Communication and Synchronisation Using Interaction Objects

H.B.M. Jonkers

Philips Research Laboratories Eindhoven,
Prof. Holstlaan 4, 5656 AA Eindhoven, The Netherlands
jonkers@natlab.research.philips.com

Abstract. In this paper we introduce a model of process communication and synchronisation, based on the concept of interaction objects. Interaction objects define an abstraction mechanism for concurrent access to data, based on a strict separation of process interaction and data access. Process interaction can be controlled by means of three basic interaction operators that operate on interaction objects. The interaction operators can be used to define various forms of communication and synchronisation including a general form of condition synchronisation. We define the concept of an interaction object and the interaction operators, and give examples of a number of interaction objects. Various aspects of interaction objects are discussed, such as the formal specification and implementation of interaction objects, and the verification of programs that use interaction objects.

1 Introduction

Most operating systems and concurrent programming languages in wide use today are based on the same basic model of process communication and synchronisation. Processes communicate and synchronise by means of intermediate objects such as shared variables, semaphores, monitors, message queues, channels, etc. The operations associated with these intermediate objects can be used concurrently by processes to pass information to each other, or to wait until certain synchronisation conditions are met. The communication and synchronisation mechanisms associated with these objects have been subject of extensive study, leading to a rich theory; see e.g. [1, 22] for comprehensive surveys.

In this paper we introduce a model of process communication and synchronisation, based on a specific type of intermediate object called an *interaction object*. Interaction objects define an abstraction mechanism for concurrent access to data, based on a strict separation of process interaction and data access. Interaction objects have non-blocking and atomic operations only, implying that they can be specified using standard sequential techniques. Interaction between processes, including blocking, is controlled by means of three basic *interaction operators* that operate on interaction objects. In combination with various types of interaction objects, these operators can be used to define several forms of communication and synchronisation, including a general form of condition synchronisation.

As we will argue in the paper, programming with interaction objects and interaction operators helps in minimising and making explicit the interference in concurrent programs, and thereby in reducing their complexity. This argument is supported by practical experience with the CoCoNut software component [16] that has been used in several applications in Philips. CoCoNut provides a full implementation of the programming model discussed in this paper and includes a collection of ready-made interaction objects.

This paper consists of two main parts. In the first part, Section 2, we define the basic concepts such as the interaction point, interaction object and interaction operator, and give an example of an interaction object. In the second part, Section 3, we deal with various aspects of the use of interaction objects, such as how to specify and implement them, and how to verify programs that use interaction objects. In Section 4 we compare interaction objects with related approaches.

The concept of an interaction object, as defined in this paper, originated from work on the SPRINT method [15, 8]. It should not be confused with the concept of an interaction object as used in the context of user interfaces, where it refers to a user-interface widget. In order to distinguish the two, the first type of objects can be characterised as *process* interaction objects and the second type as *computer-human* interaction objects.

2 Basic Concepts

2.1 Objects and Processes

We consider systems consisting of two types of entities: passive *objects* and active *processes*. An object consists of a collection of *variables* and a set of *operations*. The sets of variables of objects are disjoint. In any given state of the system a variable has a certain *value*. The values of all variables of an object define the *state* of the object. The state of an object can only be inspected or changed by means of the operations associated with the object. An operation of an object accesses the variables of that object only. An operation is said to be *enabled* if its precondition is true and *disabled* if its precondition is false.

A process is an autonomous sequential activity that operates on objects, i.e., it may inspect and change the values of variables using the operations associated with the objects. Each process has a *domain* defining the *local variables* of the process. The domains of processes are *disjoint*. Variables not in the domain of any process are referred to as *global variables*.

The operations of objects are divided into two classes: *access operations* and *interaction operations*. An access operation of an object X is an operation that, when called by a process P, will access variables of X in the domain of P only. Since process domains are disjoint, processes can never interfere by calling access operations. An interaction operation of an object X is an operation that, when called by a process P, may access variables of X outside the domain of P, i.e., global variables or variables in the domains of other processes.

Objects and processes partition the set of variables orthogonally, as illustrated in Figure 1. In this and other figures, objects are represented by rectangles, processes by ellipses, variables by small rounded rectangles and operations by arrows. The variables associated with an object may be contained in different processes. For example, the variables v_1 and v_3 of object X are contained in the domains of processes P and Q, respectively. Likewise, the variables associated with a process may be part of different objects. For example, the variables v_3 and v_5 from the domain of Q are contained in the objects X and Y, respectively. There may also be variables that are not contained in any process, such as the global variable v_2 of X, or in any object, such as the internal variable v_4 of Q.

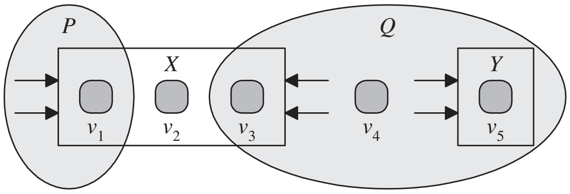

Fig. 1. Objects and Processes

2.2 Interaction Points

In the model defined above, all interaction between processes occurs by means of interaction operations of objects. Most existing communication and synchronisation mechanisms such as monitors, channels, pipes, etc., can be viewed as objects with interaction operations. For example, a message queue, as supported by most operating systems, can be seen as an object containing one global variable (a message buffer) and two interaction operations: *send* and *receive* (see Figure 2).

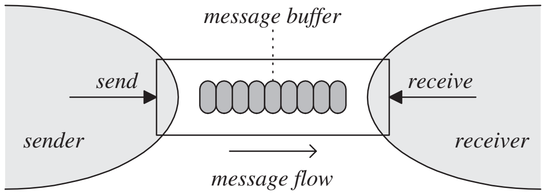

Fig. 2. Message Queue

We define an *interaction point* as a point in the code or execution of a process where the process interacts with other processes, i.e., where it potentially *influences* the behaviour of other processes (*outgoing* interaction point) or *is* potentially *influenced by* other processes (*incoming* interaction point). We use the word "potentially" because the influence of an action of one process on another may be indirect or may depend on dynamic conditions. For example, calling a *send* operation of a message queue introduces an outgoing interaction point in the sender process. The receiver process will only be influenced by this when it calls the *receive* operation of the message queue, introducing an incoming interaction point in the receiver process. There may even be no influence at all if the receiver process chooses to ignore the message queue.

A *synchronisation point* is a special type of incoming interaction point where the *progress* of a process is influenced by other processes, i.e., where the process may *block*. Calling the *receive* operation of a message queue will typically introduce a synchronisation point: if the message queue is empty, the receiver will block until a message is put into the queue by the sender. If the message queue is bounded, a *send* operation could also introduce a synchronisation point in addition to an outgoing interaction point: if the message queue is full, the sender will block until the receiver removes a message from the queue.

The complexity of concurrent programs is closely related to the number of interaction points contained in them. Even a relatively small number of interaction points can give rise to a very large number of interleavings of process actions, making it hard to establish the correctness of such programs. It is therefore important to keep the number of interaction points as small as possible, i.e., to achieve as much decoupling of processes as possible. The mechanisms referred to above only support this to a certain extent, since most operations of monitors, channels, etc., contain interaction points. This is even true for those operations that read information only. For example, an operation of a message queue that allows the receiver to determine the number of messages in the queue introduces an incoming interaction point in the receiver, since the number of messages in the queue may be influenced by the sender.

The model introduced in this paper takes a rather drastic approach in curtailing the number of interaction points. It limits process interaction to three general *interaction operators* which can be used to insert explicit interaction points in the process code. Process code not containing any of these interaction operators will not contain interaction points. The key to this approach is the concept of an interaction object, as defined in the next section.

2.3 Interaction Objects

An interaction object X is an object satisfying the following requirements:

1. All operations of X are non-blocking and atomic.
2. The variables of X consist of the following disjoint subsets per process P:
   (a) $in_{X,P}$: the *input variables* of X for P, which are local variables of P.
   (b) $iav_{X,P}$: the *interaction variables* of X for P, which are global variables.
   (c) $out_{X,P}$: the *output variables* of X for P, which are local variables of P.
3. X has exactly two interaction operations:
   (a) $commit_X$, which has no parameters. The call of $commit_X$ from a process P will be denoted as $commit_{X,P}$. It accesses variables in $in_{X,P} \cup iav_{X,P}$ only, modifies variables in $iav_{X,P}$ only, and disables itself.

(b) $sync_X$, which has a process as its parameter. The call of $sync_X$ from a process P with parameter Q will be denoted as $sync_{X,P,Q}$. It accesses variables in $iav_{X,Q} \cup out_{X,P}$ only, modifies variables in $out_{X,P}$ only, and disables itself.

The *commit* and *sync* operations of interaction objects will be represented in diagrams by curved arrows as indicated in Figure 3.

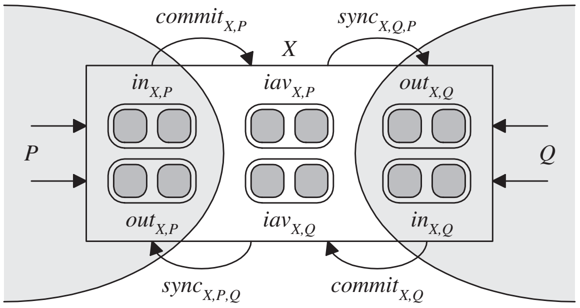

Fig. 3. Commit and Sync Operations

The following remarks can be made about the requirements listed above:

Item 1: This requirement implies that operations of interaction objects do not contain synchronisation points. It also implies that interaction objects can be fully specified using classical pre- and post-condition techniques.

Item 2: Input and output is seen from the point of view of interaction objects rather than processes. That is, the input and output variables provide input to and output from an interaction object. The interaction variables are the intermediaries between input and output variables.

Item 3a: $commit_{X,P}$ can be seen as an action that "commits" a local state change of P by making its effect visible in the interaction variables $iav_{X,P}$, thus introducing an *outgoing* interaction point in P. The self-disabling requirement reflects the fact that there is no sense in committing a local state change twice if the local state has not changed. The only way $commit_{X,P}$ can be enabled after being disabled is by calls of access operations of X by P, modifying the variables in $in_{X,P}$.

Item 3b: $sync_{X,P,Q}$ can be seen as an action that "synchronises" the local state of P with the state of the interaction variables $iav_{X,Q}$, thus introducing an *incoming* interaction point in P. The self-disabling requirement reflects the fact that there is no sense in synchronising twice to a global state change if the global state has not changed. The only way $sync_{X,P,Q}$ can be enabled after being disabled is by calls of $commit_{X,Q}$ modifying the variables in $iav_{X,Q}$.

In the remainder of this paper we will use the definition of an interaction object in a somewhat more liberal way. An object is an interaction object if its representation in terms of variables can be turned into a behaviourally equivalent representation satisfying the above requirements. This approach is justified because objects are fully encapsulated: they can be accessed by means of operations only. It does not matter which variables are used to represent the object, as long as the external behaviour of the object remains the same. This freedom can often be used to write more natural specifications of interaction objects. The disadvantage is that we may have to prove behavioural equivalence of specifications; we will use standard data transformation techniques to do this (see Section 3.2).

We will assume, from now on, that processes interact by means of interaction objects only. Other types of objects are not relevant in the context of this discussion and will be ignored.

2.4 Example of an Interaction Object

As a simple example of interaction objects we consider *observers*. An observer is an interaction object that allows one process, the *reader*, to observe the value of a variable that is changed concurrently by another process, the *writer*. An observer is similar to a shared variable except that write and read operations are "cached", with the cache being controlled by *commit*s and *sync*s. Both the writer and the reader have a cached copy of the shared variable. The writer writes to its own cached copy of the variable. These changes will not become externally visible until the writer performs a *commit* on the observer. The reader reads from its cached copy and will not see any changes until it performs a *sync* on the observer.

Conceptually, an observer X contains three variables: an input variable wv, an interaction variable iv, and an output variable rv. More precisely:

$$\begin{aligned} in_{X,P} &= \textbf{if } P = writer \textbf{ then } \{wv\} \textbf{ else } \emptyset \\ iav_{X,P} &= \textbf{if } P = writer \textbf{ then } \{iv\} \textbf{ else } \emptyset \\ out_{X,P} &= \textbf{if } P = reader \textbf{ then } \{rv\} \textbf{ else } \emptyset \end{aligned}$$

These variables are depicted in Figure 4. In diagrams of interaction objects such as Figure 4, we label the arrows representing $commit_{X,P}$ and $sync_{X,P,Q}$ actions without indicating the X, P and Q, since that information follows from the position of the arrows. Furthermore, we omit arrows representing *commit*s and *sync*s that are always disabled, such as $commit_{X,R}$ and $sync_{X,W,R}$ where $W = writer$ and $R = reader$.

We will specify interaction objects in a small subset of Z [24]. The model of an interaction object is defined by a Z schema containing the processes and variables associated with the interaction object (see the *Observer* schema below). The initial state is defined by the *init* schema and the operations by schemas with the same name as the operation (see the *write*, *read*, *commit* and *sync* schemas below). In the specifications we treat the process P calling an operation as a parameter of that operation (indicated by "P" rather than "$P?$"), though in actual code this parameter will normally be implicit. The first line in the axiom

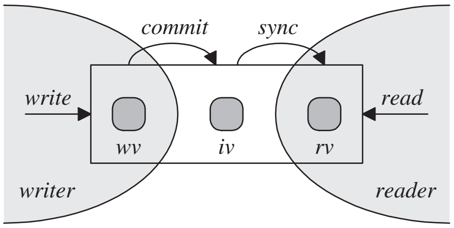

Fig. 4. Model of an Observer

part of an operation schema can be read as its pre-condition and the rest as its post-condition.

The full specification of observers is given below, where *Process* and *Value* (the values of observers) are given sets.

Observer

$writer, reader : Process$
$wv, iv, rv : Value$

---

init

$Observer$
$v? : Value$

$wv = iv = rv = v?$

ΔObserver

$Observer$
$Observer'$

$writer' = writer \wedge reader' = reader$

---

write

$\Delta Observer$
$P : Process$
$v? : Value$

$P = writer$
$wv' = v? \wedge iv' = iv \wedge rv' = rv$

---

read

$\Xi Observer$
$P : Process$
$v! : Value$

$P = reader$
$v! = rv$

$commit$

$\Delta Observer$
$P : Process$

$P = writer \wedge wv \neq iv$
$wv' = wv \wedge iv' = wv \wedge rv' = rv$

$sync$

$\Delta Observer$
$P, Q : Process$

$P = reader \wedge Q = writer \wedge rv \neq iv$
$wv' = wv \wedge iv' = iv \wedge rv' = iv$

We can easily verify that observers, as specified above, meet all requirements of interaction objects. It is also quite simple to generalise this specification for observers with multiple readers.

2.5 Basic Interaction Operators

When dealing with configurations of processes and interaction objects the pictorial representations used so far are somewhat clumsy. To represent such configurations, we introduce an additional graphical notation which is similar to the one used in [9]. As illustrated in Figure 5, processes are represented by parallellograms and interaction objects by rectangles. A line connecting a process P to

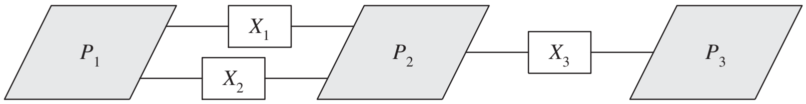

Fig. 5. Configuration of Processes and Interaction Objects

an interaction object X indicates that P is *attached to* X, i.e., that there is some form of interaction between P and X, by means of *commit*s, *sync*s, or both. If P is not attached to X, $commit_{X,P}$ and $sync_{X,P,Q}$ are always disabled and can therefore be ignored. The rectangles represent general interaction objects; we will introduce special symbols for particular types of interaction objects. The symbol for an observer, with multiple readers, is shown in Figure 6.

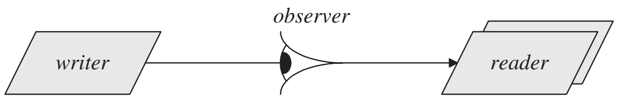

Fig. 6. Observer Symbol

Programming with interaction objects would be a nuisance if each interaction had to be programmed in terms of calls of individual *commit* and *sync* operations. For example, all interactions in process P_2, in Figure 5, would have to be programmed in terms of the *commit*s and *sync*s of interaction objects X_1, X_2 and X_3. Instead of this, we will define three basic *interaction operators* that provide a general way to control interaction. In practice, one can even make it impossible to access the individual *commit* and *sync* operations other than by means of these interaction operators (as done in CoCoNut, see Section 3.4).

The first two interaction operators, called **commit** and **sync**, are more or less obvious. When used in a process P, **commit** performs all enabled *commit*s and **sync** performs all enabled *sync*s on the interaction objects that P is attached to. Use of **commit** and **sync** in a process P will be denoted as $\mathbf{commit}_P$ and $\mathbf{sync}_P$, respectively. So, when programming process P_2 in Figure 5, the programmer could perform a number of access operations on the interaction objects X_1, X_2 and X_3, and then use **commit** to make their effect visible to P_1 and P_3, or use **sync** to synchronise with any changes made by P_1 and P_3 to the global variables in X_1, X_2 and X_3.

Statically, i.e., in the process code, $\mathbf{commit}_P$ introduces a single outgoing interaction point and $\mathbf{sync}_P$ introduces a single incoming interaction point in P. Dynamically, both $\mathbf{commit}_P$ and $\mathbf{sync}_P$ may amount to the execution of a sequence of individual atomic actions, and they may thereby introduce multiple interaction points in P. For $\mathbf{sync}_P$, this is more or less natural since the individual *sync*s executed by $\mathbf{sync}_P$ are enabled asynchronously by the *commit*s of other processes. Even while executing $\mathbf{sync}_P$, new *sync*s may be enabled. In order to avoid potential unboundedness in the execution of $\mathbf{sync}_P$, we will assume that *sync*s that are enabled during the execution of $\mathbf{sync}_P$ are ignored; they will be executed in the next call of $\mathbf{sync}_P$.

For $\mathbf{commit}_P$, the situation is different. The set of *commit*s that are executed by $\mathbf{commit}_P$ is fully predictable since these *commit*s have been enabled by P itself. Moreover, the order of execution of these *commit*s is irrelevant since each $commit_{X,P}$ operates on a disjoint set of variables: it reads from $in_{X,P} \cup iav_{X,P}$ and writes to $iav_{X,P}$. We will therefore assume that the *commit*s executed by $\mathbf{commit}_P$ are executed in a single atomic action. This can be implemented in an efficient way, as discussed in Section 3.4.

Both **commit** and **sync** are non-blocking operators. In order to provide a general interaction mechanism, we also need some way to block a process, i.e., some way to introduce a synchronisation point in the code of a process. The third interaction operator, called **wait**, does just that. When used in a process P it will block P until at least one $sync_{X,P,Q}$ is enabled for some X and Q. Use of **wait** in a process P will be denoted as $\mathbf{wait}_P$.

We will say that a process P is *in sync* with an interaction object X if $sync_{X,P,Q}$ is disabled for all Q. Otherwise we will say that P is *out of sync* with X. We will say that a process is *in sync* if it is in sync with all interaction objects it is attached to. Otherwise we will say that it is *out of sync*. So, the effect of $\mathbf{wait}_P$ can also be described as “block P until it is out of sync”.

The definitions of the three interaction operators are summarised below:

commit$_P$: Perform all enabled *commit*s of the interaction objects to which P is attached in *one* atomic action.
sync$_P$: Perform all enabled *sync*s of the interaction objects to which P is attached, where each *sync* is an individual atomic action.
wait$_P$: Block until at least one *sync* of an interaction object attached to P is enabled.

2.6 Composite Interaction Operators

The three interaction operators defined above constitute a complete set in the sense that, in combination with the proper interaction objects, they can be used to define most of the standard communication and synchronisation mechanisms. Rather than defining these mechanisms directly in terms of the basic interaction operators, it is useful to use a few composite interaction operators. In defining these operators, and also in the example process code, we will use C(++)-like macro definitions, control structures and parameter passing conventions. We will use ":=" rather than "=" as the assignment operator, and "=" rather than "==" as the equality operator.

The first composite interaction operator is the **next** operator defined by:

#define next { **commit**; **sync**; }

It is typically used after a sequence of access operations $S_1; \ldots; S_n$ in a process to make the effect of the operations globally visible and, at the same time, to synchronise with global state changes of other processes. Note that the sequence of actions:

S_1; . . . ; S_n; **commit**;

constitutes a single atomic state transition. The name of the **next** operator is inspired by the similar **nextstate** operator of SDL [6]. That is, we can read it as "finish the current atomic state transition, synchronise, and start with the next".

The second composite interaction operator is the parameterised **await** operator defined by:

#define await(C) { **next**; **while**($\neg$ C){ **wait**; **sync**; } }

await(C) will make a process wait until condition C becomes true. Any condition C is allowed, provided that only local variables of the process and side-effect free access operations of objects are used in C. (We will weaken this restriction later on.) The **await** operator provides what is known as *condition synchronisation* and can be seen as a restricted, though still fairly general form of the *await statement* [1]. Unlike the general await statement, it can be implemented very efficiently. All that is required is efficient implementations of the **commit**, **sync** and **wait** operators (see Section 3.4).

When executing **await**(C), a process P will perform a **commit** and then repeatedly perform a **sync**, evaluate C, and call **wait** as long as C returns *false*. Calling **wait** when C returns *false* is safe because C depends only on the values of local variables in P. C can only become true due to the execution of *sync*s that modify the local variables of P. If no *sync* is enabled, P can safely block in **wait** until a *sync* is enabled. Note that C may be true temporarily during the execution of a **sync**, while still being false immediately after the execution of **sync**. However, if C finally becomes true it will still be true at the beginning of the statement immediately following **await**(C). So, irrespective of the condition C, the following **assert** statement will never fail:

await(C); **assert**(C);

In the examples discussed in the remainder of this paper, we will only use **next** and **await** rather than **commit**, **sync** and **wait**. Strictly speaking, we could even restrict ourselves to using **await** since **next** is equivalent to **await**(*true*).

3 Using Interaction Objects

3.1 Using the Interaction Operators

In order to demonstrate the use of the interaction operators we consider a simple alarm control system (inspired by an example from [5]). The system should measure the temperature and humidity in a room every 50 and 100 ms, respectively, and ring a bell while the temperature and humidity are in the unsafe range. The predicate *safe*(t, h) indicates whether the combination of temperature t and humidity h is in the safe range. We use three processes: *measure_temperature* and *measure_humidity* to measure temperature and humidity, respectively, and *alarm_control* to control the alarm bell. The *alarm_control* process can read the temperature and humidity by means of two observers *temp* and *humi*, that are written to by the two measurement processes (see Figure 7).

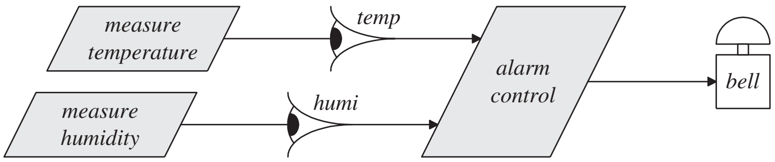

Fig. 7. Temperature Humidity Alarm System

The code of the three processes is given below, where *sleep*(n) delays a process for n milliseconds, and *measure_temp*($\&t$) and *measure_humi*($\&h$) perform measurements of temperature and humidity and assign the measured values to the variables t and h, respectively. The bell is represented by the variable *bell* with two possible values: *on* and *off*. Initially the temperature and humidity are in the safe range and the bell is off.

```
#define T   temp.read()
#define H   humi.read()
```

```
measure_temperature:
  Temperature t;
  while(true)
  { sleep(50);
    measure_temp(&t);
    temp.write(t);
    next;
  }
```

```
measure_humidity:
  Humidity h;
  while(true)
  { sleep(100);
    measure_humi(&h);
    humi.write(h);
    next;
  }
```

```
alarm_control:
  while(true)
  { await(¬ safe(T, H));
    bell := on;
    await(safe(T, H));
    bell := off;
  }
```

Informally we can argue the correctness of this program as follows. At the two synchronisation points in the *alarm_control* process, i.e., at the two occurrences of **wait** in the **await** statements, the following invariant holds:

$$(\neg\ \mathit{safe}(T,H) \wedge \mathit{bell} = \mathit{on}) \vee (\mathit{safe}(T,H) \wedge \mathit{bell} = \mathit{off})$$

When blocked at one of its synchronisation points, the *alarm_control* process is in sync with the *temp* and *humi* observers and hence T and H are equal to the measured temperature and humidity, respectively. So, as long as the *alarm_control* process is blocked, the system has the desired properties. As soon as it gets out of sync because of changes of the measured temperature or humidity, the process will de-block, re-synchronise and block again at either the same or the next synchronisation point, thereby restoring the invariant. The blocking will occur because *sync*s disable themselves and the two conditions in the **await** operators are mutually exclusive. The assumption in all of this is, of course, that the execution of the code in the body of the while loop of the *alarm_control* process takes substantially less time than the average time between two successive "out of sync" events in the observers *temp* and *humi* (which is always $\geq$ 33 ms). Note that the **next** operator in the measurement processes could be replaced by **commit**.

3.2 Specifying Interaction Objects

In Section 2.4 we have already seen a specification of a simple interaction object. Some of the more subtle details of specifying interaction objects will be illustrated here using the example of a *mailbox.* A mailbox is similar to a message queue as discussed earlier (see Figure 2) in that it provides a way to pass messages from a sender process to a receiver process asynchronously. The difference is that mailboxes satisfy the requirements of interaction objects, as reflected by the model of a mailbox in Figure 8. The *send* and *receive* operations operate on the local message queues *sm* and *rm* in the domains of the sender and receiver processes, respectively. Interaction between sender and receiver occurs through the global message queue *im* using *commit*s and *sync*s.

We will allow mailboxes to be used simultaneously by multiple senders, leading to the generalised mailbox model indicated in Figure 9. In this model, each

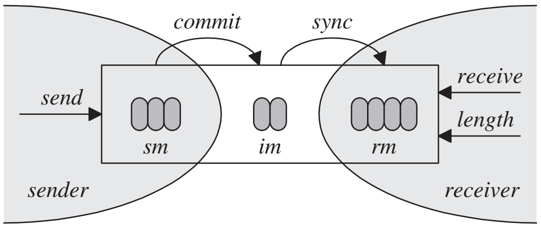

Fig. 8. Model of a Mailbox with a Single Sender

sender process P has its own local and global message queues. This is represented in the formal specification of a mailbox below, by means of the functions sm and im that map a sender process P to its associated message queues $sm(P)$ and $im(P)$. In the specification we use the types *Process* and *Message* as given sets.

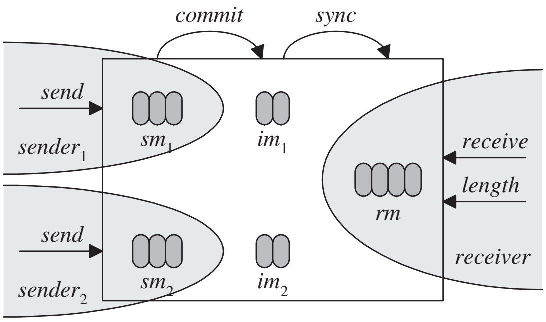

Fig. 9. Model of a Mailbox with Multiple Senders

Mailbox

$senders : \mathbb{F}\, Process$

$receiver : Process$

$sm, im : Process \pfun \operatorname{seq} Message$

$rm : \operatorname{seq} Message$

$\operatorname{dom} sm = \operatorname{dom} im = senders$

---

init

$Mailbox$

$(\forall\, P : senders \bullet sm(P) = im(P) = \langle\rangle) \land rm = \langle\rangle$

ΔMailbox

$Mailbox$

$Mailbox'$

$senders' = senders \land receiver' = receiver$

send

$$\begin{array}{l} \Delta Mailbox \\ P : Process \\ m? : Message \\ \hline P \in senders \\ sm' = sm \oplus \{P \mapsto \langle m? \rangle \frown sm(P)\} \land im' = im \land rm' = rm \end{array}$$

receive

$$\begin{array}{l} \Delta Mailbox \\ P : Process \\ m! : Message \\ \hline P = receiver \land rm \neq \langle\rangle \\ sm' = sm \land im' = im \land rm' \frown \langle m! \rangle = rm \end{array}$$

length

$$\begin{array}{l} \Xi Mailbox \\ P : Process \\ n! : \mathbb{N} \\ \hline P = receiver \\ n! = \# rm \end{array}$$

commit

$$\begin{array}{l} \Delta Mailbox \\ P : Process \\ \hline P \in senders \land sm(P) \neq \langle\rangle \\ sm' = sm \oplus \{P \mapsto \langle\rangle\} \land im' = im \oplus \{P \mapsto sm(P) \frown im(P)\} \land rm' = rm \end{array}$$

sync

$$\begin{array}{l} \Delta Mailbox \\ P, Q : Process \\ \hline P = receiver \land Q \in senders \land im(Q) \neq \langle\rangle \\ sm' = sm \land im' = im \oplus \{Q \mapsto \langle\rangle\} \land rm' = im(Q) \frown rm \end{array}$$

In contrast with observers, it is not immediately clear that mailboxes satisfy all requirements of interaction objects. The obvious way to define the sets of input, interaction and output variables of a mailbox X is as follows:

$$\begin{array}{lll} in_{X,P} & = \textbf{if } P \in senders \textbf{ then } \{sm(P)\} & \textbf{else } \emptyset \\ iav_{X,P} & = \textbf{if } P \in senders \textbf{ then } \{im(P)\} & \textbf{else } \emptyset \\ out_{X,P} & = \textbf{if } P = receiver \textbf{ then } \{rm\} & \textbf{else } \emptyset \end{array}$$

In this definition we took the liberty of interpreting $sm(P)$ and $im(P)$ as individual variables rather than as references to parts of the variables sm and im.

The reason why the above definition does not work is that $commit_{X,P}$ modifies the variables in $in_{X,P}$, and $sync_{X,P,Q}$ modifies the variables in $iav_{X,Q}$. This is not allowed according to the definition of interaction objects. It can be remedied by transforming the above specification into another behaviourally equivalent specification using a different set of variables, as sketched briefly below. We use the method of "adding and removing variables" as described in [14, 20]. Step 1 is to augment the model of a mailbox with three new variables: the input variable st, the interaction variable it and the output variable rt.

```
── Mailbox2 ──────────────────────────────
│ Mailbox
│ st, it, rt : Process ⇸ seq Message
├──────────
│ dom st = dom it = dom rt = senders
└─────────────────────────────────────────
```

The idea is to make $st(P)$ equal to the trace of *all* messages sent to the mailbox by process P, with *commit* making copies of $st(P)$ in $it(P)$ and *sync* making copies of $it(P)$ in $rt(P)$. Step 2 is to augment the operation specifications accordingly, and prove the following mailbox invariant:

$$\forall P : senders \bullet st(P) = sm(P) \frown it(P) \wedge it(P) = im(P) \frown rt(P)$$

This invariant allows sm and im to be expressed entirely in terms of st, it and rt. Step 3 is to eliminate all applied occurrences of sm and im by means of replacements. Step 4 is to remove the now redundant defining occurrences of sm and im from the model as well. The new definition of a mailbox, thus obtained, satisfies all interaction object requirements, where:

$$\begin{aligned}
in_{X,P} &= \textbf{if } P \in senders \textbf{ then } \{st(P)\} \textbf{ else } \emptyset \\
iav_{X,P} &= \textbf{if } P \in senders \textbf{ then } \{it(P)\} \textbf{ else } \emptyset \\
out_{X,P} &= \textbf{if } P = receiver \textbf{ then } \{rt(P), rm\} \textbf{ else } \emptyset
\end{aligned}$$

3.3 Communicating Using Interaction Objects

We will use mailboxes to illustrate how interaction objects can be used to communicate between processes. The symbol for a mailbox is shown in Figure 10. A

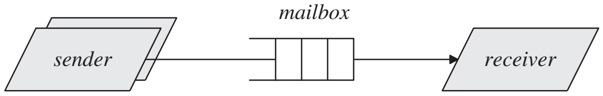

Fig. 10. Mailbox Symbol

sender process can send a message x to a mailbox mbx like this:

$mbx.send(x)$; **next**;

The receiver process can receive the message in a variable m like this:

$\textbf{await}(mbx.length > 0);\ mbx.receive(\&m);$

Here we have assumed that the output parameter $m!$, in the specification of the *receive* operation, has been implemented as a reference parameter. Compared with the normal way of receiving a message from a message queue using a single blocking call, the above may seem somewhat clumsy. This can be remedied by implementing the precondition of *receive* as a boolean return value. If the precondition is not valid, the operation returns *false* and has no side effect, allowing a message to be received like this:

$\textbf{await}(mbx.receive(\&m));$

This approach requires the use of assertions with side effects. A safe rule to contain the negative effects of this is to require that the condition C in $\textbf{await}(C)$ has no effect when it returns *false*. This implies that, in the execution of the await statement, the side effect will occur only once, i.e., when the assertion C returns *true*.

Using **or** and **and** as conditional versions of the logical operators $\vee$ and $\wedge$ (similar to the || and && operators in C), several standard communication constructs can be defined directly in terms of the **await** construct. A process that has to receive data from one of two mailboxes $mbx1$ and $mbx2$ can do so by means of the following construct:

$\textbf{await}(mbx1.receive(\&m)\ \textbf{or}\ mbx2.receive(\&m));$

By adding guards and actions, this can be extended to a general guarded input statement:

```
await(  ( GUARD1 and mbx1.receive(&m) and ACTION1 )
     or ( GUARD2 and mbx2.receive(&m) and ACTION2 )
     );
```

The only assumption we make here is that the guards have no side effect and that the actions return *true*. By replacing mailboxes by interaction objects supporting synchronous communication (such as CoCoNut channels, see Section 3.4), we can even implement a rendezvous mechanism without introducing any additional language constructs. The above guarded input statement then corresponds to an Ada select statement.

In order to demonstrate what happens if multiple processes are concurrently sending data to the same mailbox, consider a simple configuration of two producers and one consumer, as depicted in Figure 11. Each producer repeatedly sends a pair of consecutive numbers to the mailbox in a single atomic action. The consumer also receives the natural numbers in pairs, as described by the code of producers and consumer below.

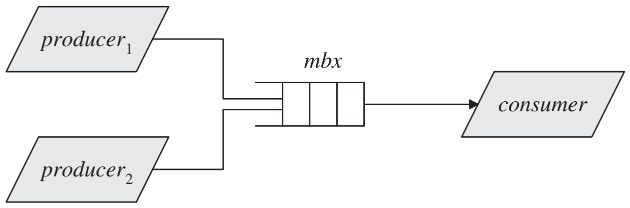

Fig. 11. Producers and Consumer

```
producer_i (i = 1, 2):
  int n := 0;
  while(true)
  { mbx.send(n);
    mbx.send(n + 1);
    next;
    n := n + 1;
  }
```

```
consumer:
  int n1, n2;
  while(true)
  { await(mbx.length ≥ 2);
    mbx.receive(&n1);
    mbx.receive(&n2);
    assert(n2 = n1 + 1);
  }
```

It is easy to infer from the specification of mailboxes that, despite the fact that the individual *send* operations of the producers are interleaved, the consumer will always receive pairs of two consecutive numbers. Hence, the **assert** statement in the consumer code will never fail. Though not advisable from the point of view of defensive programming, we could even replace the **await** statement in the consumer code by:

await($mbx.length > 0$);

3.4 Implementing Interaction Objects

In discussing the implementation of interaction objects, we will restrict ourselves to the situation that all processes execute in the same address space, i.e., that processes are *threads*. This situation is typical for embedded systems in the context of which the concept of an interaction object originated. Embedded systems often use real-time kernels to provide basic thread management, synchronisation and communication. Implementing interaction objects in distributed systems is a separate story that will not be discussed here.

The implementation problem of interaction objects is essentially the problem of implementing the three interaction operators **commit**, **sync**, and **wait**. First consider the problem of guaranteeing the atomicity of **commit** and the individual *sync*s executed by **sync**. This problem can be solved by associating a mutual exclusion mechanism with each process P. When executing **commit**, P uses this mechanism to disable access to all of its associated interaction variables, i.e., the variables in the sets $iav_{X,P}$ for all X attached to P (see Figure 3). Another process Q executing $sync_{X,Q,P}$ uses the same mechanism to disable access to the variables $iav_{X,P}$.

The mutual exclusion mechanism can be chosen per process. Mutexes and pre-emption disabling are typical choices, while interrupt disabling can be used for interrupt service routines (ISRs). Note that, insofar as interaction objects are concerned, we can treat ISRs as normal processes, even though they are dealt with differently at the operating system level. For non-ISR processes, pre-emption disabling is often a good choice because *commit*s and *sync*s are normally very short actions. For example, in a typical mailbox implementation, they amount to append operations on linked lists.

The problem of how to determine which *commit*s and *sync*s should be executed by **commit** or **sync** can be solved by introducing flags $c_{X,P}$ and $s_{X,P,Q}$ for each $commit_{X,P}$ and $sync_{X,P,Q}$, respectively, while maintaining the following invariant:

$commit_{X,P}$ is enabled $\Rightarrow$ $c_{X,P}$ is set
$sync_{X,P,Q}$ is enabled $\Rightarrow$ $s_{X,P,Q}$ is set

According to the definition of an interaction object, only the access operations of X can enable $commit_{X,P}$, so access operations of X should set $c_{X,P}$ when called by P. Likewise, only $commit_{X,Q}$ can enable $sync_{X,P,Q}$, so $commit_{X,Q}$ should set $s_{X,P,Q}$ when called by Q. **commit** and **sync** can use the flags to determine which *commit*s or *sync*s to execute. They should clear a flag immediately before the execution of the corresponding *commit* or *sync*. This cannot violate the invariant because of the self-disabling property of *commit*s and *sync*s. **wait** can use the $s_{X,P,Q}$ flags to determine whether a process should be blocked or de-blocked, without affecting the flags themselves. Note that there could be situations where a flag is set while the corresponding *commit* or *sync* is disabled. This can do no harm provided that disabled *commit*s and *sync*s have no effect when executed. Note also that in a real implementation, it is more efficient to use lists of function pointers rather than flags (similar to *active messages* [10]).

The CoCoNut component [16], developed by Philips Research, provides a full implementation of interaction objects along the lines sketched above. CoCoNut ("Control Component in a Nutshell") is a scalable, platform-independent software component defining an operating system abstraction on top of which applications using interaction objects can be developed. It provides implementations of the interaction operators and a number of interaction objects such as *observers* (see Section 2.4), *mailboxes* (see Section 3.2), *events* (asynchronous push buttons), *channels* (one-slot buffers supporting synchronous communication), *buffers* (multi-slot bounded communication buffers), *locks* (mutexes according to the interaction object model), and *timers* (programmable objects supporting timed actions). In addition to this, CoCoNut supports dynamic creation of interaction objects, dynamic attachment of processes to interaction objects, and facilities for the construction of custom-made interaction objects.

Another feature of CoCoNut is the support for organising applications as collections of *communicating state machines*, with the ability to allocate state machines to processes at compile time. This allows systematic *task inversion* [9] which is important in resource-constrained embedded systems that can only afford a small number of processes. In the extreme case that all state machines are allocated to the same process, a single-process version of CoCoNut can be used requiring no real-time kernel. Kernel-based applications of CoCoNut should nevertheless be preferred since they can take full, real-time advantage of the loose process coupling provided by interaction objects, using the pre-emptive priority-based scheduling support of the kernel. In practice, CoCoNut has been used both with and without real-time kernels.

3.5 Verifying Programs That Use Interaction Objects

Programs that use interaction objects can be verified using standard verification techniques for concurrent programs such as [7, 18, 17]. The verification task is simplified because these programs, if properly designed, can be dissected into relatively large pieces of atomic code. We exemplify this by sketching how a program that uses interaction objects can be mapped to a *transition system*, thus enabling the use of standard techniques for proving properties of transition systems (see [23], for example).

We will assume that programs use **await** and **next** as interaction operators only, and that all conditions in **await** statements are free of side effects. With the **await** and **next** macros expanded, the code of each process consists of atomic sequences of statements ending in **commit**, where each **commit** is followed by one or more executions of **sync**. For example, the code of the *alarm_control* process from the temperature/humidity alarm system in Section 3.1 expands, with some rewriting, to:

```
commit;
while(true)
{ sync; while(safe(T, H)){ wait; sync; }; bell := on; commit;
  sync; while(¬ safe(T, H)){ wait; sync; }; bell := off; commit;
}
```

Using an auxiliary control variable to model the program counter, code like this can be mapped in a standard way to a transition system. In this case we do not even need the auxiliary variable because the *bell* variable, whose initial value is *false*, can be used as the control variable. In programs that use interaction objects, the transitions will be of two types: *outgoing* transitions (the ones ending in **commit**) and *incoming* transitions (the individual *sync*s contained in **sync**). We can ignore the occurrences of **wait** because they are only there to keep the program from useless busy waiting. Since the **sync** operator is used between *all* outgoing transitions, we decouple the incoming transitions from the control flow and treat the *sync*s as autonomous transitions in the transition system. For example, the *alarm_control* process breaks down into the following transitions:

$bell = off \wedge \neg\, safe(T, H) \rightarrow bell := on;\ \mathbf{commit}_{P_A};$
$bell = on \wedge safe(T, H) \rightarrow bell := off;\ \mathbf{commit}_{P_A};$
$enabled(sync_{temp,P_A,P_T}) \rightarrow sync_{temp,P_A,P_T};$
$enabled(sync_{humi,P_A,P_H}) \rightarrow sync_{humi,P_A,P_H};$

Here P_A, P_T and P_H represent the *alarm_control*, *measure_temperature* and *measure_humidity* processes, respectively. The part on the left-hand side of an arrow is the *enabling condition* of the transition, and the part on the right-hand side is the atomic *action* of the transition. Using the formal specifications of the interaction objects, the transition rules can be expressed in terms of operations on variables and can be further simplified. For example, the input variables of interaction objects can generally be completely eliminated. In the above case, the occurrences of $\mathbf{commit}_{P_A}$ can even be omitted because they have no effect.

The decoupling of synchronisation actions from the control flow in the above mapping of programs to transition systems implies that we are actually using a more nondeterministic of version of the **sync** operator than the one defined in Section 2.5. Rather than assuming that **sync** executes *all* enabled *sync*s in a process, we assume that it executes *some* enabled *sync*s, where "some" could be anything between none and all. Fairness is assumed in the sense that an enabled *sync* will eventually be executed. This is the natural interpretation of the **sync** operator in a distributed setting. We prefer this interpretation in proving properties of programs, since it leads to simpler transition systems and wider applicability of programs. As a consequence, we should design our programs in such a way that their desired properties can be proven irrespective of the order of execution of the synchronisation actions. Note that if timing is important, as in the temperature/humidity alarm system, time steps can be added as transitions to the transition systems leading to a *timed transition systems* approach [11].

4 Related Work

In this section, we compare our concurrent system model (processes communicating by means of interaction objects) to related approaches. A common aim of our model and more fundamental models, such as transition systems [18, 7] and action systems [2], including language outgrowths such as Seuss [19] and DisCo [13], is to support the use of sequential techniques in dealing with concurrent systems, based on an interleaving model of concurrency. The link with transition systems was already discussed in Section 3.5. In terms of action systems, access operations of objects would be *private actions* and interaction operations would be *joint actions* of processes. The difference is that we restrict ourselves to very special types of joint actions (*commit*s and *sync*s) allowing the definition of general interaction operators that can be inserted in the control flow of a process. Neither transition systems nor action systems consider processes at the level of control flow.

Interaction operators similar to the ones defined in this paper can be found, for example, in the Mianjin language [21] and in SDL [6]. Mianjin is a parallel programming language supporting the concept of global objects and the ability to call methods of global objects asynchronously from different processes. The process owning the global object will not be interfered with until it calls the **poll** operator, which has a similar effect as a **sync** operator in that it executes all pending global method calls and thereby introduces an incoming interaction point in the process. The main differences are that **sync** is more restricted than

poll since it executes synchronisation actions only, and that there is no equivalent of the **commit** operator in Mianjin.

As already noticed in Section 2.6, the **nextstate** operator of SDL is similar to the **next** interaction operator. SDL is a concurrent programming language supporting state machines that communicate asynchronously using "signals". At the beginning of a transition, a state machine can receive signals and during the transition it can send signals. The **nextstate** operator is used to terminate a transition. This can conceptually be seen as "committing" any pending output and "synchronising" to new input, before going to the next transition. The model defined in this paper is more general than the SDL programming model in that it allows state machines to communicate using any type of communication mechanism (interaction object). During a transition, a state machine can perform arbitrary input and output actions (access operations on interaction objects) without the danger of affecting the atomicity of the transition.

Monitors [3, 12, 4] and interaction objects are similar in the sense that both define an abstraction mechanism for concurrent access to data. There is one essential difference: in monitors all interaction between processes accessing the data is controlled from within the monitor. Synchronisation is controlled internally e.g. using condition variables, implying that monitor operations are generally *blocking* operations. In contrast, the operations of an interaction object do not contain interaction points at all *except* for the two interaction operations *commit* and *sync* which are non-blocking. All interaction between processes is controlled *outside* an interaction object by means of three basic interaction operators. This has a number of consequences. First of all, processes can use a general form of condition synchronisation, using the **await** operator, instead of the dedicated condition synchronisation implemented by a monitor. Secondly, processes can define synchronisation conditions that involve multiple interaction objects. With monitors this is not possible because monitor operations may block. Finally, interaction objects are simpler to specify and verify than monitors: they can e.g. be specified completely using pre- and post-condition techniques.

5 Conclusion

We conclude by recapitulating some of the salient features of the model of communication and synchronisation introduced in this paper. The key characteristic of this model is the strict separation of process interaction and data access as reflected in the definition of an interaction object. All process interaction is established by means of three interaction operators and all data access is non-interfering. As a consequence of this separation of concerns, programmers get full control over the interaction points in their programs and can deal with data access using standard sequential techniques. Furthermore, various standard ways of communication and synchronisation can be be modelled directly in terms of combinations of interaction operators and interaction objects. The model can be implemented in an efficient and platform-independent way, as demonstrated by the CoCoNut component [16]. Distributed implementations of interaction objects have not been discussed in this paper and are subject of future work.

References

[1] Andrews, G.R., *Concurrent Programming*, Benjamin/Cummings (1991).

[2] Back, R.J.R., Kurki-Suonio, R., *Distributed Cooperation with Action Systems*, ACM Transactions on Programming Languages and Systems, Vol. 10, 4 (1988), 513–554.

[3] Brinch Hansen, P., *Operating System Principles*, Prentice-Hall (1973).

[4] Buhr, P.A., Fortier, M., Coffin, M.H., *Monitor Classification*, ACM Computing Surveys 27, 1 (1995), 63–107.

[5] Bustard, D., Elder, J., Welsh, J., *Concurrent Program Structures*, Prentice Hall (1988).

[6] CCITT Recommendation Z.100: *Specification and Description Language SDL*, Blue Book, Volume X.1–X.5, ITU (1988).

[7] Chandy, K.M., Misra, J., *Parallel Program Design*, Addison Wesley (1988).

[8] Feijs, L.M.G., Jonkers, H.B.M., *History, Principles and Application of the SPRINT Method*, Journal of Systems and Software 41 (1998), 199-219.

[9] Gomaa, H., *Software Design Methods for Concurrent and Real-Time Systems*, Addison-Wesley (1993).

[10] von Eicken, T., Culler, D., Goldstein, S.C., Schauser, K.E., *Active Messages: a Mechanism for Integrated Communication and Computation*, in: Proceedings of the 19th International Symposium on Computer Architecture (1992).

[11] Henzinger, T.A., Manna, Z., Pnueli, A., *Timed Transition Systems*. In: Real-Time: Theory in Practice, Lecture Notes in Computer Science, Vol. 600, Springer-Verlag (1992), 226–251.

[12] Hoare, C.A.R., *Monitors: An Operating System Structuring Concept*, Communications of the ACM, Vol. 17, 10 (1974), 549–557.

[13] Järvinen, H.-M., Kurki-Suonio, R., *DisCo specification language: marriage of actions and objects*. In: Proceedings of the 11th International Conference on Distributed Computing Systems, IEEE Computer Society Press (1991), 142–151.

[14] Jonkers, H.B.M., *Abstraction, Specification and Implementation Techniques*, Mathematical Centre Tracts, Vol. 166, Mathematisch Centrum (1983).

[15] Jonkers, H.B.M., *An Overview of the SPRINT Method*. In: Woodcock, J.C.P., Larsen, P.G. (Eds.), Industrial Strength Formal Methods, Lecture Notes in Computer Science, Vol. 670, Springer-Verlag (1993), 403-427.

[16] Jonkers, H.B.M., *Survey of CoCoNut 1.0*, Technical Report RWB-506-ir-96022, Philips Research, Information and Software Technology (1996).

[17] Lamport, L., *The Temporal Logic of Actions*, ACM Transactions on Programming Languages and Systems, Vol. 16, 3 (1994), 872–923.

[18] Manna, Z., Pnueli, A., *The Temporal Logic of Reactive and Concurrent Systems: Specification*, Springer-Verlag (1992).

[19] Misra, J., *An Object Model for Multiprogramming*, Proc. 10th IPPS/SPDP 98 Workshops, Jose Rolim (ed.), Lecture Notes in Computer Science, Vol. 1388, Springer-Verlag (1998), 881–889.

[20] Morgan, C., *Programming from Specifications*, Prentice Hall (1990).

[21] Roe, P., Szyperski, C., *Mianjin is Gardens Point: A Parallel Language Taming Asynchronous Communication*. In: Fourth Australasian Conference on Parallel and Real-Time Systems (PART'97), Springer-Verlag (1997).

[22] Schneider, F.B., *On Concurrent Programming*, Springer-Verlag (1997).

[23] Shankar, A.U., *An Introduction to Assertional Reasoning for Concurrent Systems*, ACM Computing Surveys, Vol. 25, 3 (1993), 225–262.

[24] Spivey, J.M., *The Z Notation: A Reference Manual*, Second Edition, Prentice Hall (1992)