MODELLING CONCURRENT MODULES

Rainer Isle
Klaus-Peter Löhr

Fachbereich Mathematik/Informatik
Universität Bremen
Postfach 330440, D-2800 Bremen 33

Abstract

The variety of module concepts for concurrent systems suggests looking for a unified model that would serve as a basis for specification techniques and languages. Starting from the client/server paradigm, a model for concurrent modules is developed that is able to cope with incomplete service execution due to blocking conditions. The model is shown to be applicable to different kinds of modules known from programming languages. It is generalized to support so-called implementation specification and interconnection of modules using different communication rules.

1. Introduction

If the maturity of a discipline is to be judged by the amount of agreement on its basic issues and notions, then specification of concurrent program modules is still in its infancy. This is in contrast to sequential programming. Although different specification methods and languages exist for the design specification of sequential systems, the basic notion of data abstraction is widely accepted. There are different opinions on how to describe the behaviour of an abstract module (or object), and on the composition of modules. But there is agreement that an abstract data type has to be specified without resort to its possible implementation, in a way that allows both users and implementors to rely on the specification only.

In the sequential case, a module is indeed a "passive object". Thus, a module can be specified by associating certain mappings between abstract values of the module with its operations (state machine approach), or by describing a set of possible execution sequences (history approach), or by interrelating the visible effects of the operations (algebraic approach). In any case, operation execution is considered indivisible or "timeless".

The situation is fundamentally different with concurrent systems where operations may be executed concurrently and a process may block amidst an operation. Moreover, a module may not even be "passive", but may contain an active process. Modules in concurrent systems will be collectively referred to as concurrent modules throughout this paper, independent of their inner functioning.

Many programming language constructs have been proposed for the implementation of concurrent modules. Among the best known are the process (in different flavours), the monitor [Hoare 74] and the Ada task [Ada 82]. Note that each module concept has its specific paradigm of module interaction; see, e.g., [Wegner/Smolka 83] [Andrews/Schneider 83].

The fundamental difference between a sequential module and a concurrent module has given rise to completely different approaches to concurrent module specification. On the one hand, we have the extension of classical sequential techniques to support the specification of "guarded monitors": a passive module is used concurrently (but without overlapping), and blocking can occur at module entry only [Laventhal 79] [Keramidis/Mackert 79]. This works fine in simple cases, as long as no scheduling properties have to be specified and module composition and liveness properties are of no concern.

On the other hand, methods originally developed for fine-grain concurrency analysis have been extended for use in module specification. Temporal logic plays a prominent role here, and there is a multitude of sophisticated temporal logic approaches, differing both in style and the degree of "state machine orientation" (see, e.g., [Schwartz/Melliar-Smith 82] [Hailpern/Owicki 82] [Lamport 83]).

Much of the complexity of these approaches can be avoided if there is no need for complete liveness analysis. We propose a specification method based on state machines which produces specifications both compact and readable. The method

- is general in that it allows any conceivable concurrent module to be modelled as a state machine;
- it is flexible in that it supports module parametrization and composition, and allows to write partial specifications (sometimes called implementation specifications);
- it is adaptable to existing sequential specification methods and languages, its essence being independent of the state machine view.

In section 2 of this paper, the basic ideas will be outlined. Section 3 deals with different possibilities to implement an abstract concurrent module. The notion of implementation specification (as opposed to service specification) is introduced in section 4; different paradigms of module interconnection are also discussed in this section. We conclude with section 5, commenting on context and perspective of our work.

2. Service Specification of a Module

2.1. The basic model

Let us think of a module as an independent agent that offers certain services. The agent will obey orders to execute these services: it acts as a server to certain clients producing the orders. At the moment, we do not care about the nature of clients or things such as "order transmission".

When a module accepts an order, the requested service is executed either instantaneously or with a certain delay. In the latter case, the completion of the service will be the byproduct of the acceptance of another order. In general, acceptance of an order implies the completion of 0, 1, or more orders, possibly including the order just accepted. Acceptance of an order, together with the resulting completion of previously accepted orders, is considered an indivisible, "timeless" action of

the module.

A module M can be modelled as a special purpose state machine $M = (S,A,B,\tau,\rho,s_0)$, where S denotes the states, A the inputs, and B the outputs. τ and ρ are the state transition and output functions, respectively. $s_0 \in S$ is referred to as the initial state. S has a special structure, and so has B:

$$S = V \times N \times P(N \times A),$$
$$B = P(N \times R).$$

V stands for "value", N for natural numbers, P for powerset, A for "argument", and R for "result". The following must hold for τ, ρ, s_0:

$$\tau((v,n,P), a) = (v',n+1,P'), \qquad P' \subset P \cup \{(n+1, a)\}$$
$$\rho((v,n,P), a) = Q, \qquad Q_1 = (P_1 \cup \{n+1\}) - P'_1 \quad (*)$$
$$s_0 = (v_0,0,\phi)$$

Q_1 denotes the 1-projection of Q, i.e., if $Q = \{(n_1,r_1),..,(n_k,r_k)\}$ then $Q_1 = \{n_1,..,n_k\}$. ϕ denotes the empty set.

Given a certain state $s = (v,n,P)$, v is the abstract value of M, n identifies the last order accepted, and P is the set of orders accepted but not yet completed; it will be referred to as the "order list". An output Q can be seen as a set of completion notifications for previously accepted orders. Equation (*) specifies that, upon acceptance of an order, exactly those orders which are completed are removed from P.

The mappings τ and ρ need not be completely defined - but their domains are identical. We say that if $\tau(s,a)$ (or, equivalently, $\rho(s,a)$) is undefined, "M does not accept a in state s".

Given a module M as above, a sequence of arguments $a_1a_2...a_i$ models a sequence of orders and determines a sequence of state transitions $s_0 \rightarrow s_1$, $s_1 \rightarrow s_2$, ... , $s_{i-1} \rightarrow s_i$ (if all are defined) and a corresponding sequence of sets of completion notifications $b_1b_2...b_i$ for the orders given. Due to (*), there will be at most one notification for each order.

2.2. Describing a model

The behaviour of a module as modelled above can be described using any specification method/language that seems convenient. Throughout this paper, we will use a method in the tradition of the state machine view.

It must be emphasized, however, that the model is not tied to any specific language. As we do not want to elaborate on specification languages, we will use an ad-hoc language without giving precise syntax and semantics.

Let us first "structure" both the module state and the argument/result by allowing these to be tuples and their components to be typed. In addition, we use the notion of service with the meaning of "operation", "entry", or "port". Consequently, the flavour of "argument/result" changes a little: associated with each service are a certain number of arguments and a certain number of results, all of certain fixed types. An order is the request to execute a certain service with given arguments, and to deliver the results.

Accepting an order changes the state and may produce completion notifications. These effects are specified using an ad-hoc language reminiscent of sequential specification languages like Special [Robinson et al. 79] or Ina Jo [Eggert 80] [Cheheyl et. al. 81]. To give a flavour of both the method and the language, let us

consider a simple example:

> We wish to specify a module that is capable of storing one value of a certain type, say *text*. There are two operations, ***put*** and ***get***, for storing and removing, resp., such a value. ***put***ting a value into the module can succeed only when no value is present. ***get***ting a value from the module can succeed only if a value is present. There is one special requirement, though: ***get*** has a ***boolean*** argument ***priority***; we want to specify a program module that, in the case of two "pending orders" with ***priority*** and **not** ***priority***, resp., will complete the ***priority*** order upon arrival of a value.

Now turn to the following specification text:

```
module mailbox is
        put (message: in text),
        get (priority: in boolean; message: out text);
components box: text;
initially box = ?;
service put when box = ? is
        if orders = φ then
             box' = message
        else box' = ? and
             for that x in orders sat x.priority or that x in orders sat true
             return message' = this.message fi and
        return;
service get is
        box' = ? and
        if box ≠ ? then
          return message' = box fi;
end mailbox.
```

Although the reader may guess the meaning of many phrases in this example, several comments on the ad-hoc language used here are necessary.

(1) The module head contains the syntactical part of the specification (this corresponds to the "signature" in an algebraic specification and to the "definition module" of Modula-2 [Wirth 82]).

(2) The initial state is characterized by an assertion (it need not be unique).

(3) ? is a special value available for all types; it cannot be **return**ed, though.

(4) Each service is specified by an assertion that relates old values of state components to new ones (the latter are denoted by primed component names). A component with no primed occurrence remains unaffected. Note that the description of the state transition may be highly non-determinate.

(5) The specification of a service may contain expressions of the form '**return** assertion' or '**for** order expression **return** assertion'. Such a return expression specifies the completion of the current order or the order indicated, resp. The assertion serves to characterize the return values.

(6) The expression '**that** x **in** set **sat** P(x)' denotes an arbitrary element in a given set satisfying P, or ? if no such element is present. **or** means determinate choice if the operands are non-boolean: the first operand is selected if it is not ?; otherwise, the second operand is selected.

(7) The predefined identifier *orders* denotes the order list. Its elements are variant records, according to the information given in the module head.

A salient feature of the above specification is that overflow and underflow of the mailbox are prevented by different techniques. Overflow is prevented by simply not accepting a ***put*** order if the message cannot be deposited. A ***get*** order is always accepted; its completion, however, may be delayed until a message is available and can be granted - according to the priority discipline. Note that without accepting the ***get*** orders it would be impossible to express the priority scheduling.

2.3. Non-determinacy restricted by temporal logic assertions

Non-determinate service description as possible with an assertion-oriented specification language is a powerful tool. It allows for incomplete specification of modules in cases where abstraction from behavioral details is desired. It also allows to specify modules that are intentionally or de facto non-determinate.

In the latter case, it is often necessary to restrict the set of possible state sequences to a subset with certain properties. Such restrictions cannot be expressed by relating a state to its successor state. In this situation, temporal logic can be used to specify the desired properties of state sequences (see, e.g., [Manna/Pnueli 81]). The specification may contain a phrase like

assertion temporal assertion;

Note that an assertion of the form [] P, where P is a state predicate, introduces P as a module invariant. Unlike invariants known from other specification languages, P is not meant to be provable from the rest of the specification; its purpose is to restrict the module behaviour. - The introduction of the **assertion** clause makes the **initially** clause obsolete (as can be seen in the next example).

Consider the following example which specifies an unreliable transmission mediium. This might be, e.g., a communication line between two computers, one acting as a transmitter of certain "packets", the other acting as a receiver. There is a packet buffer at the receiver site, but no flow control to prevent overwriting of this buffer. Each packet sent will be received, although possibly damaged.

```
module line is
        xmit (data: in packet),
        recv (data: out packet);
components buffer: packet;
assertion buffer = ? and []<> buffer ≠ ?;
service xmit is
        (buffer' = data or buffer' = ?) and
        return;
service recv when buffer ≠ ? is
        buffer' = ? and
        return data' = buffer;
end line.
```

The specification of *xmit* shows that transmission is unreliable. The temporal assertion, however, precludes permanent corruption of all packets.

2.4. Autonomous state transitions

It is easy to extend the basic model of 2.1. with autonomous state transitions. There are many practical cases where autonomous state transitions are useful. Typically, a process performing a certain background task over and over again will be modelled conveniently by using an autonomous state transition. As a simple example, consider the following *clock* module:

```
module clock is
        get time (t: out natural);
components time: natural;
assertion time = 0;
service get time is
        return t' = time;
auto tick is
      time' = time + 1;
end clock.
```

Note that the identifier *tick* serves documentation purposes only.

Autonomous transitions add to the degree of non-determinacy of a module. So it is not surprising that autonomous transitions will often be used in conjunction with a temporal assertion. The following example specifies a "bad memory" which is neither completely reliable nor completely broken:

module *memory* **is**
    read (v: **out** *value),*
    write (v: **in** *value);*
components *cell: value;*
assertion $cell = ?$ **and** $[]\langle\rangle\ (cell \neq ? \rightarrow \bigcirc cell \neq ?)$
service *read* **when** $cell \neq ?$ **is**
    return $v' = cell$;
service *write* **is**
    $cell' = v$ **and return**;
auto *forget* **is**
    $cell' = ?$;
end *memory*.

The temporal assertion contains the "next state" operator $\bigcirc$. The assertion states that (1) the memory is an undefined state initially and (2) it will happen infinitely often that when a state is defined its successor state will also be defined. The latter requirement guarantees that a value written into the memory can be read again - at least from time to time.

Note that the **when** clause does not necessarily imply a "delay" of the ***read*** operation; whether $cell = ?$ causes a "delay" or "exception" is beyond the expressiveness of the basic model (see, however, section 4.).

3. Correspondence between Modules and Programming Language Constructs

3.1. Processes with message passing

In a straightforward implementation, a module is realized as a sequential process interacting with its enviroment via message passing. There are different paradigms for inter-process communication. We assume the following basic chracteristics:

(1) The code of a process contains certain points where the process is willing to accept a message. If none is available (see below), the process waits for the arrival of a message.

(2) A message received by the process represents an order: it describes a service to be executed by the process.

(3) At any point, a process may decide to accept certain kinds of orders; these may be described by service names and/or certain predicates involving parameters and the process state.

(4) The process is able to signal the completion of an order by "sending an answer". It is irrelevant here, whether and how this answer is bound to the order, to a client process, to a third process, or whatever. Sending an answer does never imply waiting.

As is evident from (1) and (4), we abstract from the details of message/answer buffering, in fact from the whole inter-process communication scheme. E.g., we ignore issues like sending a message or receiving an answer. We just have an "environment" producing messages and swallowing answers. Issues of module

interconnection are deferred to section 4.

Any process with the properties mentioned above can be modelled as a concurrent module. The activity between acceptance of a message and the next acceptance of a message is modelled as a state transition. If the locus of control is encoded in the module state, the **when** clause can be used to express that a process is choosy about when to accept which kind of order. The **when** clause also serves to reflect the predicates mentioned in (3). Note that the process has to take explicit measures for the bookkeeping on orders!

3.2. Processes with remote invocation

The Ada task [Ada 82] is an example for a process using the remote invocation paradigm. The task is an autonomous process with the following characteristics (cf. 3.1.):

(1) The code of a task contains accept statements carrying the names of task entries. Accept statements may be nested. At the beginning of an accept statement, the task waits for a corresponding entry call issued by a client task, then accepts the input parameters, executes the body of the accept statement, and delivers the output parameters to the client.

(2) An entry call accepted by a task represents an order: it describes a service to be executed by the task; executing the body of the accept statement is part(!) of the execution of the service (see below).

(3) Using the selective wait statement, a task can accept different service requests, whichever happens to arrive: a group of accept statements mentions the acceptable entry calls; moreover, each accept statement may be guarded by a **when** clause.

(4) The completion of an order accepted by a certain accept statement is signalled through termination of that statement. This comprises return of output parameters. Note that the client is waiting for the termination.

As with message passing, we presuppose that a task to be modelled as a concurrent module does not use another task (this restriction will be removed in section 4). The activity between acceptance of an entry call and the next acceptance of an entry call (possibly nested) is modelled as a state transition. If the task body consists of a loop containing a selective wait, but no other accept statements, this structure will be directly mirrored in the model specification. If the structure is different, the locus of control has to be encoded in the module state (as mentioned in 3.1.). Note that there is no explicit bookkeeping for orders! An order is represented by the activation of an accept body.

If a task uses the entry attribute *e'count* (denoting the number of pending calls for *e*), the model has to accept any call for *e* unconditionally. The service request will be registered in the order list so that the module behaviour can be made dependent on *e'count*. Clearly, the effect of service *e* will now be specified as the effect of a guarded autonomous transition.

Note that a selective wait can have an **else** part which will be executed if no other alternative can be selected. This can also be modelled using an autonomous transition.

3.3. Monitors

The monitor is the most natural generalization of the "module" as known from sequential programs. Different versions of synchronization within monitors have been proposed. We refer to the Hoare version [Hoare 74], but other versions could be taken as well.

Monitor activities are executed on behalf of a process calling one of the procedures exported by the monitor. The following characteristics of a monitor are pertinent to our modeling:

(1) A monitor is "active" when a process is within a monitor procedure and is not blocked. A monitor is activated when a process is entering it.

(2) A process entering a monitor is tantamount to the monitor accepting an order. The procedure called represents the service requested.

(3) If a monitor is ready to accept an order, it will accept any order. (This is different with "guarded monitors"!)

(4) An order is completed when a process leaves a monitor: termination of a procedure exported by the monitor means completion of the corresponding order.

If a monitor is modelled as a concurrent module, the exported procedures are modelled as the services. The activity within a monitor, lasting from an entry to the next entry, is described as the service effect. Different processes may take part in this activity, e.g., if a process wakes a blocked process using a signal operation. Furthermore, an activity period is not necessarily ended when a process leaves the monitor; monitor exit just means order completion. - No explicit bookkeeping on orders is necessary, since an order is represented by a procedure activation.

3.4. Synchronized sequential modules

A state transition in a concurrent module is an indivisible action. There is no concept of "overlapping transitions". Processes, tasks, and monitors correctly implement indivisible state transitions, due to their seriality and exclusion properties, respectively. It is nevertheless possible to weaken the restriction that two or more processes must not be proceeding simultaneously within an implementation of a module.

The important observation is that clients of a module, not aware of the module's implementation, are not even aware of exclusion or serialization measures within the module. These are exclusively in the module's responsibility. E.g., if a concurrent module is implemented as a single sequential program module containing some locking operations, this is perfectly alright as long as the clients receive the same services as they would receive from a monitor implementation. Service is a matter of specification, exclusion/locking is a matter of implementation.

Verifying such a module implementation is of course more difficult than verifying a monitor or process. In addition to proving the correctness of each individual operation, we have to prove the following:

> Consider an arbitrary, overlapping execution E of orders $o_1,\dots,o_n$. Then there is a permutation $i_1,\dots,i_n$ of $1,\dots,n$ such that the net effect of the acceptances of $o_{i_1},\dots,o_{i_n}$ as specified for the abstract module is satisfied by the net effect of E.

Note that this notion of correctness does not guarantee determinacy, not even if the specification is determinate. This is not a weakness, though, it is the natural consequence of concurrent order arrival. Using data base terminology (see, e.g.,

[Ullman 80]), E involves a certain schedule for the transactions $o_1,\dots,o_n$; if the specification is determinate, we have to prove that all schedules are serializable.

If the specification is non-determinate, an overlapping execution may be allowed to produce a net effect that cannot be achieved by any serialization. This is because a determinate implementation will not usually exhaust all the possibilities allowed by a non-determinate specification. Thus, a correct implementation of a non-determinate module may well allow non-serializable schedules. A simple example will suffice to demonstrate this:

```
module growing is
        increase;
components counter: natural;
assertion counter = 0;
service increase is
        counter' > counter and return;
end growing.
```

Implementing *increase* by the non-atomic assignment *counter:= counter+1* on an integer variable *counter* with no locking is correct

- because any overlapping executions of the assignment will have a net effect that satisfies the net effect of a corresponding sequence of orders;
- despite the fact that, e.g., two overlapping executions may produce an effect different from that of two serial executions.

3.5. Multiple processes

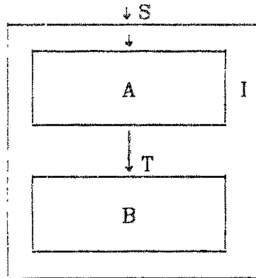
We have seen that a module implementation may contain more than one locus of control at a given time. Similarly, it is possible that a module implementation is an arbitrary "non-sequential process", i.e., contains several permanent processes interacting in arbitrary ways. And vice versa, any concurrent system of arbitrary complexity can be modelled as a concurrent module - if its visible behaviour is that of a server. A simple example is the "resource" module of the language SR [Andrews 81] which may comprise multiple processes. The most general way to obtain a concurrent system is of course composing arbitrary abstract modules. This is the subject of the next section.

4. Building Modules from Modules

4.1. Implementation Specification

If an implementation of a module A makes use of another module B (or of several other modules), it is possible to "specify the implementation". With respect to the server B, A acts as a client. From this point of view, a state transition of A is caused not only by accepting an order, but also by taking notice of a completion notification from B which signals the completion of an order previously issued by A. Of course, the effect that A issues an order must be specifiable as well.

Let us assume that B is a private server of A, i.e., B has no other clients. Then we can picture the situation as follows:



The clients of A rely on a certain service specification S. They don't know anything about B. The implementation of A relies on B's service specification T. The nature of this reliance is described by an implementation specification I for A.

As an example, let us consider the implementation specification for a scheduler A that controls access to a resource B; e.g., let A be a disk driver and B the corresponding disk drive (including the controller). A client of A sees an abstract disk which is described by the following service specification:

```
module disk is
        transfer (bn: in blockno; bl: in out block; dir: in (in,out));
components sector (1..last): block;
assertion true;
service transfer when 1 ≤ bn and bn ≤ last is
        if dir = in then return bl' = sector(bn)
        else sector'(bn) = bl and
             all i in 1..last sat (i ≠ bn → sector'(i)=sector(i)) and
             return fi;
end disk.
```

Of course, the specification doesn't say anything about scheduling, because scheduling does not affect the module's functional behaviour. For this reason, however, the specification can also be used for the disk drive! An implementation specification for the disk driver may look like this:

```
module disk driver
        using disk <transfer (bn: in blockno; bl: in out block; dir: in (in, out))>
        is transfer (bn: in blockno; bl: in out block; dir: in (in, out));
components devq: device queue; (* technical details are omitted *)
assertion length(devq) = 0;
accept transfer is
        devq' = enter(devq, this) (* realizes the scheduling discipline *)
        and
        if length (devq) = 1 then
          start disk.transfer(head (devq').bn) fi;
notice disk.transfer is
        devq' = body(devq) and
        if length(devq') > 0 then
          start disk.transfer(head (devq').bn, head (devq').bl,head (devq').dir)
        fi and
        for head(devq) return
            if dir = in then bl' = this.bl fi;
end disk driver.
```

We use the keyword **accept** instead of **service** in order to enhance the "implementation flavour".

A **start** expression specifies that a certain order is produced (like **return** produces a completion notification). At the moment, we don't care whether or not the server is ready to accept the order. A **notice** specification describes the effect of

accepting a completion notification. Noticing will be selective in the general case: there will be different notice specifications for different services, and they may be guarded by **when** clauses.

The order list is not visible in this example. References to pending orders, however, are kept in *devq*, and the expression **for** *head(devq)* **return** ... specifies an order to be removed from the order list.

As indicated in the module head, *disk driver* refers to the *disk* module. *disk* can be seen as a formal parameter of *disk driver*. The semantics of *disk driver.transfer* depends on the specification given for *disk* and on the details of inter-module communication.

It should be noted that the **notice** concept is peculiar to concurrent systems. There is no need for it in sequential systems where an order will always be executed to completion prior to acceptance of the next order. (Cf. the **effect of** clause used in Special [Robinson, et al. 79].)

4.2. Module composition

The easiest way to model communication between modules is to assume the existence of a message pool which is capable of buffering an unlimited amount of orders and completion notifications. A state transition of a module (if not autonomous) removes a message from the pool and may enter new messages into the pool. If the state of a module allows consuming an order or completion notification present in the pool, that module is said to be "enabled". In a set of communicating modules, any enabled module may "switch" at any time. Each **service/accept/notice** will consume its messages in arrival time order.

It is readily clear that this modelling is appropriate for classical inter-process communication via mailboxes and ports. Interestingly, it is applicable to monitors as well. Consider the "weak monitor" first; it will release exclusion when calling another monitor [Haddon 77], which means that it will always be ready to consume any message sent to it. Thus, the infinite message pool is an adequate model although we don't see a message pool in the realization. (Note that queues of processes waiting to enter a monitor have nothing to do with the abstract message pool; they are a transient phenomenon of the implementation only, serving to achieve atomicity of state transitions.)

A problem arises with the regular, "strong" monitor and with the Ada task. Nested invocations may cause a module to get stuck with an invocation statement. However, since an invocation statement requests synchronous service we are in good shape: the implementation specification of the module has to specify that a transition producing an order will result in a state that refuses to consume any message different from the corresponding completion notification. So a simple implementation specification for a client module might look like this:

```
module client using server <s> is
        service1, .... ; (* services offered by client! *)
components .... ,
             ready: boolean;
assertion ... ;
accept service1 when ready is
        ... and start server.s
           and not ready';
notice server.s is
        ... and ready';
end client.
```

We can conclude that it is not necessary to have different models for module interconnection; the "unlimited message pool" model will do. Communication peculiarities with different realizations of modules will be (and have to be) reflected in the implementation specifications.

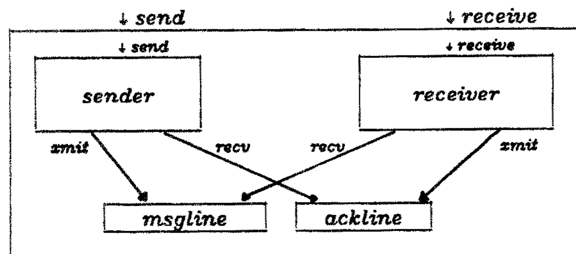
4.3. A non-trivial example of module interconnection

Several authors have used the alternating bit protocol as a vehicle for demonstrating specification techniques; see, e.g., [Schwartz/Melliar-Smith 82] [Lamport 83]. These papers also discuss the pros and cons of state machine orientation. Protocol specification using state machines is just a special case of our module specification. In considering the alternating bit protocol we will not encounter complex modules featuring order lists and delayed order completion; the example serves the mere purpose of demonstrating a non-trivial module interaction.

We assume to have two instances of an unreliable, unidirectional transmission line, similar to the *line* specified in 2.3. We would like to construct a reliable mailbox of finite capacity on top of the two lines. The mailbox should be similar to the mailbox specified in 2.2. The idea is to use one line for the transmission of messages, the other one for the transmission of acknowledgements. According to the alternating bit protocol, a message is tagged with a bit and transmitted repeatedly, until an acknowledgement carrying that tag is received; after this, repeated transmission of the next message is started, the tag being the complement of the last tag. The reliable mailbox is to behave as follows:

module *mailbox* **is**
    send (message: **in** *packet)*
    receive (message: **out** *packet);*
components *queue: infinite queue (* of messages *);*
    capacity: natural (limits number of queue elements *);*
assertion *length(queue) = 0* **and** [] *capacity* $\geq$ *1;*
service *send* **when** *length(queue) < capacity* **is**
    queue' = enter(queue, message) **and**
    return;
service *receive* **when** *length(queue) > 0* **is**
    queue' = body(queue) **and**
    return *message' = head(queue);*
end *mailbox.*

The mailbox is to be constructed from four components: a message line, an acknowledgement line, a sender module and a receiver module. The functional hierarchy will look like this (cf. the picture in 4.1):



The details of the *sender/receiver* modules are decribed by the following implementation specifications:

```
module sender
        using msgline, ackline <xmit(data: in message; tag: in boolean);
                                recv(data: out message; tag: out boolean)>
        is send (msg: in message);
components buffer: message; bit: boolean;
assertion buffer = ? and not bit;
accept send when buffer = ? is
        buffer' = msg and return;
auto retransmission when buffer ≠ ? is
      start msgline.xmit(buffer, bit) and
      start ackline.recv;
notice ackline.recv is
       if tag = bit then
         bit' ≠ bit and buffer' = ? fi;
end sender.
```

```
module receiver
        using msgline, ackline <xmit(data: in message; tag: in boolean);
                                recv(data: out message; tag: out boolean)>
        is receive(msg: out message);
components buffer: message; bit: boolean;
assertion buffer = ? and bit;
accept receive when buffer ≠ ? is
        buffer' = ? and return msg' = buffer;
auto retransmission is
      start ackline.xmit(nil, bit) and
      start msgline.recv;
notice msgline.recv is
       if tag ≠ bit and buffer = ? then
         buffer' = data and bit' = tag fi;
end receiver.
```

Let us investigate the interaction between these modules and the *line* modules. An *xmit* order is readily accepted by a *line* module, so an implementation of **start**...*xmit*.. will not require order queueing. The absence of state transitions **notice**...*xmit* specifies that the module is not interested in completion notifications; an implementation may either discard these - or they may be implicit in a synchronous interface to the line.

Treatment of *recv* orders is more subtle. Remember that such an order is not accepted until an uncorrupted packet is available. According to the above specifications, the lines are repeatedly "triggered", by means of **start** ... *recv*, to deliver a packet. When an uncorrupted packet is available, the state transition **notice**...*recv* may occur. Note that an implementation need not "buffer the trigger signals". On the other hand, it is by means of the triggering that arrival of an uncorrupted packet will indeed be noticed. An implementation may well decide to try execution of a synchronous *recv* or, if this is not possible, to proceed with retransmission. (Note that Ada's "conditional entry call" supports this kind of programming.) Looping on this activity is a correct implementation of what is specified by the **auto** and **notice** transitions.

4.4. Another example: deadlock in interconnected modules

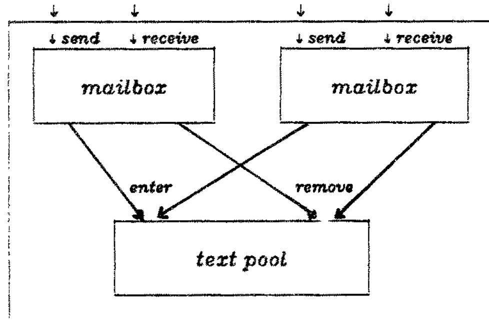
We were able to specify the alternating bit protocol without recurrence to the notion of an order list. More general cases may require the usage of order lists for explicit specification of blocking. The added complexity introduces the danger of deadlock, as shown in the following example.

Our goal is to construct a module that represents two mailboxes which are functionally independent but share a common pool of message frames. Each mailbox is to be represented by a queue of numbers that serve to identify message frames in the pool. The head of the pool specification may read like this:

```
module text pool is
        enter (message: in text; id: out natural),
        remove (id: in natural; message: out text);
```

As the pool has a finite capacity, say ***cap***, an ***enter*** order will not be accepted when no empty frames are available. Completing the specification of the pool is left to the reader.

The desired module is constructed as indicated in this picture:



We omit the service specification of this module and turn our attention to the implementation specification of ***mailbox***. If we have a monitor implementation in mind, we have to take into account the respective remarks in 4.2. This leads to the following specification:

```
module mailbox
        using text pool <enter(message: in text; id: out natural),
                         remove(id: in natural; message: out text)>
        is send (message: in text),
           receive (message: out text);
components queue: infinite queue (* of ids *),
           locked: boolean;
assertion length(queue) = 0 and not locked;
accept send when length(queue) < cap and not locked is
        start text pool.enter(message) and locked';
notice text pool.enter is
        queue' = append(queue,id) and not locked' and
        for that x in orders sat true return;
accept receive when length(queue) > 0 and not locked is
        queue' = body(queue) and locked' and
        start text pool.remove(head(queue));
notice text pool.remove is
        not locked' and
        for that x in orders sat true return message' = this.message;
end mailbox.
```

Analysis of the module triple constructed from ***text pool*** and two instances of ***mailbox*** reveals that deadlock may occur. If the pool is exhausted it will not accept an ***enter*** order given by a mailbox, which means that the transition **notice** ***text pool.enter*** will not take place. If both the mailboxes experience this situation after a transition **accept** *send*, both will be in the *locked* state. As a consequence, they are disabled forever.

Of course, this effect is not too surprising to those who know of the problems with nesting strong monitors. Deadlock does not occur if the *locked* component is removed from the above example. The resulting specification describes the behaviour of a weak monitor.

5. Conclusion

It has been demonstrated that a simple state machine augmented with an order list is capable of modelling complex modules in a concurrent environment. The approach taken is applicable to modules obeying the client/server paradigm and rests on splitting acceptance and completion of an order.

We have seen how this kind of modelling works for different kinds of concurrent modules known from programming languages. Moreover, an arbitrarily complex concurrent system can be modelled as an abstract module, provided it acts like a server to the outside world. The approach is essentially independent of particular techniques and languages for specifying the model (although we found it practical to use a state machine technique).

The notion of implementation specification, mainly known from the area of protocol specification, has been demonstrated to involve modelling of inter-module communication. An implementation specification for a module determines the nature of communication between that module and its servers, whether buffered or unbuffered, synchronous or asynchronous, or even "tentative" as in the conditional entry call of Ada.

Work remains to be done in the area of verification. This includes both implementation verification and composition verification. While the former can be attacked along well-established lines of program verification, the ground is not well prepared for the latter (cf. [Lamport 83], footnote 2!). We are working on a formal framework that allows to prove that the behaviour of a system of interconnected modules with given specifications meets a specification given for that system.

It remains to be mentioned that the perspectives for prototyping concurrent systems seem promising. Executable service specifications will allow to exercise models. If implementation specifications are made executable as well, it will be possible to test concurrent systems built from concurrent module prototypes.

Acknowledgement

We would like to thank the referees for their comments which helped to improve the presentation.

References

[Ada 82] Reference Manual for the Ada Programming Language. ANSI/MIL-STD 1815A, Ada Joint Program Office, 1982

[Andrews 81] G.R. Andrews: Synchronizing Recources. ACM TOPLAS 3.4, October 1981

[Andrews/Schneider 83] G.R. Andrews/F.B. Schneider: Concepts and Notions for Concurrent Programming. ACM CS 15.1, March 1983

[Cheheyl et al. 81] M.H. Cheheyl/M. Glasser/G.A. Huff/J.K. Millen: Verifying Security. ACM CS 13.3, September 1981.

[Eggert 80] P.R. Eggert: Overview of the Ina Jo Specification Language. TR SP-4082, SDC Santa Monica, October 1980

[Haddon 77] B.K. Haddon: Nested Monitor Calls. ACM Operating Systems Review 11.4, October 1977

[Hailpern/Owicki 82] B. Hailpern/S. Owicki: Modular Verification of Concurrent Programs. Proc. 9. Ann. ACM Symp. on Principles of Programming Languages, Albuquerque 1982

[Hoare 74] C.A.R. Hoare: Monitors: An Operating System Structuring Concept. CACM 17.10, October 1974

[Keramidis/Mackert 79] S. Keramidis/L. Mackert: Specification and Implementation of Parallel Activities on Abstract Objects. Proc. 4. Int. Conf. on Software Engineering, München 1979

[Lamport 83] L. Lamport: Specifying Concurrent Program Modules. ACM TOPLAS 5.2, April 1983

[Laventhal 79] M.S. Laventhal: A Constructive Approach to Reliable Synchronization Code. Proc. 4. Int. Conf. on Software Engineering, München 1979

[Manna/Pnueli 81] Z. Manna, A. Pnueli: Temporal Verification of Concurrent Programs. In: The Correctness Problem in Computer Science (R.S. Boyer, J.S. Moore, Eds.). Academic Press, London 1981.

[Robinson et al. 79] L. Robinson/B.A. Silverberg/K.N. Levitt: The HDM Handbook. SRI International, Menlo Park 1979

[Schwartz/Melliar-Smith 82] R.L. Schwartz/P.M. Melliar-Smith: From State Machines to Temporal Logic: Specification Methods for Protocol Standards. IEEE Trans. Comm. 30.12, December 1982

[Ullman 80] J.D. Ullman: Principles of Data Base Systems. Computer Science Press 1980

[Wegner/Smolka 83] P. Wegner/S.A. Smolka: Processes, Tasks, and Monitors: A Comparative Study of Concurrent Programming Primitives. IEEE-SE 9.4, July 1983

[Wirth 82] N. Wirth: Programming in Modula-2. Springer-Verlag, Berlin 1982.