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 Companion¹
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.
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
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
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 **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**.
- **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 $\text{AsmRule}(\text{transition}, i)$

\[
\begin{align*}
&\text{let } i = i_k \\
&\text{if } mode = i \text{ and } \text{EntryCond}_k \text{ then} \\
&\quad \text{START}(M, i) \\
&\quad \text{mode} := \text{start}(M, i) \\
&\text{if } \text{active}(M, i) \text{ then} \\
&\quad \text{if not } \text{Completed}(M, i) \text{ then } M \\
&\quad \text{else} \\
&\quad \text{FINALIZE}(M, i) \\
&\text{Exit} \\
\end{align*}
\]

where

\[
\begin{align*}
\text{Exit} = \text{forall } j \in \{j_1, \ldots, j_m\} & \quad \text{if } \text{ExitCond}_j \text{ then } \text{mode} := j \\
\text{active}(M, i) \text{ iff } \text{mode}_{M,i} \in \text{Mode}(M, i) \\
\text{Mode}(M, i) = \text{set of } \text{mode}_{M,i} \text{ values of } M_i \text{ (instance of } M) \\
\end{align*}
\]
Synchronous/Asynchronous net interpretation

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

\[ \text{SYNCAsmNet}(N) = \]
\[ \forall \text{transition} \in N \ \text{AsmRule}_{\text{sync}}(\text{transition}) \]

where

\[ \text{AsmRule}_{\text{sync}}(\text{transition}) = \]
\[ \forall i \in \{i_1, \ldots, i_n\} \ \text{AsmRule}(\text{transition}, i) \]

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

\[ \text{CONCURAsmNet}(N) : \text{a family of} \]
- agents \( a_{\text{transition}} \) with program \( \text{AsmRule}_{\text{sync}}(\text{transition}) \) or
- agents \( a_{(\text{transition}, i)} \) with program \( \text{AsmRule}(\text{transition}, i) \)
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 *conditions*
  - *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.)
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 *nodes* 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 \( \text{SyncAsmNet}(SBD) \) because

- each SBD describes one sequential process, executed by one subject
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 \(\text{SYNCAsmNet}(SBD_k)\).
- \( P \) is a concurrent ASM, in fact a family of communicating ASMs, modeled as \(\text{ASYNCAsmNet}(P)\).

To describe single SBDs, we will use mainly ASM net transitions with one entry and two exits:
We concentrate on the transitions for nodes with an associated communication action $ComAct$, either a Send or Receive action.

The requirements for $ComnActions$ 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.
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 \( \text{maxSize} \in \{0, 1, \ldots\} \) (non-negative number),
- maximal number \( \text{maxFrom}(\text{sender}) \) of messages allowed from a *sender* or \( \text{maxFor}(\text{type}) \) 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 sizeActions can be taken:

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

**SyncReceiveReq.** \( \text{maxFrom}(\text{sender}) = 0 \) and/or \( \text{maxFor}(\text{type}) = 0 \) indicates that the owner of the \( \text{inputPool} \) accepts messages from the indicated \( \text{sender} \) and/or of the indicated \( \text{type} \) only via a rendezvous (synchronously). \( \text{size} = 0 \) 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 defining the **START** component of the **SingleSendNet** transition as follows:

\[
\text{START}(\text{SingleSend}, \text{entry}) = \text{PREPARE}(\text{curMsg}, \text{entry}, \text{Send})
\]

\[
\text{PREPARE}(\text{curMsg}, \text{entry}, \text{Send}) = \quad \text{curMsg} := \text{composeMsg}(\text{msgData}(\text{entry}))
\]

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

\[
\text{sender} = \text{select}_{\text{Pool}}(\{\text{subject} \mid \text{TryingToAccess}(\text{subject}, \text{pool})\})
\]

Blocked iff SizeViolation(curMsg) = true and

\[
\text{sizeAction}(curMsg) = \text{Blocking}
\]
**SingleSendNet** component **PassMsg**

**PassMsg =**

```latex
let pool = inputPool(receiver(curMsg))
```

```latex
if (not SizeViolation(curMsg)) or
   SizeViolation(curMsg) and sizeAction(curMsg) ≠ DropIncoming
then INSERT(curMsg, pool)
```

```latex
if sizeAction(curMsg) = DropYoungest then
   DELETE(youngestMsg(pool), pool)
```

```latex
if sizeAction(curMsg) = DropOldest then
   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 \textit{MsgPreparationReq} we define:

\begin{equation*}
\text{\textsc{Prepare}}(\text{curMsg}, \text{entry}, \text{Receive}) = \\
\text{curMsg} := \text{select}_{\text{MsgKind}}(\text{ExpectedMsgKind}(\text{entry}))
\end{equation*}

\text{ExpectedMsgKind}(\text{entry}) \subseteq \{ (s, t, r) \mid s \in \text{Sender} \cup \{ \text{any} \} \ \text{and} \ t \in \text{MsgType} \cup \{ \text{any} \} \ \text{and} \ r \in \{ \text{sync}, \text{async} \} \}

\textit{ReceiveActionReq}. A Receive action can be of synchronous or asynchronous kind, specified as part of the kind of expected message described in the \textit{MsgPreparationRequirement}.

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

Otherwise the Receive action fails.
Async(Receive) iff third(curMsg) = async
Sync(Receive) iff third(curMsg) = sync
Present(curMsg) iff
forsome msg ∈ inputPool Match(msg, curMsg)
**SingleReceive** components

\[
\text{Accept} =
\]

\[
\text{let receivedMsg } = \text{let receivedMsg } = \text{selectPool}(\{ \text{msg } \in \text{inputPool} | \text{Match}(\text{msg}, \text{curMsg}) \})
\]

\[
\text{StoreLocally}(\text{receivedMsg})
\]

\[
\text{RecordSuccess(SingleReceive, } \text{async})
\]

\[
\text{DELETE}(\text{receivedMsg}, \text{inputPool})
\]

\[
\text{let sender } = \text{is}(\{ \text{s } \in \text{Sender} | \text{CanAccess}(\text{s}, \text{inputPool}(\text{receiver})) \})
\]

\[
\text{Rendezvous iff}
\]

\[
\text{Blocked(sender) and Sync(Receive)(receiver) and}
\]

\[
\text{Match(curMsg(sender), curMsg(receiver))}
\]

\[
\text{RecordLocally } =
\]

\[
\text{StoreLocally(curMsg(sender))}
\]

\[
\text{RecordSuccess(SingleReceive, sync)}
\]
Additional Requirement \textit{MultiComActReq}: In one \textit{ComAction}, a subject can handle a finite set \textit{MsgToBeHandled} of messages.

- The \textit{multitude} is defined at the SBD-node where such a \textit{MultiComAction} takes place.
- The entire set of \textit{MsgToBeHandled} has to be prepared \textit{before} the \textit{SingleComAction} is performed for each of those messages.

\begin{align*}
\text{START}(\text{MultiComAct}, \text{entry}) &= \text{PREPAREMSG}(\text{entry}, \text{ComAct}) \\
\text{PREPAREMSG}(\text{entry}, \text{Send}) &= \forall 1 \leq i \leq \text{mult}(\text{entry}) \quad \text{let } m_i = \text{composeMsg}(\text{msgData}(\text{entry}, i)) \\
\quad \text{let } m_i = \text{composeMsg}(\text{msgData}(\text{entry}, i)) \\
\quad \text{MsgToBeHandled} := \{m_1, \ldots, m_{\text{mult}(\text{entry})}\} \\
\quad \text{RoundMsg} := \{m_1, \ldots, m_{\text{mult}(\text{entry})}\} \\
\text{PREPAREMSG}(\text{entry}, \text{Receive}) &= \forall 1 \leq i \leq \text{mult}(\text{entry}) \quad \text{let } m_i = \text{select}_{\text{MsgKind}}(\text{ExpectedMsgKind}(\text{entry}), i) \\
\quad \text{let } m_i = \text{select}_{\text{MsgKind}}(\text{ExpectedMsgKind}(\text{entry}), i) \\
\quad \text{MsgToBeHandled} := \{m_1, \ldots, m_{\text{mult}(\text{entry})}\}
\end{align*}
To complete a MultiComAction the subject must Send/Receive the indicated multitude of messages without pursuing in between any other communication.

If for at least one \( m \in \text{MsgToBeHandled} \) the SingleComAct fails, then the MultiComAction fails.

The MultiComAct body iterates SingleComActions:

![Diagram](image-url)
**Iteration component of** \textsc{MultiComAct}

\begin{align*}
\text{FinishedMultiRound} & \iff \text{MsgToBeHandled} = \emptyset \\
\text{SelectNxtMsg} & = \\
& \textbf{choose} \ m \in \text{MsgToBeHandled} \\
& \text{curMsg} := m \\
& \text{DELETE}(m, \text{MsgToBeHandled})
\end{align*}

-- scheduling kept abstract
Finalize component of MultiComAct

\[ \text{if } \text{Success}(\text{MultiRound}, \text{ComAct}, \text{RoundMsg}) \text{ then} \]
\[ \text{COMPLETENormally}(\text{ComAct}) \]
\[ \text{PlusExitCond} := \text{true} \]

\[ \text{if } \text{Fail}(\text{MultiRound}, \text{ComAct}, \text{RoundMsg}) \text{ then} \]
\[ \text{HANDLEMULTIROUNDFAIL}(\text{ComAct}) \]
\[ \text{MinusExitCond} := \text{true} \]

where

\[ \text{Success}(\text{MultiRound}, \text{ComAct}, X) \iff \]
\[ \text{forall } m \in X \quad m \in \text{SuccessRecord}(\text{SINGLECOMACT}) \]
The additional *AltComActReq* reads as follows:

To perform an *ComAct* ∈ \{Send, Receive\} a subject can choose the set of *MsgToBeHandled* among finitely many *Alternatives*.

- *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 *Alternatives* have been tried out and failed. In this case the *AltComAct* fails.
To capture the new requirement, it suffices to put an iterator shell around MultiComActNet
- trying out each element of Alternative as current alternative 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.
\textbf{\textsc{Alt}}(\textit{ComAct}) \textbf{body of} \textbf{\textsc{AltNet}}(\textit{ComAct})

\begin{center}
\begin{tikzpicture}
  \node [startstop] (start) {start};
  \node [decision] (finish) {Finished TryAlt};
  \node [process] (select) {SELECT NXTALT};
  \node [process] (ambalt) {amb alt in MULTI\textsc{ComAct} NET};
  \node [decision] (no) {no};
  \node [decision] (yes) {yes};
  \node [final] (final) {FINALIZE};

  \draw [arrow] (start) -- (finish);
  \draw [arrow] (finish) -- (no);
  \draw [arrow] (finish) -- (yes);
  \draw [arrow] (no) -- (ambalt);
  \draw [arrow] (yes) -- (select);
  \draw [arrow] (select) -- (final);
  \draw [arrow] (ambalt) -- (final);
  \end{tikzpicture}
\end{center}

\textbf{START}(\textbf{\textsc{Alt}}(\textit{ComAct}), \textit{entry}) = \textbf{\textsc{Initialize}}(\textit{Alternative}, \textit{ComAct})

\textbf{Initialize}(\textit{Alternative}, \textit{ComAct}) =

\textit{Alternative} := \textit{alternative}(\textit{entry}, \textit{ComAct})

\textit{FinishedTryAlt} \textbf{iff} \textit{Alternative} = \emptyset
\textbf{Alt}(ComAct) components

\textbf{SelectNxtAlt} =

\begin{itemize}
  \item \textbf{choose} \( a \in \text{Alternative} \)
  \item \( alt := a \)
  \item \textbf{DELETE}(a, \text{Alternative})
\end{itemize}

\textbf{Finalize} =

\begin{itemize}
  \item \textbf{if} \( \text{Success}(\text{Alt}, \text{ComAct}) \) \textbf{then}
    \item \textbf{COMPLETENORMALLY}(\text{Alt}(\text{ComAct}))
    \item PlusExitCond := \text{true}
  \item \textbf{if} \( \text{Fail}(\text{Alt}, \text{ComAct}) \) \textbf{then}
    \item \textbf{HANDLEFAIL}(\text{Alt}(\text{ComAct}))
    \item MinusExitCond := \text{true}
\end{itemize}

\( \text{Success}(\text{Alt}, \text{ComAct}) \) \textbf{iff} \( \text{forsome} \ a \in \text{Alternative} \)

\( \text{Success}(\text{MultiRound}, \text{ComAct}, \text{RoundMsg}(a)) \)
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.
**CompulsoryDgmReq.** Some subdiagram entries resp. exits are declared to be *Compulsory* and determine the completion predicate of the *AltAction* as follows:

- A *Compulsory* $\text{altEntry}(D_i)$ 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* $\text{altExit}(D_j)$ 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 $\text{altEntry}(D_j)$ (whether the $\text{altEntry}(D_j)$ state is *Compulsory* or not).

At least one subdiagram has *Compulsory* $\text{altEntry}$ and $\text{altExit}$.
**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*.
reuse of ASM net model $\text{SyncAsmNet}$:

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

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

$\text{START}(\text{AltAction}, \text{altSplit}) =$

\[
\begin{align*}
\text{forall } D \in \text{AltBehDgm}(\text{altSplit}) & \\
\text{if } \text{Compulsory}(D) \text{ then } mode_D := \text{initial}(D)
\end{align*}
\]

$\text{AltAction} =$

\[
\begin{align*}
\text{choose } D \in \text{altBehDgm} & \\
\text{if } \text{Active}(D) \text{ then } \text{SyncAsmNet}(D) & \\
\text{else } mode_D := \text{initial}(D) & \quad \text{-- Start a subdiagram computation}
\end{align*}
\]
Components of \texttt{AltAction}

\begin{align*}
\text{Active}(D) & \text{ iff } \text{mode}_D \neq \text{undef} & \text{-- } D \text{ has been started} \\
\text{Completed(AltAction, altSplit)} & \text{ iff } \\
& \text{forall } D \in \text{AltBehDgm(altSplit)} \\
& \text{ if Compulsory(altEntry}(D)) \text{ then Active}(D) \\
& \text{ and if Compulsory(altExit}(D)) \text{ then } \text{mode}_D = \text{altExit}(D) \\
\text{Finalize(AltAction)} = \\
& \text{forall } D \in \text{AltBehDgm(altSplit)} \text{ mode}_D := \text{undef} \\
& \text{mode} := \text{altJoin(altSplit)}
\end{align*}
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.
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’.
E. Börger and A. Raschke: Modeling Companion for Software Practitioners. Springer 2018
http://modelingbook.informatik.uni-ulm.de


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

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

i.e. in particular under the condition that

- the original authors are mentioned
- modified slides are made available under the same licence
- the (re-) use is not commercial