Synchronous Message Passing and Semaphores: An Equivalence Proof

Iain Craig[1] and Egon Börger[2]

[1] Visiting Researcher, Department of Computer Science, University of York
(Correspondence address: Flat 4, 34 Sherbourne Road, Birmingham, UK)
idcraig@talktalk.net

[2] Dipartimento di Informatica, Università di Pisa, Italy boerger@di.unipi.it

Abstract. A natural encoding of synchronous message exchange with direct wait-control is proved to be equivalent in a distributed environment to a refinement which uses semaphores to implement wait control. The proof uses a most general scheduler, which is left as abstract and assumed to satisfy a few realistic, explicitly stated assumptions. We hope to provide a scheme that can be implemented by current theorem provers.

1 Introduction

This paper is part of an endeavor a) to rigorously model kernels of small but real-life operating systems (OS) at a high level of abstraction, b) to verify mathematically the major OS properties of interest in the models and c) to refine the models in a provably correct way by a series of steps to implementation code. So that the refinement steps reflect the major design decisions, which lead from the abstract models to executable code, and to make their correctness mathematically controllable, we use algorithmic (also called operational) models. Thus we deviate from the axiomatic (purely declarative) point of view underlying the Z-based formal models of operating system kernels in the two recent books [5, 6], which are, however, our starting point. We are careful to ensure that our models, technically speaking, Abstract State Machines [4], whose semantic foundation justifies our considering them an accurate version of pseudo-code, are understandable by programmers without further training in formal methods, so that they can be effectively used in practice by designers and teachers for a rigorous analysis of OS system functionalities.

In a first paper [3] we presented the method, focussing on modeling the behavior of the clock process, more specifically the clock interrupt service routine, which interacts with a priority-based scheduler of device, system and user processes. The paper uses an abstract form of synchronous message passing. In this paper we show that a natural high-level specification of synchronous message exchange which uses a direct wait control mechanism has a provably correct refinement (in fact is equivalent to it) which uses semaphores to implement the wait control.

The equivalence of a direct implementation of synchronous message passing and of one using semaphores is often cited in the literature (e.g., [7]). The purpose of this paper is to illustrate how one can turn this equivalence claim, in a way which supports the intuitive operational understanding of the involved concepts, into a precise mathematical statement and prove it, with reasonable generality, that is, in terms of a faithful abstract model and its refinement.

A natural direct implementation of the synchronous message-passing mechanism uses a queue of processes, whose running mode is controlled by explicit intervention of the scheduler (Sect. 3)[1]. An alternative implementation uses a semaphore structure, which allows one to separate the issues related to process control (handing the permission to processes to run in a critical section) from the message transfer functionality (Sect. 5). In a specification which already includes semaphores as a component, using semaphores in the refinement of an implementation that uses direct process control makes sense because it reduces the complexity of individual code modules as well as the related verification tasks. Proceeding this way is an example of the use of standard components for stepwise refinement in a sense similar to the library types often provided with object-oriented programming languages.

We collect, in Sect. 2, the minimal assumptions that must be guaranteed for the underlying scheduling mechanism. We exhibit these assumptions to clarify the sense in which our model and its verification apply to any OS scheduling policy encountered in real life. In Sect. 4, we specify the underlying semaphore concept. The definitions in these two sections are re-elaborations of those given in [3] to which we refer for their motivation. The equivalence proof as presented in Sect. 6 applies only to the uniprocessor case.

2 Scheduling

Both semaphores and synchronous message passing require a scheduler and its associated operations. In this section, we briefly provide this background, formulating the minimal assumptions needed for the equivalence proof.

In the uniprocessor case, an operating system comes with a notion of a unique currently-executing process, which we denote by *currp*. It is an element of a dynamic set *Process* of processes, or it is the *idleProcess*. The *currp*process can be thought of as the only one with $status(currp) = \mathsf{run}$ (read: instruction pointer *ip* pointing into its code). The scheduler selects it from a queue, *readyq*, of processes whose *status* is *ready* to execute, possibly respecting some predefined priorities (e.g. selecting device processes before system processes and those before user processes, see [3]). We make no specific assumptions as to how *readyq*ueue is ordered, so that any fair scheduling policy (as specified by the assumptions stated below) is covered by our equivalence proof.

Only two scheduler operations are required, the exact nature of which can remain abstract. One operation, defined by SuspendCurr below, suspends a sender process, *currp*, when it has sent a message to a *dest*ination process. The

[1] One reviewer suggested to use instead the "await" blocking rule constructor defined in [1] for concurrent ASMs. We want to keep the construction here as elementary as possible to easen the implementation of the equivalence theorem by theorem provers.

other operation, defined by MAKEREADY(*src*) below, makes the sender (also called source) process, *src*, ready again when the message has been communicated to (read: received by) the receiver process, *dest*.

The main effect of SUSPENDCURR is to update *currp* to the next process, which is selected by the scheduler from the *readyq*. We denote the selected element by *head*(*readyq*) to convey the idea that the selection is made based upon the order of the queue; the function *head* is defined by the scheduler and need not be further specified here. In addition, the selected process is deleted from the *readyq* and assigned the *status* run. Also, the current *state* of the previous current process must be saved and the *state* of the new current process must be restored. Since these state update operations are orthogonal to the mere scheduling part of SUSPENDCURR, we denote them here by two abstract machines that are not further specified: SAVESTATE and RESTORESTATE. Their execution is considered to be atomic, which amounts to using a locking mechanism whose specifics we can ignore here without loss of generality.[2] This leads to the following definition, where we use an abstract machine DEQUEUE to denote the deletion of an element that has been chosen from *readyq*.[3]

```
SuspendCurr =
  let $h = head(readyq)$[4] in
    UpdateState($currp, h$)
    $currp := h$
    $status(h) :=$ run
    Dequeue($h, readyq$)
where
  UpdateState($p, p'$) =
    SaveState($p$)
    RestoreState($p'$)
```

The effect of MAKEREADY(p) is to ENQUEUE(p, *readyq*) and to update its *status* to ready. As a result of executing MAKEREADY(p), process p becomes a candidate for the scheduler's next choice for updating *currp*. Here, we do not specify the order of *readyq*, so ENQUEUE is left as an abstract machine, which is assumed to respect the intended *readyq* order when inserting an element.

```
MakeReady($p$) =
  Enqueue($p, readyq$)
  $status(p) :=$ ready
```

[2] The locking effect is captured here by the atomicity of ASM steps, see [3].

[3] We remind the reader that due to the simultaneous parallel execution of all updates in an ASM rule, *currp* on the right hand side of := denotes the *prev*ious value and *currp* on the left hand side of := the newly assigned value.

[4] In the special case of an empty *readyq*, the well known *idleProcess* is chosen to become the new *currp*. This case is of no importance here, so that, without loss of generality, we omit it in this rule to streamline the exposition.

3 Directly Controlled Synchronous Message Passing

In both this section and section Sect. 5, we assume an abstract set, *Msg*, of messages as given. We analyze synchronous message passing between processes as triples of three steps, each described using abstract machine components as follows:

- StartSending out a message m from a source *src* to a *dest*ination
  - by recording m, say in a location $outbox(src)$, for the transfer

  and Wait for the synchronization with *dest*;
- StartSynchronizing with a source process *src* and Wait for the synchronization with *dest* to become effective;
- Upon synchronization bw *src* and *dest*:
  - DeliverMsg to the *dest*ination process, say by transfer of the value of $outbox(src)$ into a location $inbox(dest)$
  - TerminateSynchronization.

For both the direct implementation and the one using semaphores, we define a version for each of the named components. Note that StartSend&Wait and StartSync&Wait can be executed in any order, depending on whether the sender or the receiver initiates the message passing.

3.1 The MsgPass$_{Ctl}$ machine (ground model)

A natural way directly to control synchronous message passing in a distributed environment consists of making sender and receiver wait for each other until each one has 'seen' the other. To describe the waiting phase, one can use an extension of the scheduler's *status* control. To do this, on the sender side, execution of StartSend&Wait$_{Ctl}(m, src, dest)$ will RecordForTransfer(m, src) the message m, suspend the sender *src* (read: the currently executing process), switch its *status* to sndr and Enqueue(src) in a collection, $wtsndr(dest)$, of senders waiting for their message to be received by the *dest*ination process. The operation StartSync&Wait$_{Ctl}$ of *dest* consists of, first, testing whether there is a message waiting to be received. If $wtsndr(dest)$ is empty, *dest* switches to receiver *status* rcvr and suspends itself, whereafter only a sender move can MakeReady$(dest)$ again. If $wtsndr(dest)$ is found not to be empty (any more), both parties are synchronized. This triggers the receiver's second move PassMsg$_{Ctl}$ to DeliverMsg in the receiver side and to terminate the synchronization (i.e. MakeReady the sender and delete it from $wtsndr(dest)$).

The preceding protocol description is formalized by the following definitions. We deliberately leave the Enqueue and Dequeue operations abstract; their instantiation depends on the scheduling policy[5].

[5] Note, however, that for our equivalence proof in Sect. 6, the uniprocessor assumption is used. It guarantees that at each moment in MsgPass$_{Ctl}$, at most one Enqueue$(src, wtsndr(dest))$ move is made (as

```
$\textsc{Send}_{Ctl}(m, src, dest) = \textsc{StartSend\&Wait}_{Ctl}(m, src, dest)$ where
  $\textsc{StartSend\&Wait}_{Ctl}(m, src, dest) =$
    $\textsc{RecordForTransfer}(m, src)$
    $status(src) := \mathsf{sndr}$
    $\textsc{Enqueue}(src, wtsndr(dest))$
    if $status(dest) = \mathsf{rcvr}$ then $\textsc{MakeReady}(dest)$
    $\textsc{SuspendCurr}$
  $\textsc{RecordForTransfer}(m, src) = (outbox(src) := m)$[6]
```

For the definition of the submachine $\textsc{PassMsg}_{Ctl}$ of $\textsc{Receive}_{Ctl}$, it remains to decide whether the protocol should simultaneously permit multiple senders to $\textsc{StartSend\&Wait}_{Ctl}$ a message to the same *dest*ination process and in the positive—the more general—case, whether, and possibly how, such sender processes should be ordered in $wtsndr(dest)$. This decision influences the property that can be proved in the equivalence theorem. Since the usual model of semaphores is that they work with queues, we assume in the following that $wtsndr(dest)$ is a (possibly priority) queue.

$\textsc{Receive}_{Ctl}$ splits into a step $\textsc{StartSync\&Wait}_{Ctl}$ followed by $\textsc{PassMsg}_{Ctl}$. To formalize this sequentiality in the context of simultaneous parallel execution of ASM rules, we use the interruptable version of sequential execution introduced for ASMs in [3][7]. It is borrowed from the traditional FSM-control mechanism and denoted **step**[8]. Since $wtsndr(dest)$ is treated as a queue, we again use a 'head' function, denoted hd, to select (possibly in a priority-based manner) the next element to be $\textsc{Dequeue}$d from $wtsndr(dest)$.

```
$\textsc{Receive}_{Ctl}(dest) =$
```

part of a $\textsc{StartSend\&Wait}_{Ctl}(m, src, dest)$ move) and, in $\textsc{MsgPass}_{Sema}$, at most one corresponding $\textsc{WaitSema}(insema(dest))$ move (as part of a $\textsc{StartSend\&Wait}_{Sema}(m, src, dest)$ move—see the definition of $\textsc{Send}_{Sema}(m, src, dest)$, below).

[6] Since $\textsc{StartSend\&Wait}_{Ctl}$ suspends the sender, there is no buffering of messages at the sender side; i.e. *outbox* is only an internal location for recording a message a sender wants to send in a synchronized fashion, it is not a message box.

[7] One reviewer observed that given the mutually exclusive guards of $\textsc{StartSync\&Wait}_{Ctl}$ and $\textsc{PassMsg}_{Ctl}$, we could have avoided here (but not in the analogous situation of $\textsc{Send}_{Sema}$ in Sect. 5) to use the **step** notation, which forces the receiver to each time perform two steps (the first of which may result in only changing the implicit control state). However, eliminating **step** here would slightly complicate the comparative analysis of sender and receiver moves in the two protocols in Sect. 6, where we exploit the simple correspondence of message exchange triples.

[8] M_1 **step** M_2 consists of two rules:

```
if $ctl\_state = 1$ then
  $M_1$
  $ctl\_state := 2$
```

and the same with interchanging 1 and 2.

```
  $\text{StartSync\&Wait}_{Ctl}(dest)$ **step** $\text{PassMsg}_{Ctl}(dest)$
where
  $\text{StartSync\&Wait}_{Ctl}(p) =$
    if $wtsndr(p) = \emptyset$ then
      $status(p) :=$ rcvr
      SuspendCurr
  $\text{PassMsg}_{Ctl}(p) =$
    if $wtsndr(p) \neq \emptyset$ then
      let $src = hd(wtsndr(p))$ in
        $\text{DeliverMsg}(src, p)$
        $\text{TerminateSync}(scr, p)$
  $\text{DeliverMsg}(q, p) = (inbox(p) := outbox(q))$
  $\text{TerminateSync}(s, p) =$
    $\text{Dequeue}(s, wtsndr(p))$
    $\text{MakeReady}(s)$
```

MsgPass_{Ctl} denotes an asynchronous (also called distributed) ASM where sender agents are equipped with the $\text{StartSend\&Wait}_{Ctl}$ program and receiver agents with the Receive_{Ctl} program.

3.2 Properties of MsgPass_{Ctl} runs

In this section, we justify the definition of MsgPass_{Ctl} by proving that it specifies a correct synchronous message passing scheme, which, under minimal scheduler assumptions, is also a fair one. First of all we have to clarify what correctness and fairness mean in this context.

Due to the distributed context and depending on the scheduler, it may happen that multiple senders send a message to the same destination process before the latter has had a chance to synchronize with any of the former. This produces an asymmetry between sender and receiver moves: it is decided at the receiver's side which one of the waiting senders is considered next for synchronization. Therefore the waiting phase a sender *src* enters to send a message is terminated only by the synchronization with the *dest*ination process (see the *SndrWait* property in Theorem 1), whereas the receiver *dest* may enter and terminate various waiting phases (namely for receiving messages from other senders) before entering its waiting phase for *src* (if at all), which can be terminated only by synchronization with *src* (see the *RcvrWait* condition in Theorem 1).

To support the reader's general intuitions, we speak in this section of sender (source) or receiver (destination) process to refer to an agent with program $\text{StartSend\&Wait}_{Ctl}$ or Receive_{Ctl}, respectively. Saying that a process p is scheduled, is a shorthand for $p = currp$. In the following theorem, we first formulate the (intuitive requirements for the) correctness property in general terms and then make them precise and prove the resulting statement for MsgPass_{Ctl}.

Theorem 1. *(Correctness Property for Message Passing) The following properties hold in every run of* MsgPass_{Ctl}.

SndrWait *Whenever a source process src is scheduled for sending a message, m, to a receiver process dest, it will record the message for transfer and then wait (without making any further move) until it is synchronized with dest; it will wait forever if it cannot be synchronized with dest. In* MsgPass$_{Ctl}$*, src is said to be synchronized with dest when the receiver process dest has performed a* StartSync&Wait *move and is scheduled, ready to receive a message sent from src (i.e.* $src = hd(wtsndr(p))$*).*

RcvrWait *Whenever a process, dest, is scheduled for receiving a message, it will wait (i.e. not perform any further move) until it is synchronized with a sender process src.*

Delivery *Whenever the sender src of a message and its receiver dest are synchronized, the message is delivered at the receiver process, one at a time and at most once, and the synchronization of the two processes terminates.*

To turn the wording of this theorem into a precise statement for MsgPass$_{Ctl}$ so that it can be proved, we use the notion of runs of an asynchronous (multi-agent) ASM. Such a run is defined as a partially ordered set of moves (execution of ASM steps by agents) which a) satisfies an axiomatic coherence condition, b) yields a linear order when restricted to the moves of any single agent (sequentiality condition), c) for each single move of any agent has only finitely many predecessor moves (so-called finite history condition). The coherence condition guarantees that for each finite run segment, all linearizations yield runs with the same final state (see [4] for a detailed definition).

The axiomatic description of the notion of a run of asynchronous (multi-agent) ASMs fits well with the purpose of this paper. Without providing any information on how to construct a class of admissible runs (read: to implement a scheduler), it characterizes the minimal ordering conditions needed so that, when of a run the ordering of some moves of some agents could matter for the outcome, this ordering appears explicitly in the ordering conditions (read: the partial order). Therefore the properties of runs of an arbitrary MsgPass$_{Ctl}$ that we formulate and prove in this section, hold in full generality for every implementation or refinement of MsgPass$_{Ctl}$ by a scheduling policy that extends the partial order (e.g. to a linear order).

Proof. For the rest of this section, whenever we speak of a run we refer to an arbitrarily given, fixed run of MsgPass$_{Ctl}$ with respective sender and receiver agents. The proof of Theorem 1 is by induction on the number of times a process is scheduled for sending or receiving a message in MsgPass$_{Ctl}$ runs.

The first two claims of property *SndrWait* follow from the execution of the first three updates and the SuspendCurr submachine of StartSend&Wait$_{Ctl}$. The *waiting phase of a sender* is characterized here by the following two properties of the sender: it must

- be in sndr *status*—which prevents it from being scheduled because, in order to be scheduled, a process must be in ready *status* and in (usually at the head of) the scheduler's *readyq*ueue;
- be an element of *wtsndr* of some destination process.

The third claim follows because, by the definition of $\textsc{MsgPass}_{Ctl}$, only a $\textsc{PassMsg}_{Ctl}$ move can $\textsc{MakeReady}(src)$; for this to happen, the *dest*ination process involved, upon being scheduled,[9] must have determined *src* as its next waiting sender to synchronize with. Before $\textsc{PassMsg}_{Ctl}(dest)$ checks this 'readiness to receive from *src*' condition (namely by the guard $src = hd(wtsndr(dest))$), by definition of $\textsc{Receive}_{Ctl}(dest)$, the *dest* process must already have been scheduled to execute its $\textsc{StartSync\&Wait}_{Ctl}(dest)$ move once (possibly a **skip** move, which does not cause the *status* to change to rcvr *status*).

For the proof of the *RcvrWait* property, let us assume that a process, *dest*, is scheduled to receive a message from some sender process and has executed its $\textsc{StartSync\&Wait}_{Ctl}(dest)$ step. Case 1: there is no waiting sender. Then *dest* switches to *status* rcvr. At this point, by definition of $\textsc{MsgPass}_{Ctl}$, only a $\textsc{StartSend\&Wait}_{Ctl}$ move can $\textsc{MakeReady}(dest)$, after which *dest* can again be scheduled by the scheduler. Case 2: $wtsndr(dest) \neq \emptyset$. Then *dest* is still scheduled (unless, for some reason, the scheduler deschedules it, possibly rescheduling it again later). In both cases, whenever *dest* is scheduled, it is ready to receive a message from the sender process, *src*, it finds at the head of its queue of waiting senders ($src = hd(wtsndr(dest))$). Therefore *dest* and *src* are synchronized so that now, and only now, the $\textsc{DeliverMsg}$ can take place, as part of the $\textsc{PassMsg}_{Ctl}(dest)$ move, for the message sent by *src*.

The *Delivery* property holds for the following reason. When two processes, *src* and *dest*, are synchronized, by definition of $\textsc{PassMsg}_{Ctl}(dest)$, exactly one execution of $\textsc{DeliverMsg}(src, dest)$ is triggered, together with the machine $\textsc{TerminateSync}(scr, dest)$ which terminates the sender's waiting phase. Note that *dest*, by being scheduled, has just terminated its waiting phase. Note that the one-at-a-time property holds only for the uniprocessor case.

□

Remark on Fairness. Although in the presence of timeouts fairness plays a minor role, fairness issues for $\textsc{MsgPass}_{Ctl}$ can be incorporated into Theorem 1.

An often-studied fairness property is related to overtaking. For example, to guarantee that messages are delivered (if at all) in the order in which their senders present themselves to the receiver (read: enter its *wtsndr* collection), it suffices to declare $wtsndr(p)$ as a queue where the function *hd* in $\textsc{PassMsg}$ is the head function. In addition, one has to clarify the order in which senders simultaneously presenting to the same *dest*ination process are enqueued into $wtsndr(dest)$.

Any fairness property of the underlying scheduler results in a corresponding fairness property of $\textsc{MsgPass}_{Ctl}$. For example, if the scheduler repeatedly schedules every (active) process, every message sent can be proved eventually to be delivered to the receiver.

[9] Obviously, if *dest* is never scheduled, *src* can never synchronize with it.

4 Semaphores

We borrow the ASM specification of semaphores from [3]. For the equivalence proof in Sect. 6 binary semaphores, often called *mutual exclusion* or *mutex* semaphores, suffice. They permit at most one process at any time in their critical section, thus bind their counter to one of two values (e.g. 0,1) at any time.

A (counting) semaphore, s, has two locations, a counter and a queue of processes waiting to enter the critical section guarded by s, written $semacount(s)$, resp. $semaq(s)$ or just *semacount* resp. *semaq* if s is clear from the context. The semaphore counter is usually initialized to a value $allowed(s) > 0$, the number of processes simultaneously permitted by s in its associated critical section; the semaphore queue is assumed to be initialized to an empty queue.

Semaphores have two characteristic operations: WaitSema, which is executed when trying to access the critical section, and SignalSema, which is executed when leaving the critical section. WaitSema subtracts 1 from *semacount*; SignalSema adds 1 to it. As long as *semacount* remains non-negative, nothing else is done when WaitSema is scheduled, so that *currp*process can enter the critical section. If *semacount* is negative, at least *allowed* processes are currently in the critical section. Therefore, in this case, the WaitSema move will SuspendCurr, add *currp* to the semaphore queue *semaq* and put it into *status* semawait(s). Only a later SignalSema move can bring the suspended process back to *ready status* (see below). This leads to the following definition, where we use abstract Enqueue and Dequeue operations. For the sake of generality we use a *caller* parameter, which we will use below only for $caller = currp$.[10]

```
WaitSema($s$, $caller$) =
  let $newcount = semacount(s) - 1$ in
    $semacount(s) := newcount$
    if $newcount < 0$ then
      Enqueue($caller$, $semaq(s)$)
      $status(caller) :=$ semawait(s)
      SuspendCurr
WaitSema($s$) = WaitSema($s$, $currp$)
```

The Signal operation, which is assumed to be performed each time a process leaves the critical section, adds one to *semacount*. If the new value of *semacount* is not yet positive, *semaq* still contains some process that is waiting to enter the critical section. Then the process which first entered *semaq* is removed from the queue and made ready, so that (when scheduled) it can enter the critical section.

```
SignalSema($s$) =
  let $newcount = semacount(s) + 1$ in
    $semacount(s) := newcount$
```

[10] The operations performed by Wait and Signal must be atomic, as guaranteed by an ASM step. In the literature, auxiliary Lock and UnLock operations guarantee the atomicity.

```
if $newcount \leq 0$ then
  let $h = head(semaqueue(s))$ in
    DEQUEUE($h, semaqueue(s)$)
    MAKEREADY($h$)
```

SEMA(s) is the ASM with rules WAITSEMA(s, *currp*), SIGNALSEMA(s).

Theorem 2. *(Semaphore Correctness Theorem) For every semaphore, s, and every properly initialized* SEMA(s) *run the following properties hold:*

Exclusion *There are never more than allowed(s) processes within the critical section guarded by s.*

Fairness *If every process that enters the critical section eventually leaves it with a* SIGNALSEMA(s) *move and if the head function is fair (i.e. eventually selects every element that enters semaqueue(s)) and if the scheduler is fair, then every process that makes a* WAITSEMA(s) *move in an attempt to enter the critical section will eventually enter it.*

Proof. The exclusion property initially holds because the semaphore queue is initialized to empty. Since the initial counter value satisfies *semacount*(s) = *allowed*(s) and in each WAITSEMA(s), resp. SIGNALSEMA(s) move, the counter is decremented, resp. incremented, after successive *allowed*(s) WAITSEMA(s) moves that are not yet followed by a SIGNALSEMA(s) move, every further move WAITSEMA(s) before the next SIGNALSEMA(s) move has *newcount* as a negative number. Thus it triggers an ENQUEUE(*currp*, *semaq*(s)) operation, blocking *currp* from entering the critical section until enough SIGNALSEMA(s) moves turn the content of *semacount*(s) into a non negative value.

To prove the fairness property, assume that in a given state a process makes a WAITSEMA(s) move. If in this move, *newcount* is not negative, then the process is not ENQUEUEd into the semaphore queue and thus can directly enter the critical section. Otherwise, the process is ENQUEUEd into *semaq*(s) where, by the first two fairness assumptions, it will eventually become the value of *head*(*semaq*(s)) and thus be made *ready*. Then the scheduler, which is assumed to be fair, will eventually schedule it, so that the process, from the point where it was suspended by its WAITSEMA(s) move, does enter the critical section. □

5 Semaphore-based Synchronous Messages

In this section, we define a specification MSGPASS$_{Sema}$ of synchronous message passing using semaphores, which can be viewed as a refinement of MSGPASS$_{Ctl}$. We start from scratch, thinking about what is needed for a most general solution of the problem, i.e. a semaphore-based solution with minimal assumptions. We then define the meaning of correctness of MSGPASS$_{Sema}$ and provide a direct proof for the correctness theorem. The equivalence theorem in Sect. 6 together with the correctness Theorem 1 for MSGPASS$_{Ctl}$ provide an alternative correctness proof for MSGPASS$_{Sema}$.

5.1 The MsgPass$_{Sema}$ machine

The idea is to refine entering a sender's waiting period, which has to take place as part of a StartSend&Wait$_{Sema}(m, src, dest)$ move, to a WaitSema move for a binary semaphore $insema(dest)$, which for handshaking is signalled by the receiver process $dest$. Thus the receiver controls the permission, given at any time to at most one process src, to write to its $inbox(dest)$. To let the receiver get ready again after its handshaking move, but only after the DeliverMsg$(src, dest)$ move has been made, its $inbox(dest)$-write-permission move SignalSema$(insema(dest))$ is coupled to a WaitSema move for another binary semaphore $outsema(dest)$, which in turn is signalled by the sender process when DeliverMsg$(src, dest)$ is performed.

In the following and for the equivalence theorem in Sect. 6, both semaphores $insema(dest)$ and $outsema(dest)$ are assumed to be initialized with 0.

This leads to the following definition of an asynchronous MsgPass$_{Sema}$ by sender, resp. receiver, agents with rules Send$_{Sema}$, resp. Receive$_{Sema}$, where Send$_{Sema}$ is sequentially composed out of a StartSend&Wait$_{Sema}$ and PassMsg$_{Sema}$ submachine.

```
Send$_{Sema}(m, src, dest)$ =
  StartSend&Wait$_{Sema}(m, src, dest)$ **step** PassMsg$_{Sema}(src, dest)$
**where**
  StartSend&Wait$_{Sema}(m, src, dest)$ =
    RecordForTransfer$(m, src)$[11]
    WaitSema$(insema(dest))$
  PassMsg$_{Sema}(src, dest)$ =
    DeliverMsg$(src, dest)$[12]
    SignalSema$(outsema(dest))$

Receive$_{Sema}(dest)$ = StartSync&Wait$_{Sema}(dest)$
**where**
  StartSync&Wait$_{Sema}(d)$ =
    SignalSema$(insema(d))$
    Wait$(outsema(d))$
```

Note that a Send$_{Sema}(m, src, dest)$ move is executed when $currp = src$ and a Receive$_{Sema}(dest)$ move when $currp = dest$.

5.2 Correctness Proof for MsgPass$_{Sema}$

Theorem 3. *(Correctness Property for Message Passing in* MsgPass$_{Sema}$*) The message passing correctness properties SndrWait, RcvrWait and Delivery hold in every run of* MsgPass$_{Sema}$.

[11] RecordForTransfer is defined as for StartSend&Wait$_{Ctl}$.
[12] DeliverMsg is defined as for Receive$_{Ctl}$.

Proof. To turn the theorem into a precise statement, it remains to define when exactly two processes *src* and *dest* are synchronized in $\textsc{MsgPass}_{Sema}$ runs. We define this to be true in any one of the two following situations, depending on whether the sender or the receiver starts the attempt to synchronize for a message exchange:

SenderStarts *src* has made a $\textsc{StartSend\&Wait}_{Sema}(m, src, dest)$ move that $\textsc{Enqueue}$ed it into the $insema(dest)$-queue); thereafter *dest* has made a (first) $\textsc{Receive}_{Sema}(dest)$ move that $\textsc{Dequeue}$d *src* from the $insema(dest)$-queue and *src* is again scheduled.

ReceiverStarts *dest* has made a $\textsc{Receive}_{Sema}(dest)$ move which is followed by a (first) $\textsc{StartSend\&Wait}_{Sema}(m, src, dest)$ move that does not $\textsc{Enqueue}$ *src* into the $insema(dest)$-queue.

SenderStarts implies that, from its $\textsc{StartSend\&Wait}_{Sema}(m, src, dest)$ move until reaching the synchronization point, *src* stayed in the $insema(dest)$-queue without making any further move. *ReceiverStarts* implies that between the two moves $\textsc{Receive}_{Sema}(dest)$ and $\textsc{StartSend\&Wait}_{Sema}(m, src, dest)$, *dest* has made no move; it will be $\textsc{Dequeue}$d from the $outsema(dest)$-queue by the subsequent *src*-move $\textsc{PassMsg}_{Sema}(src, dest)$. Note that in the *ReceiverStarts* case, after the indicated $\textsc{StartSend\&Wait}_{Sema}(m, src, dest)$ move *src* remains scheduled (or will be rescheduled after an interrupt).

From the definition of synchronization and of $\textsc{MsgPass}_{Sema}$, the correctness properties *SndrWait*, *RcvrWait* and *Delivery* follow by an induction on $\textsc{MsgPass}_{Sema}$ runs.
□

Remark. The above algorithm for exchanging messages using semaphores was implemented as a collection of threads in C by the first author. Experiments with this implementation demonstrated that it behaves in a fashion equivalent to synchronous message passing. In the next section we mathematically define and prove this equivalence for the above two specifications.

6 Equivalence Proof

In this section, we formulate what it means (and prove) that $\textsc{MsgPass}_{Sema}$ and $\textsc{MsgPass}_{Ctl}$ are equivalent. We base the analysis upon the notion of correct (and complete) ASM refinement defined in [2] and show that $\textsc{MsgPass}_{Sema}$ is a correct refinement of $\textsc{MsgPass}_{Ctl}$, and vice versa (completeness). As a by-product, this implies, by Theorem 1, an alternative proof for the correctness of $\textsc{MsgPass}_{Sema}$. The two machines have different operations; also the ways the operations are structured slightly differ from each other. Therefore, we have to investigate in what sense one can speak of pairs of corresponding and possibly equivalent runs in the two machines.

In $\textsc{MsgPass}_{Ctl}$ or $\textsc{MsgPass}_{Sema}$ runs, each successful message exchange is characterized by a *message exchange triple* of moves

```
    (StartSend&Wait(m, src, dest) | StartSync&Wait(dest))
  ; PassMsg(dest)
```

where by $m \mid m'$, we indicate that the two moves m, m' can occur in any order in the run in question (remember: both a sender and a receiver can initiate an attempt to synchronize with a partner for message exchange), and by $m; m'$ the sequential order of m preceding m' in the run note: a synchronized PassMsg move can come only after the corresponding two moves StartSync&Wait and StartSend&Wait). The message exchange triple components are furthermore characterized by the following requirements:

- The StartSync&Wait$_{Ctl}$(*dest*) move is the last StartSync&Wait$_{Ctl}$ move of the receiver agent *dest* that precedes the PassMsg$_{Ctl}$(*dest*) move executed when *src* and *dest* are synchronized.
- By the SignalSema(*insema*(*dest*)) move of StartSync&Wait$_{Sema}$(*dest*) *src* is made ready to be scheduled to PassMsg$_{Sema}$(*src*, *dest*).

We call message exchange triple moves in the two runs corresponding to each other if they have the same name and concern the same parameters among $(m, src, dest)$. Similarly we speak about correspondence of message exchange triples. Pairs of corresponding MsgPass$_{Ctl}$ and MsgPass$_{Sema}$ runs are those runs which are started in equivalent corresponding initial states and perform corresponding message exchange triple moves in the same order. Since single moves correspond to each other, the segments of computation of interest to be compared in corresponding runs consist only of the given single moves. The locations of interest to be compared are *inbox*(*dest*) and *outbox*(*dest*), the ones updated by RecordForTransfer resp. DeliverMsg moves in the two states of interest, the same in both machines. The equivalence is here simply the identity of the values of these locations in the corresponding states of interest.

By the ASM refinement framework defined in [2], these definitions turn the following sentence into a mathematically precise statement.

Theorem 4. MsgPass$_{Sema}$ *is a correct refinement of* MsgPass$_{Ctl}$.

Proof. One has to show that given corresponding MsgPass$_{Sema}$, MsgPass$_{Ctl}$ runs, for each message exchange triple move in the MsgPass$_{Sema}$ run, one can find a corresponding message exchange triple move in the MsgPass$_{Ctl}$ run such that the locations of interest in the corresponding states of interest are equivalent. This follows by an induction on runs and the number of message exchange triple moves. The basis of the induction is guaranteed by the stipulation that the two runs are started in equivalent corresponding states. The induction step follows from the one-to-one relation between (occurences of) corresponding message exchange triple moves described above, using the definitions of the respective machine moves and the fact that the two crucial operations RecordForTransfer and DeliverMsg, which update the locations of interest, are by definition the same in both machines. □

Symmetrically, one can prove the following theorem. The two theorems together constitute the equivalence theorem (also called bisimulation).

Theorem 5. MsgPass_{Ctl} *is a correct refinement of* MsgPass_{Sema}.

7 Concluding Remarks and Future Work

Possible extensions of interest concern stronger machines, as in the example below. Similar examples include the consideration of timeouts, etc. A central question is whether, and how, the equivalence-proof scheme can be extended to work in the multiprocessor case.

Assume we want to use MsgPass_{Ctl} for the case that Receive_{Ctl} is restricted to receive from a set of *expectedsndr*s, which is assumed to be defined when Receive_{Ctl} is called. The first issue to decide is whether the receiver waits until a message from an expected sender shows up (blocking case) or whether in absence of such a message the receiver may receive other messages.

In the non-blocking case the issue is simply a question of priority. Therefore it suffices to refine the function *hd* used in PassMsg to choose the first element from $wtsndr(p) \cap expectedsndr(p)$ if this set is not empty, and the first element from $wtsndr(p)$ otherwise. This extension includes the case that the set *expectedsndr* itself may contain elements of different priorities.

In the blocking case, it suffices to strengthen the Receive_{Ctl} rule by replacing $wtsndr(p)$ with $wtsndr(p) \cap expectedsndr(p)$ in the guards of the two subrules. The notion of *src* being synchronized with *dest* in Theorem 1 has to be refined by restricting *scr* to elements in $expectedsndr(dest)$. This refinement represents a conservative extension of the abstract model (meaning that the effect of corresponding steps in the abstract and the refined machine is the same over the common signature).

Acknowledgement. We thank Gerhard Schellhorn for a discussion of an earlier version of the proof, which led to its simplification.

References

1. M. Altenhofen and E. Börger. Concurrent abstract state machines and ^{+}CAL programs. In A. Corradini and U. Montanari, editors, *WADT 2008*, volume 5486 of *LNCS*, pages 1–17. Springer, 2009.
2. E. Börger. The ASM refinement method. *Formal Aspects of Computing*, 15:237–257, 2003.
3. E. Börger and I. Craig. Modeling an operating system kernel. In V. Diekert, K. Weicker, and N. Weicker, editors, *Informatik als Dialog zwischen Theorie und Anwendung*, pages 199–216. Vieweg+Teubner, Wiesbaden, 2009.
4. E. Börger and R. F. Stärk. *Abstract State Machines. A Method for High-Level System Design and Analysis.* Springer, 2003.
5. I. Craig. *Formal Models of Operating System Kernels.* Springer, 2007.
6. I. Craig. *Formal Refinement for Operating System Kernels.* Springer, 2007.
7. A. S. Tanenbaum. *Modern Operating Systems: Design and Implementation.* Prentice-Hall, 1987.