_m}^{m \text{ steps of } M}"] \arrow[d, leftrightarrow, "\equiv"'] & S' \arrow[d, leftrightarrow, "\equiv"] \\
\text{State } S^* \arrow[r, "\underbrace{\sigma_1 \ \cdots \ \sigma_n}_{n \text{ steps of } M^*}"'] & S^{*\prime}
\end{tikzcd}$$

$\equiv$ is an equivalence notion between data
in locations of interest in corresponding states.

Fig. 9. The ASM refinement scheme.

For an ASM refinement of an ASM M to an ASM M^*, as designer one has the freedom to tailor the following "handles" to the needs for detailing the design decision which leeds from M to M':

- a notion of *refined state*,
- a notion of *states of interest* and of *correspondence* between M-states S and M^*-states S^* of interest, i.e. the pairs of states in the runs one wants to relate through the refinement, including usually the correspondence of initial and (if there are any) of final states,

- a notion of abstract *computation segments* $\tau_1, \ldots, \tau_m$, where each τ_i represents a single M-step, and of corresponding refined computation segments $\sigma_1, \ldots, \sigma_n$, of single M^*-steps σ_j, which in given runs lead from corresponding states of interest to (usually the next) corresponding states of interest (the resulting diagrams are called (m, n)-diagrams and the refinements (m, n)-refinements),
- a notion of *locations of interest* and of *corresponding locations*, i.e. pairs of (possibly sets of) locations one wants to relate in corresponding states, where locations represent abstract containers for data,
- a notion of *equivalence* $\equiv$ of the data in the locations of interest; these local data equivalences usually accumulate to a notion of equivalence of corresponding states of interest.

The scheme shows that an ASM refinement allows one to combine in a natural way a change of the signature (through the definition of states and of their correspondence, of corresponding locations and of the equivalence of data) with a change of the control (defining the "flow of operations" appearing in the corresponding computation segments).

Once the notions of corresponding states and of their equivalence have been determined, one can define that M^* is a correct refinement of M if and only if every (infinite) refined run simulates an (infinite) abstract run with equivalent corresponding states, as is made precise by the following definition. By this definition, refinement correctness implies for the special case of terminating runs the inclusion of the input/output behavior of the abstract and the refined machine.

Definition Fix any notions $\equiv$ of equivalence of states and of initial and final states. An ASM M^* is called a *correct refinement* of an ASM M if and only if for each M^*-run $S_0^*, S_1^*, \ldots$ there is an M-run $S_0, S_1, \ldots$ and sequences $i_0 < i_1 < \ldots, j_0 < j_1 < \ldots$ such that $i_0 = j_0 = 0$ and $S_{i_k} \equiv S_{j_k}^*$ for each k and either

- both runs terminate and their final states are the last pair of equivalent states, or
- both runs and both sequences $i_0 < i_1 < \ldots$, $j_0 < j_1 < \ldots$ are infinite.

Often the M^*-run $S_0^*, S_1^*, \ldots$ is said to simulate the M-run $S_0, S_1, \ldots$. The states $S_{i_k}, S_{j_k}^*$ are the corresponding states of interest. They represent the end points of the corresponding computation segments (those of interest) in Figure 9, for which the equivalence is defined in terms of a relation between their corresponding locations (those of interest). Sometimes it is convenient to assume that terminating runs are extended to infinite sequences which become constant at the final state.

M^* is called a *complete refinement* of M if and only if M is a correct refinement of M^*.

This definition of ASM refinement underlies numerous successful applications of ASMs to high-level system desing and analysis (see the survey in the history chapter in [12]) and generalizes and integrates well-known more specific notions of refinement (see [20,21] for a detailed analysis).

Acknowledgement The bulk of the work on this paper was done when the second author was on sabbatical leave at SAP Research, Karlsruhe, Germany. We thank M. Altenhofen and W. Reisig for critical comments on earlier versions of this paper.

References

1. M. Altenhofen, E. Börger, and J. Lemcke. An abstract model for process mediation. In R. Banach, editor, *Proc. 7th International Conference on Formal Engineering Methods (ICFEM 2005)*, LNCS. Springer, 2005.
2. A. Barros, M. Dumas, and A. ter Hofstede. Service interaction patterns: Towards a reference framework for service-based business process interconnection. Technical Report FIT-TR-2005-02 (To be presented at BPM'2005, Third International Conference on Business Process Management 2005, September 2005, Nancy, France), Faculty of Information Technology, Queensland University of Technology, Brisbane (Australia), March 2005.
3. E. Börger. High-level system design and analysis using Abstract State Machines. In D. Hutter, W. Stephan, P. Traverso, and M. Ullmann, editors, *Current Trends in Applied Formal Methods (FM-Trends 98)*, volume 1641 of *Lecture Notes in Computer Science*, pages 1–43. Springer-Verlag, 1999.

4. E. Börger. The ASM ground model method as a foundation of requirements engineering. In N.Dershowitz, editor, *Verification: Theory and Practice*, volume 2772 of *LNCS*, pages 145–160. Springer-Verlag, 2003.
5. E. Börger. The ASM refinement method. *Formal Aspects of Computing*, 15:237–257, 2003.
6. E. Börger. Linking architectural and component level system views by abstract state machines. In C. Grimm, editor, *Languages for System Specification and Verification*, CHDL, pages 247–269. Kluwer, 2004.
7. E. Börger. The ASM method for system design and analysis. A tutorial introduction. In B. Gramlich, editor, *FroCoS 2005*, volume 3717 of *Lecture Notes in Artificial Intelligence*, pages 264–283. Springer, 2005.
8. E. Börger. Design pattern abstractions and Abstract State Machines. In D. Beauquier, E. Börger, and A. Slissenko, editors, *Proc.ASM05*, pages 91–100. Université de Paris 12, 2005.
9. E. Börger. From finite state machines to virtual machines (Illustrating design patterns and event-B models). In E. Cohors-Fresenborg and I. Schwank, editors, *Präzisionswerkzeug Logik–Gedenkschrift zu Ehren von Dieter Rödding*. Forschungsinstitut für Mathematikdidaktik Osnabrück, 2005. ISBN 3-925386-56-4.
10. E. Börger. Linking content definition and analysis to what the compiler can verify. In *Proc.IFIP WG Conference on Verified Software: Tools, Techniques, and Experiments*, Lecture Notes in Computer Science, Zurich (Switzerland), October 2005. Springer.
11. E. Börger and J. Schmid. Composition and submachine concepts for sequential ASMs. In P. Clote and H. Schwichtenberg, editors, *Computer Science Logic (Proceedings of CSL 2000)*, volume 1862 of *Lecture Notes in Computer Science*, pages 41–60. Springer-Verlag, 2000.
12. E. Börger and R. F. Stärk. *Abstract State Machines. A Method for High-Level System Design and Analysis*. Springer, 2003.
13. R. Farahbod. Extending and refining an abstract operational semantics of the web services architecture for the business process execution language. Master's thesis, Simon Fraser University, Burnaby, Canada, July 2004.
14. R. Farahbod, U. Glässer, and M. Vajihollahi. Abstract operational semantics of the Business Process Execution Language for web services. Technical Report SFU-CMPT-TR 2004-03, Simon Fraser University School of Computing Science, April 2004.
15. R. Farahbod, U. Glässer, and M. Vajihollahi. Specification and validation of the Business Process Execution Language for web services. In W. Zimmermann and B. Thalheim, editors, *Abstract Sate Machines 2004*, volume 3052 of *Lecture Notes in Computer Science*, pages 78–94. Springer-Verlag, 2004.
16. U. Glässer, Y. Gurevich, and M. Veanes. Abstract communication model for distributed systems. *IEEE Transactions on Software Engineering*, 30(7):1–15, July 2004.
17. C. A. R. Hoare. The verifying compiler: A grand challenge for computing research. *J. ACM*, 50(1):63–69, 2003.
18. L. Lavagno, A. Sangiovanni-Vincentelli, and E. M. Sentovitch. Models of computation for system design. In E. Börger, editor, *Architecture Design and Validation Methods*, pages 243–295. Springer-Verlag, 2000.
19. W. Reisig. *Elements of Distributed Algorithms*. Springer-Verlag, 1998.
20. G. Schellhorn. Verification of ASM refinements using generalized forward simulation. *J. Universal Computer Science*, 7(11):952–979, 2001.
21. G. Schellhorn. ASM refinement and generalizations of forward simulation in data refinement: A comparison. *Theoretical Computer Science*, 336(2-3):403–436, 2005.
22. R. F. Stärk. Formal specification and verification of the C# thread model. *Theoretical Computer Science*, 2005. To appear.
23. R. F. Stärk and E. Börger. An ASM specification of C# threads and the .NET memory model. In W. Zimmermann and B. Thalheim, editors, *Abstract State Machines 2004*, volume 3052 of *Lecture Notes in Computer Science*, pages 38–60. Springer-Verlag, 2004.
24. M. Vajihollahi. High level specification and validation of the Business Process Execution Language for web services. Master's thesis, School of Computing Science at Simon Fraser University, April 2004.
25. W. M. van der Aalst, A. ter Hofstede, B. Kiepuszewski, and A. Barros. Workflow patterns. *Distributed and Parallel Databases*, 14(3):5–51, July 2003.
26. C. Wallace, G. Tremblay, and J. N. Amaral. An Abstract State Machine specification and verification of the location consistency memory model and cache protocol. *J. Universal Computer Science*, 7(11):1089–1113, 2001.