Journal of Universal Computer Science, vol. 23, no. 2 (2017), 129-145
submitted: 9/11/16, accepted: 13/2/17, appeared: 28/2/17 © *J.UCS*

Communication in Abstract State Machines

Egon Börger
(Università di Pisa, Dipartimento di Informatica, Italy
boerger@di.unipi.it)

Klaus-Dieter Schewe
(Software Competence Centre Hagenberg, Austria
kd.schewe@scch.at)

Abstract: Up to recently the majority of applications of the Abstract State Machines method for design and verification of computational systems used the shared variable approach. However in particular with distributed systems only various forms of communication may be available to share information. We define communicating ASMs by using instead of shared locations an explicit, abstract concept of Sending/Receiving messages which can be added to existing ASM execution engines. We aim to provide a definition which is a conservative extension of traditional ASMs, uniformly captures synchronous and asynchronous communication and is not bound to any specific message passing mechanism but can be instantiated to the major communication concepts in the literature. We illustrate the concept by an application to synchronous and asynchronous networks of communicating processes.

Key Words: distributed system, concurrency, Abstract State Machine, communication
Category: F.1.1, F.3.1, F.3.2, D.2.7

1 Introduction

Communication plays a major role for the design and verification of distributed algorithms. In state-based approaches it comes in handy to specify abstract communication schemes using shared variables; we mention some examples in the section on related work. Abstract State Machines (see [Börger and Stärk(2003)]) offer also a simple way to deal with communication using no shared functions, read: possibly parameterized variables which can be read and written by multiple processes. The only functions used are input (also called monitored) functions, output functions and local (private, also called controlled) functions (see Sect. 2).

In this paper we define in Sect. 2 *communicating* ASMs as ASMs with only internal actions (using only local functions) and Send/Receive actions affecting mailboxes. In this way we separate local (internal) computation steps of processes from their interaction behavior. We define these ASMs as a conservative extension of traditional ASMs (as defined in [Börger and Stärk(2003)]). As a consequence communicating ASMs could be implemented as instances of a CoreASM [CoreASM(2015)] extension which communicate through an appropriate API; see [Rothstein and Schreckling(2016)] for a project description (where communicating ASMs appear under the name Abstract State Interaction Machines) and [Schreckling and Rothstein(2016)] for the code and some examples.

We keep the communication actions abstract so that they can be instantiated to common communication concepts in the literature. See the section on related work for details. We illustrate the expressiveness and flexibility of communicating ASMs by a mathematically precise formulation of the synchronous (Sect. 3.2,3.3) and the asynchronous (Sect. 3.4) network execution models on which the investigation of distributed algorithms is based in [Lynch(1996)]. We then use these models to rigorously prove (Sect. 4) the correctness of the local synchronization scheme defined in [Lynch(1996), 16.2]. If this scheme is applied to a set of algorithms which are designed to be executed in the synchronous network model, then it allows one to run them with the same local behavior in the asynchronous network model. The abstractions ASMs offer for doing this simplify the construction itself as well as the correctness proof.

2 Communication in ASMs

In ASMs as defined in [Börger and Stärk(2003)] communication can and has been dealt with as particular form of interaction of processes using no shared function but only monitored (i.e. input), output and controlled (private, local) functions. A function that is output function for (i.e. only written by) a process p and monitored (i.e. only read) by each process q in a set of processes acts like a common mailbox location. In this location all those q simultaneously receive the value written by (i.e. sent from) p so that they can read it as input for their next step. The resulting model of what one could call *directly communicating ASMs* abstracts completely from any specific communication medium and works particularly well for the synchronous computation model; for details see the section on related work.

However, to model this way communication mechanisms with asynchronous Send/Receive actions or where messages may get lost one has to consider in addition when the processes q which monitor an output location of a sender process p receive the output value produced by p (or whether they receive it at all). To avoid such complications we define and analyze in this paper *communicating ASMs* as a class of multiple agent ASMs where machines perform only internal actions (read: ASM steps using only local functions to perform actions on their pairwise disjoint local states). Communication happens only via Send/Receive actions affecting mailboxes. We keep the mailbox concept and the Send/Receive actions abstract so that they can be instantiated to the main communication concepts used in the literature (see the section on related work). As will become clear below the definition works for both synchronous communication (typically in the context of synchronized parallel ASM runs like in Sect. 3.2,3.3) and asynchronous communication (typically in the context of concurrent ASM runs as defined in [Börger and Schewe(2016)], see for example Sect. 3.4).

Definition. *Communicating ASMs* (processes) are defined as families of single-agent ASMs $p = (ag(p), pgm(p), mailbox(p))$, consisting of

- an agent $ag(p)$ that executes step by step
- a program $pgm(p)$, called the rule(s) of the ASM, and is equipped with
- a $mailbox(p)$ for incoming messages

with disjoint signatures, read: private (also called internal) state and the following abstract communication actions and predicates besides the usual internal ones:

$\textsc{Send}(m, to(q))$ // deliver $(m, from(\mathbf{self})^1, to(q))$ to $mailbox(q)$
$Received(msg) = (msg \in mailbox(\mathbf{self}))$ // msg has been delivered
$\textsc{Consume}(msg) = \textsc{Delete}(msg, mailbox(\mathbf{self}))$

The abstract view of the $\textsc{Send}(m, to(q))$ action assumes that the sender process p knows the (address of the) receiver process q and triggers the (agent of the) communication medium to try to $\textsc{Insert}$ the message $(m, from(p), to(q))$ into the *mailbox* of the destination process. The delivery can happen in various ways, depending on the properties of the communication medium. We mention as outstanding examples:

- immediately: in the synchronous model with reliable communication where every message sent in one 'step' (synchronous round) is in the receiver's mailbox at the beginning of the next step,
- eventually: in the asynchronous model with reliable communication (where no message is lost) or with multiple delivery attempts (message repetition) such that at least one succeeds,
- maybe eventually (possibly never): in the asynchronous model with unreliable communication where messages can get lost, also called at-most-once delivery.

This abstracts from introducing explicit communication channels (see Sect. 3.1). Unless otherwise stated we treat *mailbox* as a set, but the following alternatives are also possible:

- *mailbox* could be a FIFO-queue or a priority queue as in *Akka* [Akka(2011-2016)];
- *mailbox* could be a FIFO-queue, where however the $\textsc{Receive}$ action may ignore or defer certain messages in the mailbox (as it happens in the P machines [Gupta et al.(2012)]);
- *mailbox* could be a multiset to distinguish among multiple occurrences of identical messages with same sender, receiver and message content (payload)—a set will do if sender timestamps are added to messages.

[1] **self** denotes the executing agent

When speaking about 'eventually' in the asynchronous model we refer to a logical time model as defined in [Lamport(1978)] and adopted by the "happens before" actor send rules in *Akka* [Akka(2011-2016)].

The above definition provides a conservative extension of traditional ASMs. One can interprete the communication related abstractions as update instructions (in the sense of CoreASM [CoreASM(2015)]. These are instructions which are defined to generate genuine ASM updates, depending on the intended kind of message delivery.

For the sake of generality we add *sender* resp. *receiver* to the message payload (content) m to form a complete message $(m, \mathit{from}(p), \mathit{to}(q))$. We use the notation $\mathit{payload}(\mathit{msg})$, $\mathit{sender}(\mathit{msg})$ and $\mathit{receiver}(\mathit{msg})$ for the message components. When it is clear from the context we omit notationally the sender or receiver of messages or write (m, p, q) instead of $(m, \mathit{from}(p), \mathit{to}(q))$. Often we notationally omit also agents and mailboxes, focussing on the ASM program.

The disjoint signature assumption guarantees that processes have no shared locations, differently from traditional ASMs (see [Börger and Stärk(2003)]). This does not exclude that two ASMs $(a, M, \mathit{mailbox}(a)), (b, N, \mathit{mailbox}(b))$ have the same program $M = N$ with the same function symbols; we simply assume all function symbols f as implicitly parameterized ('instantiated') by the executing agents a, b in the form f_a, f_b, thus guaranteeing disjoint locations (read: local states). It is also possible that communicating ASMs have input or output locations, but those are used only for providing input or output from/to the environment (if any) and not for interprocess communication (in case the environment is not seen as an additional process).

Remark on receive actions. The predicate $\mathit{Received}(\mathit{msg})$ can be interpreted as signalling that a RECEIVE(msg) event has happened at the receiver. RECEIVE is considered as an action processes can execute besides internal actions and the SEND action (see for example [Lamport(1978), Lynch(1996), Riccobene and Scandurra(2014)]). The predicate notation allows one to consider RECEIVE as an action of the communication medium affecting the receiver's mailbox and to abstractly describe when and how the receiver retrieves messages from its mailbox; see the section on related work for some examples.

Numerous concrete examples of communicating ASMs (without being named so) can be found in [Börger(2016)] and are not repeated here.

3 Network Computation Models

To test applying the concept of communicating ASMs we formulate in this section in terms of communicating ASMs the two principal execution models for networks of processes which are used in [Lynch(1996)] to describe distributed algorithms, namely:

- the synchronous network execution model (with message loss (Sect. 3.3) or without (Sect. 3.2) message loss),
- the asynchronous network execution model (including its variation with shared memory), see Sect. 3.4.

For each of these execution models examples for communicating ASMs executing distributed algorithms from [Lynch(1996)] can be found in [Börger(2016)]. As an illustration for a general relation between the synchronous and the asynchronous execution model we analyse in Sect. 4 the local synchronizer defined in [Lynch(1996), 16.2]. When it is applied to a set of algorithms which are designed to be executed in the synchronous network model, it allows one to run them with the same local behavior in the asynchronous network model.

3.1 Explicit notation for process mailboxes

Networks of processes are in general defined as directed graph $(Process, E)$ whose nodes are (labeled by) *processes which* SEND *messages only to their neighbors*, a frequent case for distributed algorithms:

- $inNeighb(p) = \{q \mid (q, p) \in E\}$: neighbors from which p receives messages
- $outNeighb(p) = \{q \mid (p, q) \in E\}$: neighbors to which p outputs messages
  - when it is clear from the context what is meant, for example when SENDing from p or when the graph is undirected, we write $neighb(p)$ instead of $outNeighb(p)$, similarly for $inNeighb(p)$
- in [Lynch(1996)] for each edge $(p, q) \in E$ a channel $chan_{p,q}$ is assumed to contain messages (usually at most one per state)[2] sent by p to q
  - abstracting from the details of the communication medium one can treat (in particular the singleton set view of) $chan_{p,q}$ as an output location of the sender p and an input location of the receiver q
- we abstract from such channels by using for each agent p a mailbox containing all messages which have been received by p on any channel via which it is connected to any of its $inNeighb(p)$:

$$mailbox(p) = chan_{q_1,p}, \ldots, chan_{q_{(in(p))},p}$$
$$\textbf{where } inNeighb(p) = \{q_1, ..., q_{in(p)}\}$$

Because of the abstraction from the details of the communication medium there seems to be no need to define separate notions $inMailbox(p)$ resp. $outMailbox(p)$ for incoming resp. outgoing messages.

[2] Analogously to *mailbox* also *chan*nels can be treated as sets, multisets, lists, etc., depending on what kind of communication mechanism one has to deal with.

3.2 Synchronous network system without message loss or delay

Let a network of processes $(Process, E)$ be given. In the synchronous network model of [Lynch(1996), Ch.2] a network *execution* is a sequence

$$(State_r, SentMsg_r, ReceivedMsg_r)_{r=1,2,\ldots}$$

of states—comprising the individual states of all processes—in 'round' r and of all messages sent resp. received in 'round' r by any process; here a 'round' leading from $State_r$ to $State_{r+1}$ consists of the following successive lock-step actions:

- each process in $State_r$ generates and sends messages to its (outgoing) neighbors in E,[3] followed by
- each process performs an action (considered as not furthermore analysed (atomic) internal computation step) using the messages just generated by and received from its (incoming) neighbors in E. These are the sent messages if no message is lost, otherwise it is a subset of the sent ones. As a result $State_{r+1}$ is obtained.

Representing the processes as communicating ASMs one can treat received messages as part of their state and sending messages as part of their action. In this way each process in each round performs the following three actions as one abstract step (NB. with ASMs a round consists of one step):

- ReceiveMsgs: read and possibly Consume some messages delivered to *mailbox*, a special case of reading a monitored location,
- ComputeNextState: update some controlled locations,
- SendMsgs: a special case of updating some output locations.

To follow the notation in [Lynch(1996)] we write a step as composed of separate internal and communication (Send/Receive) actions. Therefore we define for a set *Process* of communicating ASMs SyncNet$(Process, E)$ as an ASM with reliable immediate message delivery and the following rule. For the sake of comparison to the *rounds* needed in [Lynch(1996)] we include the ASM step counter *curRound*, assuming that initially $curRound = 0$.

SyncNet$(Process, E) =$
  forall $p \in Process$ // all processes perform simultaneously one step
    ReceiveMsgs$_p$ // read/consume messages in $mailbox(p)$
    ComputeNextState$_p$
    SendMsgs$_p$
  IncreaseRound // i.e. $curRound := curRound + 1$

We often write SyncNet$(Process)$ instead of SyncNet$(Process, E)$ if the graph structure E is clear from the context. Note that SyncNet$(Process)$ is a

[3] A *null* message is assumed to represent that no message (content) is sent.

single-agent ASM whose notion of run (see [Börger and Stärk(2003), Sect.2.4.3]) accurately defines 'executions' in the synchronous network model of [Lynch(1996), Ch.2]. Each single step of SyncNet(*Process*) models a 'round' of the synchronous network of *Process*es (assuming reliable communication with immediate delivery). For the sake of precision we mention that the model defined in [Lynch(1996), Ch.2] is deterministic, whereas using ASMs with the **choose** construct also a non-deterministic version is covered.

3.3 Synchronous network system with message loss but no delay

The notion of ASM run as defined in [Börger and Stärk(2003), Sect.2.4.3] supports also the variation of SyncNet(*Process*) where the communication is not reliable so that messages may get lost. In fact one can view the loss of messages as result of an environment action which happens between ASM steps (read: rounds) so that some messages sent by SendMsgs_p may not appear in $mailbox(q)$ of destination processes q.

One can easily make the effect of this environment action explicit in the ASM rule by splitting $chan_{p,q}$ into two versions: an $outChan_{p,q}$ location of p where p via SendMsgs_p places messages sent in round r to q and an $inChan_{p,q}$ location of q where q receives in round $r + 1$ some (maybe all) messages sent by p in the preceding round r.[4] This allows one to define LossySyncNet(*Process*, E) as SyncNet(*Process*) followed by a TransmitMsgs_p rule. This rule uses an abstract *NoLoss* predicate to describe that after all processes have performed their SendMsgs the communication medium (acting as synchronized environment) may deliver (only) some of the involved messages. Here **seq** denotes the turbo ASM sequentialization operator which turns two successive ASM steps P followed by Q into one atomic step P **seq** Q.

LossySyncNet(*Process*, E) =
  SyncNet(*Process*) **seq**
    forall $p \in Process$ TransmitMsgs_p // synchronized env action
  where
    TransmitMsgs_p = **forall** $q \in outNeighb_p$
      if $NoLoss(outChan_{p,q}, curRound)$
      then DeliverMsgFromTo(p, q)
    DeliverMsgFromTo(p, q) =
      $inChan_{p,q} := (m, from(p))$ **where** $outChan_{p,q} = (m, to(q))$

[4] As a consequence $mailbox(q)$ becomes $inChan_{p_1,q}, \ldots, inChan_{p_{in(q)},q}$ for $inNeighb(q) = \{p_1, ..., p_{in(q)}\}$.

3.4 Asynchronous network system

In [Lynch(1996)] the essential difference between the synchronous and an asynchronous network model is that in the latter 'there are no synchronous rounds for communication' but instead 'asynchrony in both the process steps and the communication' [Lynch(1996), pg.458]. Both process and communication channel components of an asynchronous send/receive network system are modeled in [Lynch(1996), Ch.14] as I/O automata. The network is modeled by their I/O composition where outputs of one component can be matched with (same-named) inputs of another component. Except for same-named actions in different components, which in any execution are always executed simultaneously by all those components, the model is based upon the interleaving assumption that 'actions are performed one at a time, in an unpredictable order' [Lynch(1996), p.201].

We generalize this concept in two directions. First we replace I/O automata by communicating ASMs. Their parallelism covers the concept of simultaneous execution of same-named actions in the composition of I/O automata. The concept of monitored and output locations covers the concept of input resp. output actions by which processes communicate with the environment, e.g. with an external user. Second we replace the interleaving assumption by concurrency. Thus for a set *Process* of communicating ASMs, nodes of a directed graph with edge set E, we define $\textsc{AsyncNet}(\mathit{Process}, E)$ as concurrent (multi-agent) ASM in the sense of [Börger and Schewe(2016)] whose components are the communicating ASMs in *Process*.

By this definition an execution (a *concurrent run*) of $\textsc{AsyncNet}(\mathit{Process}, E)$ is a sequence $S_0, S_1, \ldots$ of states—one may think of it as the union of the observed snapshots of the states of all processes—together with a sequence $P_0, P_1, \ldots$ of subsets of *Process* such that each state S_{n+1} is obtained from S_n as follows:

- applying to it all the updates computed by the processes $p \in P_n$. Each of them performs its current internal step on the basis of its current input (i.e. its current *mailbox* and if present the values of its monitored environment locations) in its current local state, i.e. the restriction of S_n to the signature of p,
- applying the delivery of messages by the communication medium (read: placing messages into the *mailbox* of their destination process) and in case applying the updates of monitored locations by the environment.[5]

The interleaving case is characterized by singleton sets P_n. We remind the

[5] Note that by asynchrony the communication medium acts as an additional independent process, appearing to each p in *Process* as part of the environment. In fact a $\textsc{Send}(m, \mathit{from}(p), \mathit{to}(q))$ performed by process p in state S may later change $\mathit{mailbox}(q)$ via message delivery and thereby q's substate in S without q performing a step in between.

reader that by asynchrony the messages any p finds in its *mailbox* in state S_n must have been sent by some other process in a preceding state S_j $(j < n)$.

As the communicating ASMs in *Process* are assumed to have disjoint signatures, there is an alternative representation of $\text{AsyncNet}(\textit{Process}, E)$ executions, which reflects the cartesian product structure defined in [Lynch(1996), Ch.8.2.1] for the composition of I/O automata to (a)synchronous network systems. Here executions are sequences F_n of families $(S_{stepsOf(p),p})_{p \in Process}$ of local states $S_{stepsOf(p),p}$ process p reaches after $stepsOf(p)$ steps where $stepsOf(p) \leq n$ (initially $stepsOf(p) = 0$) for each p. F_{n+1} is obtained from F_n by a) some processes p—in the general case each process in P_n, exactly one process in the interleaving case—performing one step in F_n, thereby updating $stepsOf(p)$ to $stepsOf(p) + 1$, and b) message delivery by the communication medium and (if any) updates of monitored locations by the environment.

The above definition of $\text{AsyncNet}(\textit{Process}, E)$ is easily adapted to the *asynchronous shared memory system* variation considered in [Lynch(1996), Ch.9]. It suffices to let the processes have shared locations (and use communication only for interaction with the environment), a property which is covered in ASMs by shared locations. In fact the definition in [Lynch(1996), Ch.9] is even more restrictive since to avoid inconsistency it allows in each step at most one process to update one shared location.

4 Local synchronization pattern

It is usually easier to design and verify distributed algorithms for execution in synchronous networks than in asynchronous networks, due to the characteristic common clock ('round') concept of synchronous process networks. One can exploit this experience by defining schemes for the synchronization of distributed network algorithms, as done in [Lynch(1996), Ch.16]. They allow algorithms defined for the synchronous model to be executed in an asynchronous network where the synchronization scheme maintains the intended local behavior of each algorithm.

In this section we generalize the local synchronizer of distributed network algorithms found in [Lynch(1996), Ch.16] to communicating ASMs. The synchronization is 'local' because it involves synchronization only among neighbor processes in the given network. To 'synchronize' a process p with its neighbors each step of p (read: *curRound* step) is executed one by one by a $syncShell(p)$ component with slightly modified program $retainMsgsExecuting(pgm(p))$ (see below). This component after each *curRound* p-step suspends executing p-steps until all messages sent in this round to p have been collected by a $synchronizer(p)$ component. This $synchronizer(p)$ component then transmits the messages to the mailbox of p and wakes up $syncShell(p)$ to execute the next p-step. The intended effect we prove below is that 'running p' in the resulting asynchronous network $\text{LocSyncPattern}(\textit{Process}, E)$ and running it in the synchronous network $\text{SyncNet}(\textit{Process}, E)$ yields for each p the same (local) behavior.

For a network $(\textit{Process}, E)$ of communicating ASMs we define for each $p \in \textit{Process}$ two components of $\text{LocSyncPattern}(\textit{Process}, E)$, an asynchronous network we use to simulate $\text{SyncNet}(\textit{Process}, E)$:

- a synchronization shell $\textit{syncShell}(p)$ with program $\text{SyncShell}(p)$ to execute p-steps under a $\textit{ReadyForNxtRound}(p)$-constraint,
- a local synchronizer $\textit{synchronizer}(p)$ with program $\text{Synchronizer}(p)$.

In $\text{LocSyncPattern}(\textit{Process}, E)$ each p is linked (read: via communication channels in both directions) to its $\textit{synchronizer}(p)$, each $\textit{synchronizer}(p)$ is linked to the $\textit{synchronizer}(q)$ of each neighbor q of p in E. Each $\textit{syncShell}(p)$ has

- a local (Boolean valued) flag $\textit{WaitingForNxtRoundTick}_p$ it Resets and
- a local 'round' counter $\textit{curRound}_p$ it Increases

to $\text{Suspend}(p)$ after the execution of one step of p, together with informing the *synchronizer* of each neighbor of p and of p itself by a $\textit{stepInfo}(p)$ message that it has *MadeOneStep* of p in the $\textit{curRound}_p$. Performing a step of p under the supervision of the $\textit{syncShell}(p)$ has for message sending the effect to temporarily retain messages. The $\textit{retainMsgsExecuting}(\textit{pgm}(p))$ is the transformation of $\textit{pgm}(p)$ where each

$\text{Send}(m, \textit{to}(q))$ is replaced by $\text{Send}((m, \textit{curRound}_p), \textit{to}(\textit{synchronizer}(q)))$
$\text{Received}(m, \textit{from}(q))$ is replaced by
$\quad \textit{ASMReceived}((m, \textit{curRound}_p - 1, \textit{from}(q)), \textit{from}(\textit{synchronizer}(p)))$

In other words the round information is added to the message payload and the neighbor q destination is replaced by the $\textit{synchronizer}(q)$, thus deviating the message to $\textit{mailbox}(\textit{synchronizer}(q))$.[6] $\text{SyncShell}(p)$ does $\text{Resume}(p)$ when a *resume* message from the $\textit{synchronizer}(p)$ is *Received*.

$\text{SyncShell}(p) =$
 if $\textit{ReadyForNxtRound}_p$ **then**
  $\textit{retainMsgsExecuting}(\textit{pgm}(p))$
  $\text{Suspend}(p)$
 elseif $\textit{Received}(\textit{resume}, \textit{from}(\textit{synchronizer}(p)))$ **then** $\text{Resume}(p)$
where
 $\text{Suspend}(p) =$
  $\text{Reset}(p)$

[6] We simplify the exposition by condensing the two-step message passing from p via its $\textit{synchronizer}(p)$ to $\textit{synchronizer}(q)$ of all $q \in \textit{neighb}(p)$ into one communication step.

$\quad\quad\text{InformAboutStep}(p)$
$\quad\quad\text{Increase}(curRound_p)$
$\quad\text{Reset}(p) = (WaitingForNxtRoundTick_p := true)$
$\quad\text{InformAboutStep}(p) =$
$\quad\quad\textbf{forall } q \in neighb(p) \cup \{p\}$
$\quad\quad\quad\text{Send}((stepInfo(p, curRound_p), from(p)), to(synchronizer(q)))$
$\quad\text{Increase}(l) = (l := l + 1)$
$\quad\text{Resume}(p) =$
$\quad\quad WaitingForNxtRoundTick_p := false$
$\quad\quad\text{Consume}((resume, from(synchronizer(p)))$
$\quad ReadyForNxtRound_p$ iff
$\quad\quad Resumed(p) \textbf{ and } ReceivedAllMsgsFor(curRound_p, p)$
$\quad Resumed(p)$ iff $WaitingForNxtRoundTick_p = false$
$\quad ReceivedAllMsgsFor(r + 1, p)$ iff
$\quad\quad\textbf{forall } q \in Neighb(p) \textbf{ forsome } m$ // msg sent in round r
$\quad\quad\quad Received((m, r, from(q)), from(synchronizer(q)))$

The $synchronizer(p)$ waits until p and each $q \in neighb(p)$ has *MadeOneStep* in the to be synchronized $curRound_{synchronizer(p)}$, waiting for a *stepInfo* message with payload $curRound_{synchronizer(p)}$ from each of those.[7] Then $synchronizer(p)$ will $\text{TransferMsgsTo}(p)$, Send a *resume*-message to $\text{WakeUp}(p)$ and last but not least Increase its *curRound*.[8]

$\text{Synchronizer}(p) =$
$\quad\textbf{let } r = curRound_{synchronizer(p)}$
$\quad\quad\textbf{if forall } q \in neighb(p) \cup \{p\}\ MadeOneStep(q, r)$
$\quad\quad\quad\textbf{and } ReceivedAllMsgsToPassTo(p, r)$
$\quad\quad\textbf{then}$
$\quad\quad\quad\text{TransferMsgsTo}(p, r)$
$\quad\quad\quad\text{WakeUp}(p)$
$\quad\quad\quad\text{Increase}(curRound_{synchronizer(p)})$
$\quad\textbf{where}$
$\quad\quad MadeOneStep(q, r) = Received(stepInfo(q, r), from(q))$
$\quad\quad\text{TransferMsgsTo}(p, r) =$
$\quad\quad\quad\textbf{forall } q \in neighb(p)$
$\quad\quad\quad\textbf{forall } ((m, r), from(q)) \in ProcessMsgs \cap mailbox_{synchronizer(p)}$

[7] For notational convenience we assume the set of such messages to be disjoint from the set *ProcessMsgs* of messages in mailboxes of synchronizers which record messages exhanged between *Process*es, e.g. appearing in a $\text{Send}(msg)$ of some $pgm(p)$ with $p \in Process$.

[8] A garbage collector would add a $\text{ClearSyncMailbox}(p, curRound)$ component to delete from $mailbox_{synchronizer(p)}$ all round messages $((m, r), from(q)) \in ProcessMsgs$ and all step info messages $(r, from(q))$ it contains.

$\textsc{Send}(m, \mathit{from}(q), \mathit{to}(p))$
$\textsc{WakeUp}(p) = \textsc{Send}(\mathit{resume}, \mathit{to}(\mathit{syncShell}(p)))$
$\mathit{ReceivedAllMsgsToPassTo}(p, r) =$
forall $q \in \mathit{Neighb}(p)$ **forsome** m
$\mathit{Received}((m, r, \mathit{to}(p)), \mathit{from}(\mathit{synchronizer}(q)))$

All SEND actions are assumed to be reliable. For SYNCNET it is also assumed that SENDing has immediate effect, i.e. messages sent in round r are *Received* and thereby in the mailbox of their destination in the next round.[9] A run R_s of $\textsc{SyncNet}(\mathit{Process}, E)$ is called properly initialized if initially $\mathit{curRound} = 0$ and $\mathit{mailbox}(p) = \emptyset$ for each $p \in \mathit{Process}$; a run R_a of $\textsc{LocSyncPattern}(\mathit{Process}, E)$ is called properly initialized if initially for each component c $\mathit{curRound}_c = 0$, $\mathit{mailbox}(c) = \emptyset$ and for each $p \in \mathit{Process}$ $\mathit{ReadyForNxtRound}_p = \mathit{true}$. For each round number r and each process p the round r of p consists in R_s of the one step p performs when $\mathit{curRound} = r$; in R_a its critical steps are the following three successive ones:

- a $\mathit{syncShell}(p)$ step when $\mathit{curRound}_p = r$ **and** $\mathit{ReadyforNxtRound}_p = \mathit{true}$,
- a $\mathit{synchronizer}(p)$ step when $\mathit{curRound}_{\mathit{synchronizer}(p)} = r$ and **forall** $q \in \mathit{neighb}(p) \cup \{p\}$ $\mathit{MadeOneStep}(q, r)$ and $\mathit{ReceivedAllMsgsToPassTo}(p, r)$,
- a $\mathit{syncShell}(p)$ step after having $\mathit{Received}(\mathit{resume}, \mathit{from}(\mathit{synchronizer}(p)))$ to $\textsc{Resume}(p)$ (whereafter eventually it will become again $\mathit{ReadyforNxtRound}_p$.

For simplicity of exposition we assume that in each round each process sends to each of its neighbors a (possibly empty) message.

Local Synchronization Correctness Property. Let R_s, R_a be any properly initialized runs of $\textsc{SyncNet}(\mathit{Process}, E)$, $\textsc{LocSyncPattern}(\mathit{Process}, E)$ started with equal values in same-named locations. For every process p and every round number r the following holds: when p starts its round r in R_s resp. in R_a the values of same-named p-locations are the same in the two runs and the payload and destination of messages p sends in round r to its neighbor processes (where they appear as *Delivered* from p in round $r + 1$) are the same.[10]

Formally the equality of values of same-named p-locations at the beginning of round r can be expressed as $\mathit{StateEquality}(p, r)$:

$$\mathit{state}_{R_s}(r) \downarrow \Sigma_p = \mathit{state}_{R_a}(p, r) \downarrow \Sigma_p$$

where $r = \mathit{curRound}_{\textsc{SyncNet}(\mathit{Process},E)} = \mathit{curRound}_p$, $\mathit{state}_{R_s}(r)$ denotes the state of $\textsc{SyncNet}(\mathit{Process}, E)$ in which p starts its r-th round in R_s, $\mathit{state}_{R_a}(p, r)$ denotes the state of $\mathit{syncShell}(p)$ in which p starts its round r in R_a, $S \downarrow \Sigma_p$ denotes the restriction of state S to the signature Σ_p of p.

[9] For LOCSYNCPATTERN this additional assumption is not needed.

[10] For each action $\textsc{Send}((m, \mathit{curRound}_p), \mathit{to}(\mathit{synchronizer}(q)))$ in R_a we call $(m, \mathit{to}(q))$ a message sent by p in $\mathit{curRound}_p$.

Proof by induction on r. From $StateEquality(p, r)$ (which for $r = 0$ is true by the initialization condition) and the first $SyncShell(p)$-step in round r it follows that the p-locations are updated the same way in both runs, without further updates until p becomes again $ReadyForNxtRound$ $r + 1$ by a RESUME step and the arrival of the messages sent in round r. Also for each message $(m, from(p), to(q))$ sent by p in round r in SYNCNET($Process, E$) to neighbor q, by the first $SyncShell(p)$-step the message

$$((m, r), from(p), to(synchronizer(q)))$$

is sent in R_a to $mailbox(synchronizer(q))$. From there $(m, from(p), to(q))$ is transfered to $mailbox(q)$ in the round r step of p performed by $synchronizer(p)$ when all neighbors of p have $MadeOneStep$ in round r. This establishes the message claim of the Local Synchronization Correctness Property. The last round r step of p in R_a performed by $syncShell(p)$ prepares making p $ReadyForNxtRound$ $r + 1$, namely when from its $synchronizer(p)$ it $ReceivedAllMsgsFor(r + 1, p)$.

5 Related work

We mentioned in the introduction that in state-based approaches it comes in handy to specify abstract communication schemes using shared variables. An example is the communication mechanism between abstract machines in the B-method [Abrial(1996)]: synchronous communication can happen among machines which are in the containment relation (with variable sharing governed by visibility rules), asynchronous communication via some component that provides variable sharing among one writer and possibly multiple reader components. Also the extension of synchronous communication between abstract B machines to dynamically created component machines (machines ‘instances’) in [Aguirre et al.(2005)] uses variable sharing, namely in synchronous calls of parameterized operations using connectors as communication links. For action systems asynchronous communication is defined in [Plosila et al.(2002)] using communication channels, each composed of a shared request/acknowledgement variable and a data variable to define a submachine call/return mechanism that allows the designer to compose abstract machines out of library components; the scheme is an instance of the composition of Abstract State Machines (ASMs) out of FSM-like control-state ASMs [Börger(1999)] where a separate submachine is executed in each control (for Finite State Machines called ‘internal’) state.

In the introduction we also mentioned that the ASM function classification offers in addition a simple way to deal with communication without using shared functions. This has been exploited in numerous applications of the ASM method. As example is [Barros and Börger(2005)] where various ASM interaction patterns have been defined based upon a few basic forms of bilateral and multilateral Send/Receive actions. They have been implemented within the ASM-based framework described in [Riccobene and Scandurra(2014)] for modeling and prototyping service-oriented applications.

As our definition keeps the the Send, Receive and mailbox concepts abstract, communicating ASMs can be instantiated to models of a large variety of common communication concepts. Each of the just mentioned communication patterns (see [Barros and Börger(2005)]) and their implementation (see [Riccobene and Scandurra(2014)]) are instances of communicating ASMs. An example in specification languages is the communication concept used in the ITU-T language SDL-2000 where signals are transported over possibly delaying channels which connect agents via gates; see [Glässer et al.(2003), 3.2.1] for an ASM specification of the SDL-signal-flow communication model. Other instances of communicating ASMs can be found in the communication concept of concurrent programming languages, e.g. *Erlang* [Erlang(1999-2016), Larson(2008)] or *Akka* [Akka(2011-2016)] which use an Actor Model [Hewitt et al.(1973)]. The concept of a dynamic set of agents which execute each some (possibly changing) ASM allows us to abstract from particular features actors come with. In fact the set of ASM agents can be instantiated to the specific hierarchical features in *Akka* [Akka(2011-2016)]—*Children* and *Supervisor Strategy*—or the *worker process* concept in *Erlang* [Erlang(1999-2016), Larson(2008)]. In those languages they are used to implement the creation and supervision of agents for (sub-) task delegation.

The definition of communicating ASMs abstracts from the specifics of the reliable SEND and of the pattern-matching-based (possibly blocking) RECEIVE actions in *Erlang* [Larson(2008), Fig.2] and from the functional and sequential specifics of *Erlang* programs.

A similar remark applies to the control-state ASM [Börger(1999)] like component machines of P programs [Gupta et al.(2012)] and their interesting responsiveness concept: message delivery is immediate and reliable but to a mailbox queue with a possibly delayed RECEIVE action (read: retrieving events for processing from the input queue).

Also concrete mailbox concepts like input-pools in the business process modeling method S-BPM turn out to be specifiable by instances of communicating ASMs. This concerns in particular the input-pool-configuration-dependent RECEIVE actions in S-BPM [Fleischmann et al.(2012), pg.335-337]. Another example is in the area of message routing in networks. In [Glässer et al.(2004)] the authors define an ASM to model the routing of messages through a network. They use an abstract encoding of the network topology by a routing table to specify how messages are transferred between processes which are connected to the network.

Our definition is influenced by the abstract way communication is treated in [Lynch(1996)] within a state-based specification framework. However by choosing ASMs as programs with which agents are equipped we generalize the notion of programs used in [Lynch(1996)] and abstract from their sequential nature. *Directly communicating ASMs* mentioned in the introduction work particularly well for the synchronous computation model. See for example CSP [Hoare(1985)], where all processes which are linked by the producer/consumer relation—between processes which output to a function f and those which monitor f—are synchronized by a common clock and communication does not fail. A similar remark applies to CCS [Milner(1982)] and the π-calculus [Milner(1999)] (see the ASM model for it in [Glavan and Rosenzweig(1993)]).

6 Conclusion

We have equipped traditional ASMs with an abstract form of the three basic communication constructs (*mailbox*, *Send* and *Receive* actions). We briefly pointed out by refering to the related literature that the resulting class of *communicating* ASMs permits uniform descriptions of common synchronous and asynchronous communication concepts. We have illustrated the concept by defining rigorous models for synchronous and asynchronous networks of communicating processes and proving the correctness of a local synchronization scheme for such networks.

Acknowledgement

The research reported in this paper results from the project *Behavioural Theory and Logics for Distributed Adaptive Systems* supported by the **Austrian Science Fund (FWF: [P26452-N15])**. The research has further been supported by the Austrian Ministry for Transport, Innovation and Technology, the Federal Ministry of Science, Research and Economy, and the Province of Upper Austria in the frame of the COMET center SCCH **(FFG: [844597])**.

References

[Abrial(1996)] Abrial, J.-R.: The B-Book; Cambridge University Press, Cambridge, 1996.

[Aguirre et al.(2005)] Aguirre, N., et al.: "Towards dynamically communicating abstract machines in the B method"; K.-K.Lau, R. Banach, eds., ICFEM 2005; volume 3785 of LNCS; 141–155; Springer, 2005.

[Akka(2011-2016)] Akka: "Akka Java/Scala documentation"; `http://www.akka.io` (2011-2016).

[Barros and Börger(2005)] Barros, A., Börger, E.: "A compositional framework for service interaction patterns and communication flows"; K.-K. Lau, R. Banach, eds., Formal Methods and Software Engineering. Proc. 7th International Conference on Formal Engineering Methods (ICFEM 2005); volume 3785 of LNCS; 5–35; Springer, 2005.

[Börger(1999)] Börger, E.: "High-level system design and analysis using Abstract State Machines"; D. Hutter, W. Stephan, P. Traverso, M. Ullmann, eds., Current Trends in Applied Formal Methods (FM-Trends 98); volume 1641 of Lecture Notes in Computer Science; 1–43; Springer-Verlag, 1999.

[Börger(2016)] Börger, E.: "Modeling distributed algorithms by Abstract State Machines compared to Petri Nets"; M. Butler, K.-D.Schewe, A. Mashkoor, M.Biro, eds., ABZ 2016; volume 9675 of Lecture Notes in Computer Science; 3–34; Springer-Verlag, 2016; dOI: 10:1007/978-3-319-33600-8.1.

[Börger and Schewe(2016)] Börger, E., Schewe, K.-D.: "Concurrent abstract state machines"; Acta Inf.; 53 (2016), 5, 469–492; http://link.springer.com/article/10.1007/s00236-015-0249-7, DOI 10.1007/s00236-015-0249-7.

[Börger and Stärk(2003)] Börger, E., Stärk, R. F.: Abstract State Machines. A Method for High-Level System Design and Analysis; Springer, 2003.

[CoreASM(2015)] CoreASM: "The CoreASM Project"; `https://github.com/CoreASM/` (2015).

[Erlang(1999-2016)] Erlang: "Erlang on-line documentation"; `http://www.erlang.org/docs` (1999-2016).

[Fleischmann et al.(2012)] Fleischmann, A., et al.: Subject-Oriented Business Process Management; Springer Open Access Book, Heidelberg, 2012; `www.springer.com/978-3-642-32391-1`.

[Glässer et al.(2003)] Glässer, U., et al.: "Formal semantics of SDL-2000: Status and perspectives"; Computer Networks; 42 (2003), 3, 343–358.

[Glässer et al.(2004)] Glässer, U., et al.: "Abstract communication model for distributed systems"; IEEE Transactions on Software Engineering; 30 (2004), 7, 1–15.

[Glavan and Rosenzweig(1993)] Glavan, P., Rosenzweig, D.: "Communicating evolving algebras"; E. et al., ed., Computer Science Logic (CSL'92); volume 702 of LNCS; 182–215; Springer, 1993.

[Gupta et al.(2012)] Gupta, V., et al.: "P: Safe asynchronous event-driven programming"; Technical Report MSR-TR-2012-116; Microsoft Research (2012).

[Hewitt et al.(1973)] Hewitt, C., et al.: "A universal modular ACTOR formalism for artificial intelligence"; IJCAI'73 Proceedings of the 3rd international joint conference on Artificial Intelligence IJCAI'73; 235–245; Morgan Kaufmann Publishers Inc., San Francisco,CA, 1973.

[Hoare(1985)] Hoare, C. A. R.: Communicating Sequential Processes; Prentice-Hall, 1985; available at `http://www.usingcsp.com/`.

[Lamport(1978)] Lamport, L.: "Time, clocks, and the ordering of events in a distributed system"; Commun. ACM; 21 (1978), 7, 558–565.

[Larson(2008)] Larson, J.: "Erlang for concurrent programming"; acmqueue Magazine 6.5, pages 18-23 (2008); available through ACM Digital Library `http://dl.acm.org/citation.cfm?id=1454463`.

[Lynch(1996)] Lynch, N. A.: Distributed Algorithms; Morgan Kaufmann, 1996.

[Milner(1982)] Milner, R.: A Calculus of Communicating Systems; Springer, 1982; iSBN 0-387-10235-3.

[Milner(1999)] Milner, R.: Communicating and Mobile Systems: The Pi-Calculus; Springer, 1999; iSBN 9780521658690.

[Plosila et al.(2002)] Plosila, J., et al.: "Design with asynchronously communicating components"; F. deBoer et al., ed., FMCO 2002; volume 2852 of LNCS; 424–442; Springer, 2002.

[Riccobene and Scandurra(2014)] Riccobene, E., Scandurra, P.: "A formal framework for service modeling and prototyping"; Formal Aspects of Computing; 26 (2014), 10771113.

[Rothstein and Schreckling(2016)] Rothstein, E., Schreckling, D.: "Execution framework for interaction computing"; EU Project BIOMICS

Deliverable D5.2 of WP5: Proof-of-Concept Prototype of Interaction Computing Environment (2016); http://www.biomicsproject.eu/file-repository/category/11-public-files-deliverables?download=271:deliverable-5-2-execution-framework-for-interaction-computing. An implementation is available in [Schreckling and Rothstein(2016)].

[Schreckling and Rothstein(2016)] Schreckling, D., Rothstein, E.: "ICEF - Interaction Computing Execution Framework"; https://github.com/biomics/icef (2016); code for ICEF as specified in [Rothstein and Schreckling(2016)], see also http://docs.icef.apiary.io/.