Egon Börger (Pisa) & Alexander Raschke (Ulm)

Communicating Business Processes

The Subject-Oriented BP Communication Model

A Modeling-For-Change Case Study

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

See Ch. 5.2 of Modeling Companion[1]
http://modelingbook.informatik.uni-ulm.de

[1] Except where stated otherwise, the figures are copied from the book and © 2018 Springer-Verlag Germany, reprinted with permission.

Theme: Modeling-For-Change with Abstract State Machines

- We replay specifying an industrial workflow engine, developed for the execution of Business Process Models (BPMs), whose requirements were presented piecemeal, during the modeling effort.
  - Each time additional requirements were introduced, they were captured by an ASM refinement of the previously developed model.
- The involved ASMs are instances of a pattern which occurs frequently in the field of Business Processes (BPs), namely
  - control state ASMs whose transitions may have an iterative structure that is guided by a task-completion concept, coming with entry and exit conditions and actions.

Therefore, in this lecture we

- first define this class of *ASM Net Diagrams*, tailored for modeling BPs,
- then use them to stepwise develop the *S-BPM communication model* which is characteristic for the Subject-Oriented Business Process Modeling approach.

Goal of ASM nets: tailor ASMs for the domain of BPs

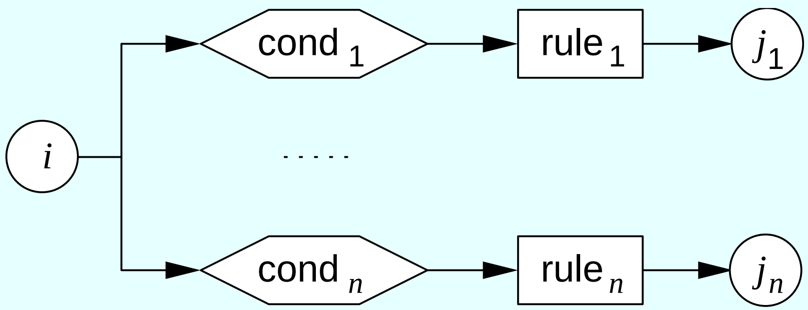
by defining a BP-specific class of ASM Nets based upon which

- BP experts can *express a BP design*
  - using directly BP-knowledge-based terms/notations which are supported by ASM constructs expressing their intuitive understanding
    - correctly: controllable by BP experts via inspection/validation
    - precisely: for the sw expert as spec for the implementation
  - combining control, data, communication and resource aspects
- system designers can *implement the experts' BP design*
  - using behavior preserving ASM refinements to executable code
    - documented (for explanation) and justified (for correctness)

such that the BP developer can work with the underlying intuitive understanding of the graphical constructs, supported by a mathematically precise, easy to check definition ('formalization') of the behavioral interpretation of the graphical notations

Technical Idea: Build Iteration into Control State ASMs

- Control state ASM have all their rules of the form:[2]

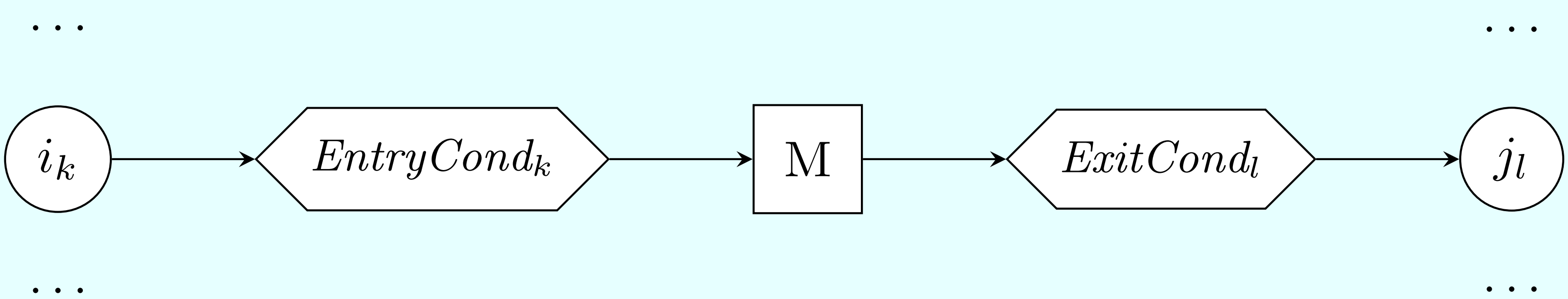


- Enrich control-state ASMs by an iterated execution view of components $rule_k$ that occurs frequently with BPMs and is guided by
  – a ('milestone') completion concept
  – entry ('guard') and exit ('termination') conditions/actions

[2] Figure copied from the AsmBook, © 2003 Springer-Verlag Berlin Heidelberg, reprinted with permission

Defining ASM net diagrams

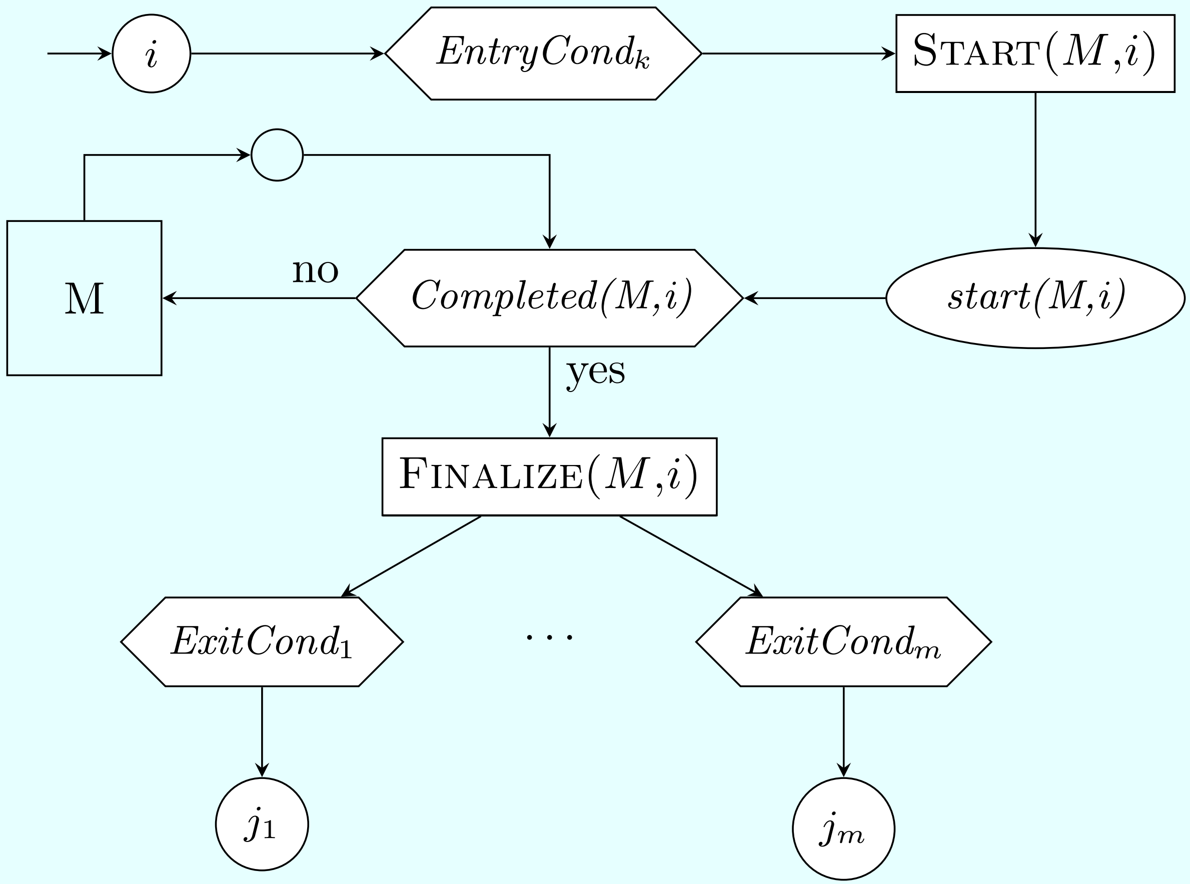
An ASM net is a directed graph built from ASM net transitions



connecting exitnodes j_l with at most one entry node i_k. Behavior:

- execution of (an instance of) body M begins with executing $\textsc{Start}$ and passing control to M if $mode = i_k$ for some entry node i_k and $EntryCond_k$ is satisfied.
- M is executed as long as the body is $active$ and not yet $Completed$
- $\textsc{Finalize}$ is executed when M's execution is $Completed$, letting the net computation proceed to $mode = j_l$ for the exit node j_l whose $ExitCond_l$ is satisfied.

Flowchart visualizing ASM net rule behavior (let $i = i_k$)



Often used with one entry (true guard), two exits (+,-)

Textual definition of $\textsc{AsmRule}(\textit{transition}, i)$

let $i = i_k$

if $mode = i$ **and** $EntryCond_k$ **then** -- enter M

$\quad\textsc{Start}(M, i)$

$\quad mode := start(M, i)$ -- make M $active$

if $active(M, i)$ **then**

$\quad$**if not** $Completed(M, i)$ **then** M -- iterate M

$\qquad$**else**

$\qquad\quad\textsc{Finalize}(M, i)$

$\qquad\quad\textsc{Exit}$

where

$\quad\textsc{Exit} =$ **forall** $j \in \{j_1, \ldots, j_m\}$ **if** $ExitCond_j$ **then** $mode := j$

$\quad active(M, i)$ iff $mode_{M,i} \in Mode(M, i)$

$\quad Mode(M, i) =$ set of $mode_{M,i}$ values of M_i (instance of M)

Synchronous/Asynchronous net interpretation

Sync: *same* $mode$ *location used in all net transitions and bodies*

$\textsc{SyncAsmNet}(N) =$

forall $transition \in N$ $\textsc{AsmRule}_{sync}(transition)$

where

$\textsc{AsmRule}_{sync}(transition) =$

forall $i \in \{i_1, \ldots, i_n\}$ $\textsc{AsmRule}(transition, i)$

Concur: *each* $transition$ *has its own* $mode$ *location*.

$\textsc{ConcurAsmNet}(N)$: a family of

- agents $a_{transition}$ with program $\textsc{AsmRule}_{sync}(transition)$ or
- agents $a_{(transition,i)}$ with program $\textsc{AsmRule}(transition, i)$

ASM nets rigorously capture UML activity dgms

- UML event driven activity diagram scheme:
  - *If a certain event or state configuration (situation) occurs, perform an action and proceed* along the indicated control flow
- control-state ASMs provide a general, mathematically rigorous, abstract meaning of:
  - *situation* = configuration of whatever items/signals/data/resources (ASM state), expressed by rule guarding $cond$itions
  - *action* = change of the configuration (value) of some items (state transition/update), expressed by ASM transition rules
  - *proceed* = update ctl_state (change mode)
- NB. Each (synchronous) UML activity diagram can be built from alternating branching and action nodes of ASM net transitions, for each agent

See the ASM-based platform built at U of Ulm for design and exec of rigorous UML diagrams (Saarstedt, Guttmann, Raschke et al.)

Industrial Case Study: S-BPM communication model

Goal: develop an ASM net to specify an interpreter for S-PBM which directly and faithfully reflects how the S-BPM engine executes:

- process agents (called *subjects*) which
- perform 2 kinds of *actions* on arbitrary *objects* (data type operations)
  - $internal$ process actions (in the agent's local state)
  - $external$ process actions: communication with other agents
    - thus the processes can be modeled as communicating ASMs

walking through the $node$s of an associated *Subject Behavior Diagram* SBD with associated actions $service(node)$.

Each Subject Behavior Diagram modeled as ASM net

The S-BPM semantics of SBDs is characterized by the following *action execution requirement*:

- once an agent entered an SBD-node (read: the current process $stage$),
- it enters the next SBD-node (to execute the associated next action) only upon completion of the action associated with the current $stage$,

Thus, an SBD *can be modeled directly as an ASM net*:

- with exactly one $initial$ node
- possibly multiple end nodes such that
  - each path starting at the $initial$ node leads to at least one end node
- each net transition has one $entry$ and possibly multiple $exit$ nodes

The behavioral interpretation is $\textsc{SyncAsmNet}(SBD)$ because

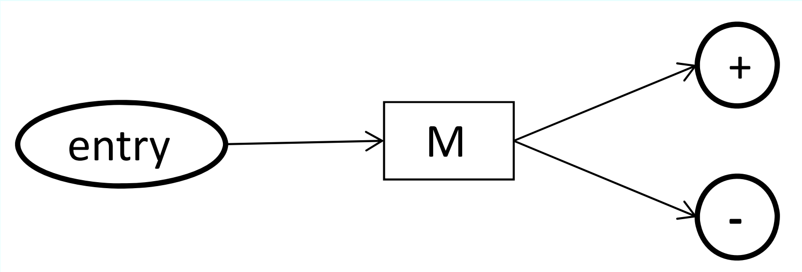
- each SBD describes one sequential process, executed by one subject

Distributed S-BPM processes modeled as concurrent ASM

Let $P = (subj_k, SBD_k)_{1 \leq k \leq l}$ be a distributed S-BPM process.

- Each participating subprocess $(subj_k, SBD_k)$ is a sequential process, modeled as $\textsc{SyncAsmNet}(SBD_k)$.
- P is a concurrent ASM, in fact a family of communicating ASMs, modeled as $\textsc{AsyncAsmNet}(P)$.

To describe single SBDs, we will use mainly ASM net transitions with one entry and two exits:



Capture piecemeal presented requs by stepwise refinements

We concentrate on the transitions for nodes with an associated communication action $ComAct$, either a Send or Receive action.

The requirements for $ComnAct$ions have been formulated in three steps:

- for communication of single msgs,
- for communication of multiple msgs,
- for communication alternatives.

At a later stage, an additional requirement was formulated

- for arbitrary alternative actions.

To illustrate how ASM models support design for change, we specify these actions in the indicated order, starting with a ground model and 3 times refining the previous model.

We define the components of the corresponding ASM net transitions.

S-BPM InputPoolSizeReq

The communication bw S-BPM processes is characterized by specific mailbox requirements. Mailboxes are called input pool.

We first list the requirements for communication of single msgs.

InputPoolSizeReq. The $inputPool$ has the following size restrictions, based upon an underlying classification of messages into types:

- overall capacity $maxSize \in \{0, 1, \ldots\}$ (non-negative number),
- maximal number $maxFrom(sender)$ of messages allowed from a $sender$ or $maxFor(type)$ of a $type$.

At configuration time the user can indicate for any $sender$ and message $type$ the desired $size$ limit and the $sizeAction$ to be taken in case of a $size$ violation.

S-BPM SizeViolationActionReq and SyncReceiveReq

SizeViolationActionReq. If a message m violates a $size$ restriction, one of the following $sizeAction$s can be taken:

- m is dropped (not inserted into the $inputPool$),
- m is blocked (not inserted into the $inputPool$) but can be tried to be sent synchronously; $maxSize$ violation (for $maxSize < \infty$) implies $action = Blocking$,
- either the $oldestMsg$ or the $youngestMsg$, determined in terms of its $insertionTime$ into the $nputPool$), is deleted from the $inputPool$ and m is inserted.

SyncReceiveReq. $maxFrom(sender) = 0$ and/or $maxFor(type) = 0$ indicates that the owner of the $inputPool$ accepts messages from the indicated $sender$ and/or of the indicated $type$ only via a rendezvous (synchronously). $size = 0$ implies $action = Blocking$. Positive size limits are used for asynchronous communication.

MsgPreparationReq for communication of single messages

MsgPreparationReq. A communication action starts with defining the $curMsg$ to be handled: for Send this is a concrete message with its data; for Receive it is the kind of message to look for in the $inputPool$, namely either any message or a message from a particular $sender$ or a message of a particular $type$ or a message of a particular type from a particular sender.

This requirement is satisfied by defing the $\textsc{Start}$ component of the $\textsc{SingleSendNet}$ transition as follows:

$$\textsc{Start}(\textsc{SingleSend}, entry) = \textsc{Prepare}(curMsg, entry, Send)$$

$$\textsc{Prepare}(curMsg, entry, Send) =$$
$$\quad curMsg := composeMsg(msgData(entry))$$

NB. The functions $msgData, composeMsg$ can be varied by different implementations.

SendActionReq for single messages

SendActionReq. If a Send action $CanAccess$ the $inputPool$ of the receiver, it inserts its message m asynchronously into the $inputPool$ if this implies no $SizeViolation$.

If $SizeViolation(m) = true$ there are three possible cases:

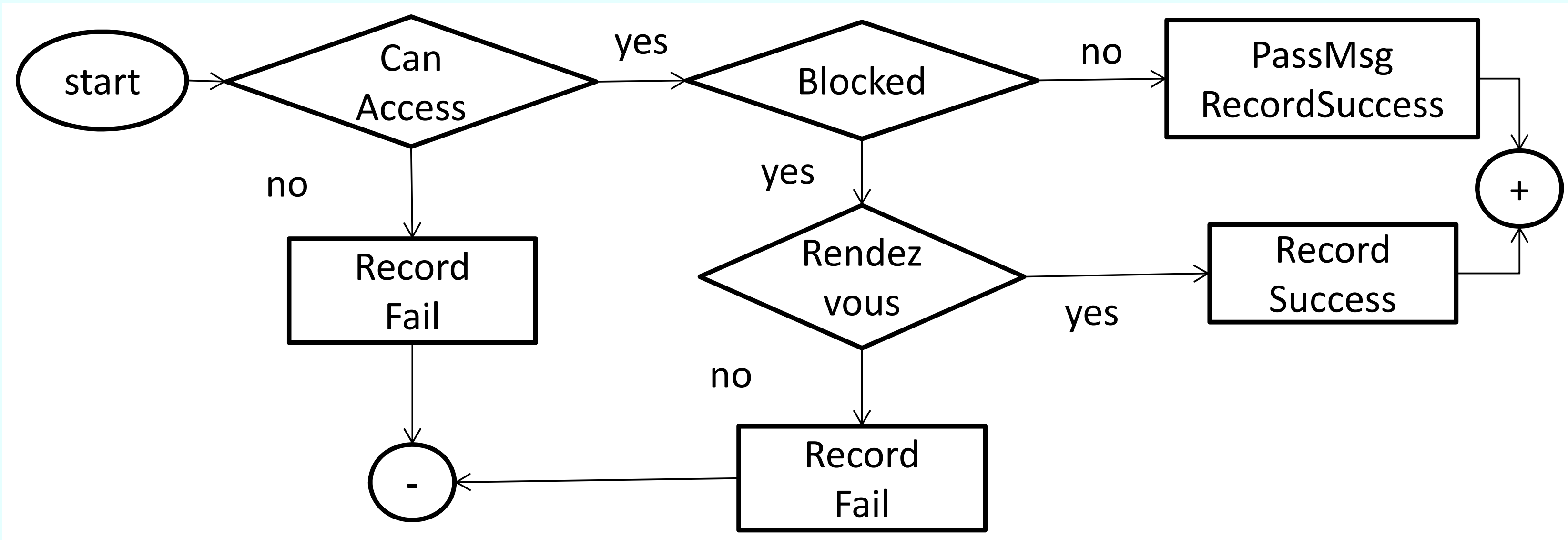
$CancelingSend$ case: m is inserted if the corresponding $sizeAction(m)$ is to drop the youngest or the oldest message from the $inputPool$

$DropIncoming$ case: m is simply not inserted

$Blocking$ case: m is not inserted but an attempt is made to synchronously Send m

The Send action fails if a synchronous Send attempt is made but fails or if the $inputPool$ could not be accessed.

SingleSend body of SingleSendNet transition



$CanAccess(sender, pool)$ iff

$sender = select_{Pool}(\{subject \mid TryingToAccess(subject, pool)\})$

$Blocked$ iff $SizeViolation(curMsg) = true$ **and**

$sizeAction(curMsg) = Blocking$

$\textsc{SingleSendNet}$ **component** $\textsc{PassMsg}$

$\textsc{PassMsg} =$

let $pool = inputPool(receiver(curMsg))$

if $($**not** $SizeViolation(curMsg))$ **or**

$SizeViolation(curMsg)$ **and** $sizeAction(curMsg) \neq DropIncoming$

then $\textsc{Insert}(curMsg, pool)$

if $sizeAction(curMsg) = DropYoungest$ **then**

$\textsc{Delete}(youngestMsg(pool), pool)$

if $sizeAction(curMsg) = DropOldest$ **then**

$\textsc{Delete}(oldestMsg(pool), pool)$

NB. The Send action with $sizeAction(curMsg) = DropIncoming$ in case of a $SizeViolation$ is considered to complete with success.

For $Rendezvous$-with-the-receiver predicate see below.

ReceiveActionReq for single messages

To satisfy the *MsgPreparationReq* we define:

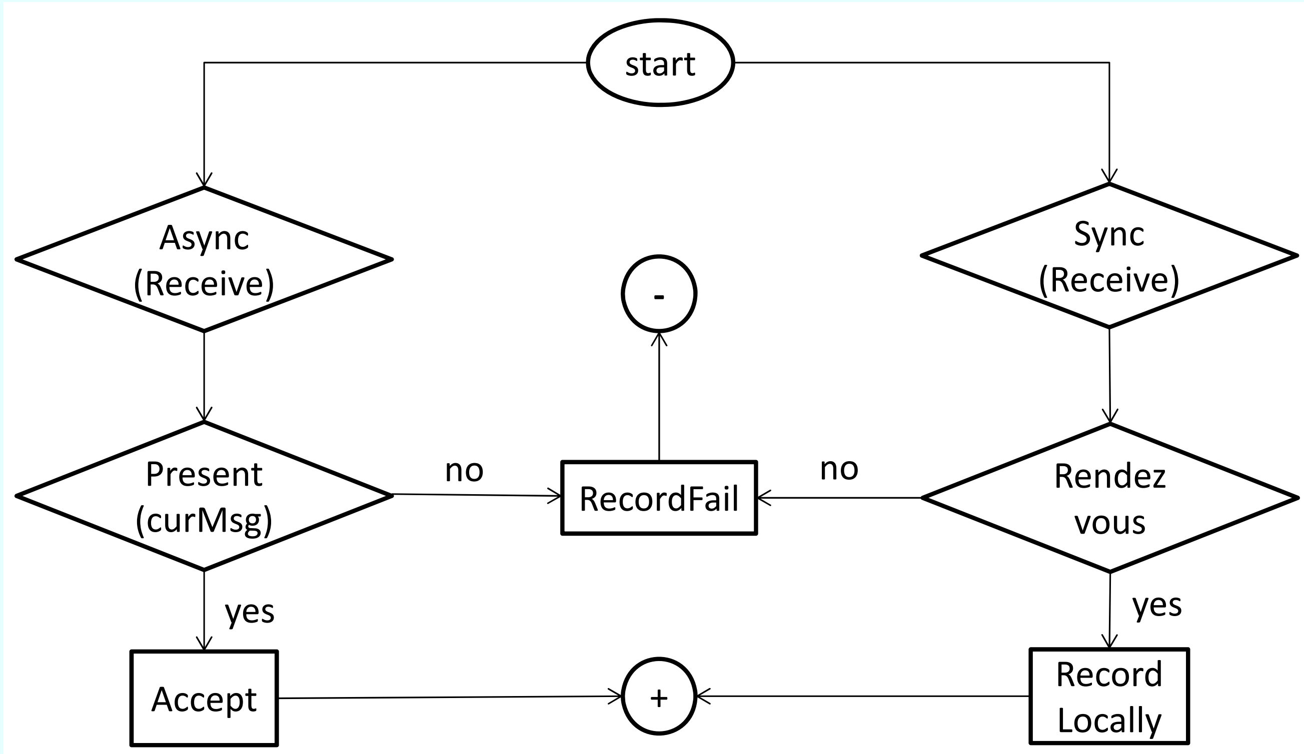
$$\textsc{Prepare}(curMsg, entry, Receive) =$$
$$\quad curMsg := select_{MsgKind}(ExpectedMsgKind(entry))$$
$$ExpectedMsgKind(entry) \subseteq \{(s,t,r) \mid s \in Sender \cup \{any\} \textbf{ and }$$
$$\quad t \in MsgType \cup \{any\} \textbf{ and } r \in \{sync, async\}\}$$

ReceiveActionReq. A Receive action can be of synchronous or asynchronous kind, specified as part of the kind of expected message described in the *MsgPreparationReq*uirement.

- An asynchronous Receive succeeds if the $inputPool$ contains a message of the kind of expected message.
- A synchronous Receive succeeds if there is a $sender$ which tries to synchronously Send a message of the kind of expected message.

Otherwise the Receive action fails.

SingleReceive **body of** SingleReceiveNet



$Async(Receive)$ iff $third(curMsg) = async$

$Sync(Receive)$ iff $third(curMsg) = sync$

$Present(curMsg)$ iff

forsome $msg \in inputPool\ Match(msg, curMsg)$

SINGLERECEIVE **components**

ACCEPT =

let *receivedMsg* =

$select_{Pool}(\{msg \in inputPool \mid Match(msg, curMsg)\})$

STORELOCALLY(*receivedMsg*)

RECORDSUCCESS(SINGLERECEIVE, *async*)

DELETE(*receivedMsg*, *inputPool*)

let $sender = \iota s(\{s \in Sender \mid CanAccess(s, inputPool(receiver))\}$

Rendezvous iff

$Blocked(sender)$ **and** $Sync(Receive)(receiver)$ **and**

$Match(curMsg(sender), curMsg(receiver))$

RECORDLOCALLY =

STORELOCALLY(*curMsg*(*sender*))

RECORDSUCCESS(SINGLERECEIVE, *sync*)

1st refinement: communication of multiple messages

Additional Requirement *MultiComActReq*: In one $ComAct$ion, a subject can handle a finite set $MsgToBeHandled$ of messages.

- The $mult$itude is defined at the SBD-node where such a $MultiComAct$ion takes place.
- The entire set of $MsgToBeHandled$ has to be prepared *before* the $SingleComAct$ion is performed for each of those messages.

$\textsc{Start}(\textsc{MultiComAct}, entry) = \textsc{PrepareMsg}(entry, ComAct)$

$\textsc{PrepareMsg}(entry, Send) = \textbf{forall } 1 \leq i \leq mult(entry)$

$\quad \textbf{let } m_i = composeMsg(msgData(entry, i))$

$\qquad MsgToBeHandled := \{m_1, \ldots, m_{mult(entry)}\}$

$\qquad RoundMsg := \{m_1, \ldots, m_{mult(entry)}\}$

$\textsc{PrepareMsg}(entry, Receive) = \textbf{forall } 1 \leq i \leq mult(entry)$

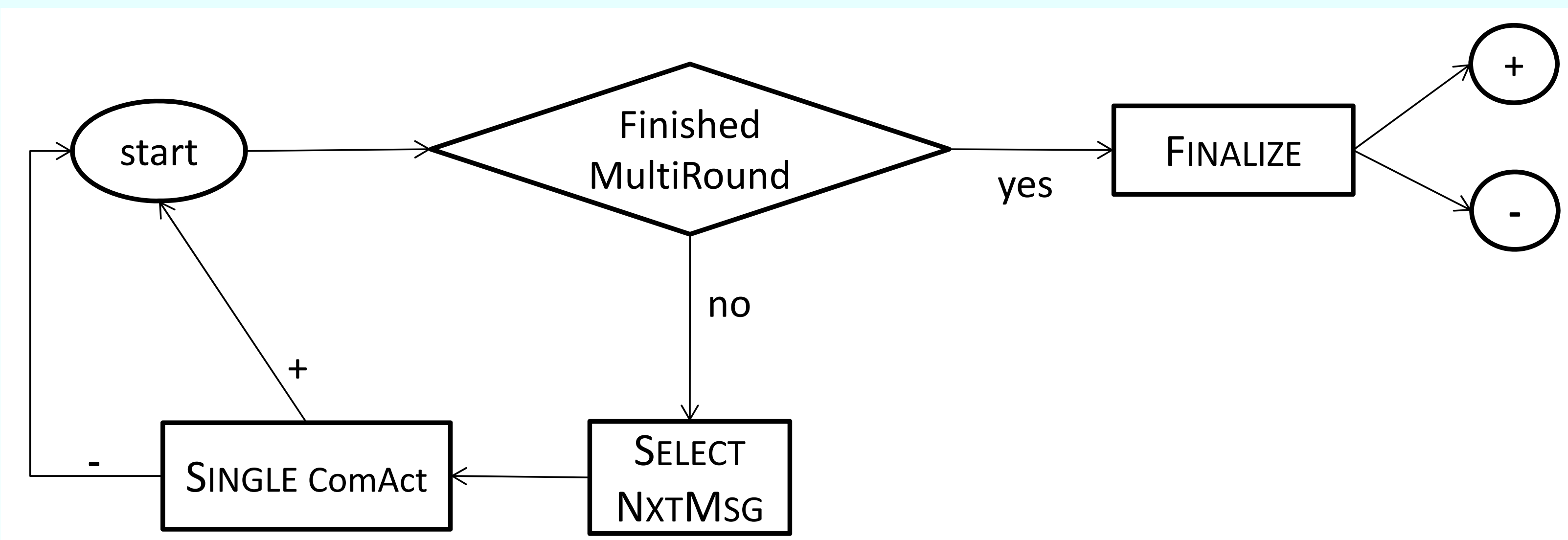
$\textbf{let } m_i = select_{MsgKind}(ExpectedMsgKind(entry), i)$

$\quad MsgToBeHandled := \{m_1, \ldots, m_{mult(entry)}\}$

MultiComActReq (Cont'd) and MultiComAct body

- To complete a *MultiComAct*ion the subject must Send/Receive the indicated *mult*itude of messages without pursuing in between any other communication.
- If for at least one $m \in MsgToBeHandled$ the *SingleComAct* fails, then the *MultiComAct*ion fails.

The MultiComAct body iterates *SingleComAct*ions:



Iteration component of $\textsc{MultiComAct}$

$FinishedMultiRound$ iff $MsgToBeHandled = \emptyset$

$\textsc{SelectNxtMsg} =$ -- scheduling kept abstract

choose $m \in MsgToBeHandled$

$curMsg := m$

$\textsc{Delete}(m, MsgToBeHandled)$

FINALIZE **component of** MULTICOMACT

if $Success(MultiRound, ComAct, RoundMsg)$ **then**

$\quad$ $\textsc{CompleteNormally}(ComAct)$

$\quad$ $PlusExitCond := true$

if $Fail(MultiRound, ComAct, RoundMsg)$ **then**

$\quad$ $\textsc{HandleMultiRoundFail}(ComAct)$

$\quad$ $MinusExitCond := true$

where

$\quad$ $Success(MultiRound, ComAct, X)$ iff

$\qquad$ **forall** $m \in X \;\; m \in SuccessRecord(\textsc{SingleComAct})$

2nd refinement: communication alternatives

The additional *AltComActReq* reads as follows:

To perform an $ComAct \in \{Send, Receive\}$ a subject can choose the set of $MsgToBeHandled$ among finitely many $Alternative$s.

- $Alternative$ is determined by a function $alternative(entry, ComAct)$.
- If the $ComAct$ succeeds for at least one alternative , the $AltComAct$ succeeds and can be $Completed$ normally.
- If the $AltComAct$ fails the subject chooses the next alternative until:
  - either one of them succeeds or
  - all $Alternative$s have been tried out and failed. In this case the $AltComAct$ fails.

ASM refinement to capture communication alternatives

To capture the new requirement, it suffices to put an iterator shell around MULTICOMACTNET

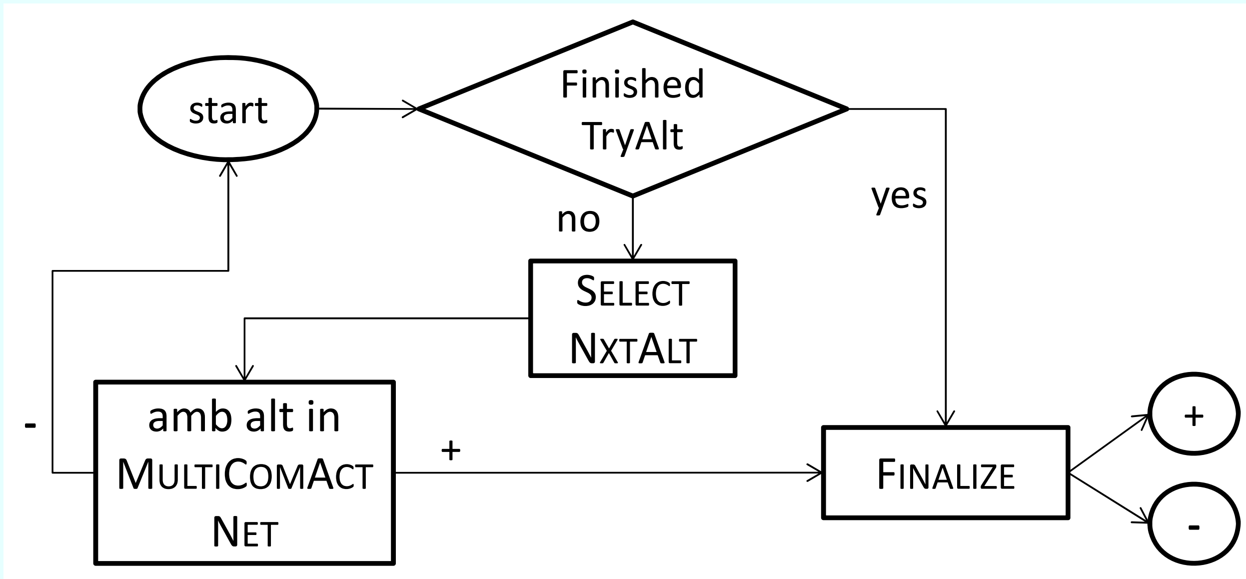
- trying out each element of $Alternative$ as current altternative to be executed by the MULTICOMACTNET

Use ambient ASMs for passing the chosen alternative to START

- declare $composeMsg$ and $select_{MsgKind}$ to be ambient dependent functions

Thereby the set of $MsgToBeHandled$ (together with its copy $RoundMsg$) defined by PREPAREMSG become parameterized by alt.

$\text{ALT}(ComAct)$ **body of** $\text{ALTNET}(ComAct)$



$$\text{START}(\text{ALT}(ComAct), entry) = \text{INITIALIZE}(Alternative, ComAct)$$

$$\text{INITIALIZE}(Alternative, ComAct) =$$

$$\quad Alternative := alternative(entry, ComAct)$$

$FinishedTryAlt$ iff $Alternative = \emptyset$

$\textsc{Alt}(ComAct)$ components

$\textsc{SelectNxtAlt} =$

choose $a \in Alternative$

$alt := a$

$\textsc{Delete}(a, Alternative)$

$\textsc{Finalize} =$

if $Success(\textsc{Alt}, ComAct)$ **then**

$\textsc{CompleteNormally}(\textsc{Alt}(ComAct))$

$PlusExitCond := true$

if $Fail(\textsc{Alt}, ComAct)$ **then**

$\textsc{HandleFail}(\textsc{Alt}(ComAct))$

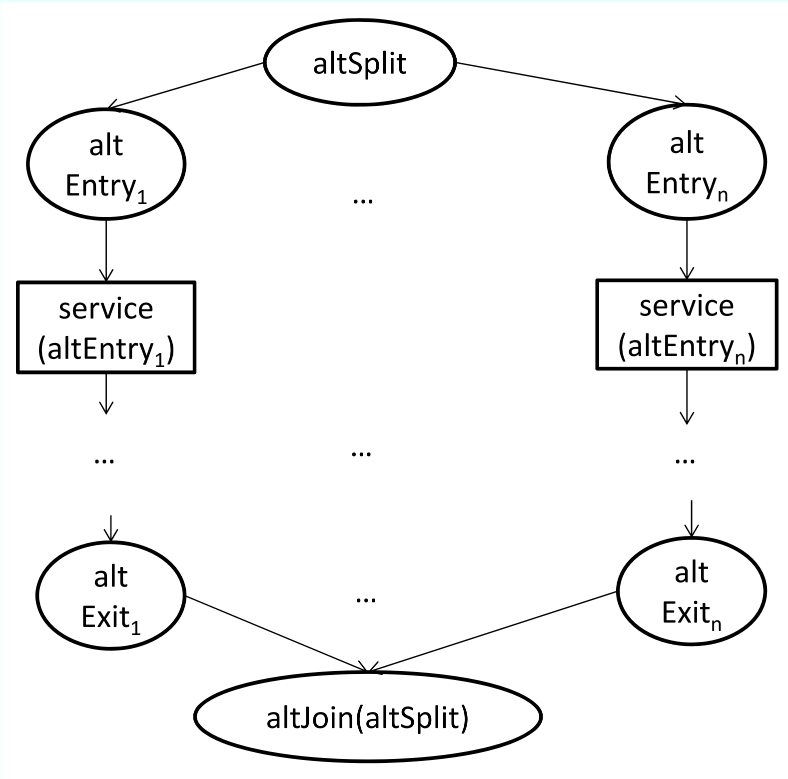
$MinusExitCond := true$

$Success(\textsc{Alt}, ComAct)$ iff **forsome** $a \in Alternative$

$Success(MultiRound, ComAct, RoundMsg(a))$

3d refinement: order independent work on multiple actions

AltActSubdiagramReq. In an *altSplit* node SBD splits into finitely many SBDs $D_i \in AltBehDgm(altSplit)$ with an arrow from *altSplit* to the unique $altEntry(D_i)$ (for each $1 \leq i \leq n$) and an arrow from its unique $altExit(D_i)$ to an *altJoin* node in the SBD.



Requirement on compulsory actions

CompulsoryDgmReq. Some subdiagram entries resp. exits are declared to be $Compulsory$ and determine the completion predicate of the $AltAction$ as follows:

- A *Compulsory* $altEntry(D_i)$ node must be entered during the run so that the D_i-subcomputation must have been started before the $AltAction$ can be $Completed$.
- A *Compulsory* $altExit(D_j)$ node must be reached in the run, for the $AltAction$ to be $Completed$, if during the run the D_j-subcomputation has been entered at $altEntry(D_j)$ (whether the $altEntry(D_j)$ state is *Compulsory* or not).

At least one subdiagram has *Compulsory* $altEntry$ and $altExit$.

Requirements for alternative action nodes

AltActionReq. To perform the $AltAction$ associated with an *altSplit* node means to complete some of the subdiagram computations, step by step in an interleaved (order-independent) way.

CompletionReq. The $AltAction$ associated with node *altSplit* is $Completed$ if all subdiagrams D_i with *Compulsory* $altEntry(D_i)$ have been entered and all computations of subdiagrams D_j with $Compulsory$ $altExit(D_j)$ have been $Completed$.

AltAction body

reuse of ASM net model SyncAsmNet:

- $mode$ location is implicitly parameterized by the SBD where it guides the control

so that step control in D_i is in terms of $mode_{D_i}$.

$\textsc{Start}(\textsc{AltAction}, altSplit) =$

forall $D \in AltBehDgm(altSplit)$

if $Compulsory(D)$ **then** $mode_D := initial(D)$

$\textsc{AltAction} =$

choose $D \in altBehDgm$

if $Active(D)$ **then** $\textsc{SyncAsmNet}(D)$

else $mode_D := initial(D)$ -- Start a subdiagram computation

Components of $\textsc{AltAction}$

$Active(D)$ iff $mode_D \neq \mathbf{undef}$ -- D has been started

$Completed(\textsc{AltAction}, altSplit)$ iff

forall $D \in AltBehDgm(altSplit)$

if $Compulsory(altEntry(D))$ **then** $Active(D)$

and if $Compulsory(altExit(D))$ **then** $mode_D = altExit(D)$

$\textsc{Finalize}(\textsc{AltAction}) =$

forall $D \in AltBehDgm(altSplit)$ $mode_D := \mathbf{undef}$

$mode := altJoin(altSplit)$

Remark on a BP certification procedure

- build correct models for meaning of (graphical) BP notations
  - define meaning in precise application domain terms
  - define ASM net ground models (*end-user-oriented domain-knowledge-expressing interfaces*) for the meaning
  - validate ground models to 'correctly' represent intended meaning
- provide guaranteed correct BP ground model
  - design BP using the defined (graphical) notations
  - inspect/validate BP design to correctly reflect intentions
- provide guaranteed correct ground model implementation
  - use resulting ground model ASM net as precise and complete spec for sw implementation of the BP
  - verify the coding to be correct

Result: implementation is guaranteed (and can be certified) to correctly reflect the meaning the BP expert intended by high-level BPM.

Degrees of certificate quality

Quality (degree of reliability) of a correctness certificate for a BP is proportional to the quality of:

- the ground model validation, e.g. by model inspection, model checking, model-based testing
- verification of the stepwise refinements used to develop/generate code for an executable version of the BP spec, e.g. by
  - compiling ground model ASM net using a verified compiler
  - providing proof sketches or standard mathematical or machine supported (interactive or fully automated) proofs of (some critical or all) code generating refinement steps

ASM Net approach to BP development offers all the ingredients which allow one to produce certifiably correct industrial BPs

- NB. This is a BP-specific version of Hoare's 'verified software grand challenge'.

References

- E. Börger and A. Raschke: Modeling Companion for Software Practitioners. Springer 2018 http://modelingbook.informatik.uni-ulm.de
- E. Börger and A. Fleischmann: *Abstract State Machine Nets. Closing the Gap between Business Process Models and their Implementation*.
  - Proc. S-BPM ONE 2015, ACM Digital Library ISBN 978-1-4503-3312-2
- A. Fleischmann, W. Schmidt, C. Stary, S. Obermeier, E. Börger: *Subject-Oriented Business Process Management*.
  - Springer Open Access Book 2012.

Copyright Notice