Using Formal Validation Techniques to Develop a Microkernel

PJA de Villiers
Department of Computer Science
University of Stellenbosch, Stellenbosch 7600, South Africa

October 14, 1997

Abstract

Computer-aided validation is an established field of research and the techniques are mature enough for industrial application. Currently, however, suitable case studies are needed to introduce these techniques to industry. In this paper the validation of a microkernel is described. While the validation of a specific design is a one-time event, the techniques presented can be applied to validate other reactive systems.

1 Introduction

In mature engineering disciplines it is standard practice to develop models or prototypes of designs before implementation. The main advantage is, of course, that models can be analysed to detect errors as soon as possible. Computer-aided validation is basically the same idea applied to software engineering. An abstract version of a program—a model—is developed and analysed to detect logical errors before an implementation is attempted.

Models are expressed in a special validation language that is basically a kind of high-level programming language designed for efficient validation. The correctness properties to be checked are usually expressed as temporal logic formulae. To check that a specific correctness claim holds for a given model, a computerised validation system is needed. Examples of correctness properties that are important to check for concurrent systems are: deadlock freedom, mutual exclusion, absence of starvation, correct response to all external events, etc. When some correctness property is violated, an execution sequence is produced that leads to the error. This is a great help to find and correct errors during the design process.

Validation techniques revolutionised the design of protocols. However, most other reactive systems, such as operating systems, process control systems and embedded systems, are still designed in the traditional way. The goal of this paper is to describe how computer-aided validation was used to design a microkernel. Instead of just presenting a model of a system and some correctness claims that were verified, the goal is to describe the model-based design process. A first version of the kernel was implemented in Modula-2 [4]. Using this design as a basis, the kernel was rewritten in Oberon to use the object-oriented features of the language to develop extensible device drivers [9]. This second implementation was guided by the validation effort. To study the interactions between various kernel components, a high-level model of the kernel architecture was developed first. This model is described in Section 3. Various components were refined by developing separate models. To illustrate this technique, a model of interprocess communication is presented in Section 4. An example of the final Oberon code is described in Section 5 to show the correspondence between a model and its implementation. Finally, the experience gained is summarised as a set of modelling guidelines that may simplify the development of other reactive systems.

Perhaps the research that is most closely related to the work described here is that of Cattel—a first attempt at validating a real-time kernel [1]. The project revealed a serious design error that had remained undetected. However, the modelling process is not described in the paper.

2 Validation System

The validator used to validate the microkernel was designed from scratch. While this was a sure way to learn about such systems, it also provided the opportunity to adapt the validator to the task at hand. Models are expressed in a validation language that is called ESML. It is a process oriented language based on CSP [5] and guarded commands. Only those features of ESML that are needed to understand the kernel models will be described here. A more detailed discussion and motivation of the design of ESML can be found in [3].

An ESML model is translated by a compiler into an equivalent low-level form (a finite transition system) that can be executed by the validator to explore all reachable states. To make the validation process efficient, as many states as possible are stored to detect revisited states. This makes it essential to represent states compactly to save memory and this concern is reflected in the design of the validation language. For example, user definable types include the idea of subranges to restrict the size of states. In addition, efficient state compression techniques are used to further reduce the state size. The model shown below illustrates the basic ideas of ESML:

```
MODEL Example;
TYPE INT = 0..10; Stream = {int(INT)};
VAR ch0, ch1: Stream;

PROCESS Sender(OUT out: Stream);
BEGIN
  DO TRUE ->
   IF TRUE -> out!int(1)
   [] TRUE -> out!int(2)
   END
  END
END Sender;

PROCESS Buffer(IN in: Stream; OUT out: Stream);
VAR m: INT;
BEGIN
  DO TRUE -> in?int(m); out!int(m)  END
END Buffer;

PROCESS Receiver(IN in: Stream);
VAR m: INT;
BEGIN
```

```
23   DO TRUE -> in?int(m) END
24 END Receiver;
25
26 BEGIN
27   Sender(ch0); Buffer(ch0, ch1); Receiver(ch1)
28 END Example.
```

The model defines three processes. The line numbers are for purposes of referencing only. The processes *Sender* and *Buffer* communicate over a channel *ch0* and *Buffer* and *Receiver* communicate over another channel *ch1*. The messages sent over both channels are of type Stream, which can be integers ranging from 0 to 10 (line 2-3). As is typical for reactive systems, the processes execute non-terminating loops. Process *Sender* decides nondeterministically to send either a 0 or a 1 over channel *ch0* to *Buffer* (lines 8-9). An IF command with guards that are both true is used to avoid modelling the selection mechanism explicitly. Process *Buffer* receives these messages and transmits them in a similar way over channel *ch1* to process *Receiver*. The command *in?int(m)* (line 17) indicates that a message *int* is received from port *in* and stored in variable *m*. Similarly, the command *out!int(m)* is used to transmit a message. Ports are connected to channels when processes are activated. For example, channel *ch0* is connected to output port *out* of process *Sender* and input port *in* of process *Buffer* (line 27).

3 Architectural Model of the Kernel

It is hard to justify the cost of developing several prototypes during the traditional design phase. The correctness of the design therefore depends on informal reasoning alone. It is hardly surprising that many logical errors remain undetected until the first implementation is attempted or, even worse, until the completed system has been in use for some time.

The goal of model-based design is to change all this. Designs are described in a formal notation and automated validation makes it feasible to formally prove that each idea works as the design proceeds. Reasoning starts from a set of requirements and the goal is to develop and validate a series of partial models by refining ideas until a final validated design is obtained that is detailed enough to serve as a blueprint for implementation. This strategy was used to develop a validated design for the Gneiss microkernel. The development process, rather than the complete kernel design, is described here.

A firm decision was taken about the relative importance of some conflicting software properties: simplicity was placed first because complex code is often unreliable and simple systems are easier to validate. However, to rule out overly simplistic designs, efficiency was placed second. Adaptability was considered least important since it is often in conflict with efficiency. Fortunately, apart from adding new device drivers, microkernels are seldom modified.

A microkernel is a resource manager that offers a service to client processes. It is natural to use client-server terminology when modelling a microkernel. The simplest kind of server—a *synchronous* server—forces clients to wait for each operation to be completed. A server is always willing to engage in one of a number of operations; which one, is determined by its clients. Some synchronous servers only manage data and a simple input-output style of interaction is adequate.

A more complex kind of server—an *asynchronous* server—is sometimes needed. The simplest kind of asynchronous server accepts a request for service and allows its clients to proceed without waiting. However, only one operation can be accepted at a time; all further requests for service are ignored until an *external signal* arrives to indicate that the operation has been completed. An asynchronous server can be made to accept more than one request at a time by adding a buffer. This can improve throughput substantially if the overhead of buffering is low compared to the average time needed for operations to complete. Many peripheral device drivers can be modelled by such an asynchronous server that maintains a list of pending requests.

The first task was to develop a high-level model of the kernel to define a workable structure. The kernel is seen by user processes as a synchronous server that provides a transaction service. The design process started from that perspective. Transactions are supported by kernel calls. A server called `Running` manages the currently active user process. The following model captures the kernel call mechanism between a user process and the kernel:

```
PROCESS User(OUT running: Trap);
BEGIN
  DO TRUE -> running!kcall END
END User;

PROCESS Running(IN request: Trap);
BEGIN
  DO TRUE -> request?kcall END
END Running
```

The OUT port of process `User` is coupled to the IN port of `Running` by a channel variable (not shown) and the message transmitted over this channel represents a kernel call. Process `Running` continually accepts requests (kernel calls) from `User`. The kernel actually offers three different kernel calls. A request for service is issued by executing the kernel call `Transaction`. The kernel call `ReceiveRequest` enables a server to accept a request and it responds by executing the `SendReply` kernel call. However, at this stage it is unnecessary to model such detail explicitly.

A typical kernel request starts an input or output operation on a peripheral device. The model was extended to include a device driver—another server used by `Running`. A request `doio` is issued by `Running` to start an input or output operation on behalf of the waiting user process:

```
PROCESS Running(IN request: Trap; OUT devicedriver: IOcommand);
BEGIN
  DO TRUE -> request?kcall; devicedriver!doio END
END Running;

PROCESS Devicedriver(IN request: IOcommand);
BEGIN
  DO TRUE -> request?doio END
END Devicedriver
```

There are two channels: one between `User` and `Running` and another between `Running` and `Devicedriver`. A set of legal messages (the channel alphabet) is defined for each channel. The definition of suitable channel alphabets was found to be important. Many design errors involve sending the wrong kind of message or specifying the wrong port. Alphabet definitions enable the validation system to detect such errors.

This simple model was easily validated for absence of deadlock, but it was still too unrealistic to be of much value: there was only one user process waiting for `Running` to accept a kernel call request, with `Running` waiting in turn on `Devicedriver` to accept a `doio` operation. Because device operations are slow, an asynchronous server was needed to allow user processes to be served while some slow operation on a peripheral device is in progress. For this, it was necessary to modify `Running` to respond to kernel calls as well as interrupts. The POLL command is used to select one of several possible messages.

```
PROCESS Running(IN in: Request; OUT todevicedriver: iocommand);
VAR curproc: procid;
BEGIN
  in?new(curproc);
  DO TRUE ->
    POLL in?kcall(curproc) -> todevicedriver!doio(curproc)
    [] in?int -> todevicedriver!iocomplete
    END; in?new(curproc)
  END
END Running;
```

It was also necessary to add another server called `Ready`—the first beginnings of a scheduler—to select a new process when the current process requested an IO operation. For simplicity, it was assumed that a new process to be activated would always be available. This meant that no scheduler queue was required; it could be added later.

```
PROCESS Ready(IN in: ProcID; OUT out: ProcID);
BEGIN
  DO TRUE ->
    POLL out!new(Proc) -> SKIP
    []  in?old(Proc) ->  SKIP
    END
  END
END Ready;
```

The model was refined to capture the activation of one user process at a time. User processes (different instantiations of the same process in ESML) were modified to wait for a resume signal after every kernel call. Resume signals are sent by `Running` to indicate which process to activate. The variable p in the message indicates which user process to activate.

```
PROCESS User(id: procid; IN in: Resume; OUT out: KernelCall);
VAR p: procid;
BEGIN
  (* process registers itself *)
  out!old(id);
  DO TRUE -> out!kcall(id);
    POLL in?resume(p) & p = id -> SKIP END
  END
END User;
```

A device driver was modelled as an asynchronous server that accepts requests for IO issued by the currently active user process. Such requests are handled by `Running`. Since `Running` responds to all external events, including interrupts, it reactivates the device driver when the IO-completion interrupt arrives. The variable `deviceidle` is used to ensure that only one IO request at a time can be pending. The appropriate user process is resumed by returning its process record to the scheduler. The device controller hardware is modelled as a simple synchronous server (not shown). It accepts IO commands from the device driver and generates an interrupt for every request.

Some peripheral devices work more efficiently when a queue of waiting requests is maintained. To exploit this idea, a request queue is used. When the device is idle, an IO operation can be started immediately, but if the device is busy, a request for IO must be entered in a queue. This means that, on arrival of the IO completion interrupt, the next IO operation can be started immediately. Operation *EMPTY* is used to initialise the request queue. The operations *APPEND*, *HD* and *REMOVEHD* are used to add and remove queue items, and *LEN* returns the length of the queue. Here is the device driver model:

```
PROCESS DeviceDriver(OUT toready: readyrequest;
  IN request: iocommand;
  OUT todevice: devicecommand;
  OUT touser: continue);
CONST qmax = 2;
TYPE requestqueue = LIST[qmax] OF procid;
VAR curproc, id: procid; rq: requestqueue; deviceidle: BOOLEAN;
BEGIN
  EMPTY(rq);  deviceidle := TRUE;
  DO TRUE ->
    POLL request?doio(id) & deviceidle ->
        curproc := id; todevice!startio; deviceidle := FALSE
    [] request?doio(id) & ~deviceidle -> APPEND(rq, id)
    [] request?iocomplete ->
       touser!resume(curproc); toready!enterproc;
       IF LEN(rq) = 0 -> deviceidle := TRUE
       [] LEN(rq) > 0 ->
          curproc := HD(rq);
          REMOVEHD(rq);
          todevice!startio
       END
    END
   END
END DeviceDriver;
```

After validating this global system model, the next task was to divide it into relatively independent subsystems, each consisting of a number of functionally related components. The first of these, the kernel proper, is centered around the currently active process and includes the following components: `Running`, that manages the currently running process, the scheduler `Ready`, and `Storage`, the memory manager. Device drivers form a second subsystem, designed as a set of asynchronous servers that respond to requests from the kernel proper. A third subsystem supports communication between clients and servers. The refinement of this subsystem is described here as an example.

4 Refining a Subsystem: Interprocess Communication

The communication subsystem supports message passing and manages ports—a flexible naming scheme for services. Ports provide an extra level of indirection to avoid direct naming of user processes that are communication partners. Inter-process communication is based on transactions, each representing a request directed at a specified server. Each server is known through a unique *port identifier*.

Communication in the Gneiss kernel was modelled for a single port, routing of messages via multiple ports being a straightforward extension. A port has an associated queue of processes waiting for messages to be transferred. Because messages are transferred immediately when a communication partner is available, a single queue per port is sufficient. It will contain either client processes or server processes, but not both at the same time.

The port queue was modelled as a list of process identifiers. The communication subsystem is a server that accepts the three kernelcalls `Transaction`, `ReceiveRequest` and `SendReply` as requests. The control flow mainly involves managing the port queue. There was no point in modelling the copying of messages explicitly. The identifier of the user process that executed a kernel call is passed on to the communication subsystem. A POLL command forms the backbone of the model, its three guards handling the three kernel calls:

```
POLL
   in?Transaction(cp) -> ...
[] out!ReceiveRequest(sp) -> ...
[] in?SendReply(sp) -> ...
END
```

A user process that executes a `Transaction` primitive is suspended until another user process—its communication partner—executes a matching `SendReply` primitive. The process identifier of the client process (the process that executed the `Transaction` primitive) is entered in the port queue. If a server process is already waiting on the specified port, the message associated with a transaction is transferred immediately from client to server. The server will later need to identify the client that issued the request and therefore a temporary binding between communication partners is set up before activating the server process by means of an array `Partner`.

```
IF clients OR LEN(pq) = 0 -> (* no server is ready *)
     APPEND(pq, cp); clients := TRUE
[] ~clients & LEN(pq) > 0 -> (* a server is ready *)
     sp := HD(pq)); REMOVEHD(pq);
     Partner[sp] := cp;
     toReady!enterProc(sp)
END
```

A server process executes a `ReceiveRequest` primitive to accept a request from a specified port. If there is a pending transaction, and thus a waiting client process in the port queue, the message is transferred immediately to the server. Otherwise, the server process is suspended and entered in the port queue.

```
IF clients & LEN(pq) > 0 -> (* a client is ready *)
   Partner[sp] := HD(pq); REMOVEHD(pq);
   toReady!enterProc(sp)
[] ~clients OR LEN(pq) = 0 -> (* no client is ready *)
   APPEND(pq, sp); clients := FALSE
END
```

A `SendReply` primitive is executed by a server process to complete a transaction. The server process and its communication partner are both returned to the scheduler queue.

```
toReady!enterProc(Partner[sp]);
Partner[sp] := nullproc;
toReady!enterProc(sp)
```

The final model for the communication subsystem is shown in Appendix A. It corresponds closely to the implementation as will be shown in the next section.

5 Implementation

In some cases it is possible to generate implementations directly from validated models. However, this is difficult in the case of low-level code that must be efficient. A good programmer can write more efficient code than can be produced by a general purpose code generator. It is proposed that models should be used as guidelines for coding to make errors unlikely. This approach is not defendable from a purely logical perspective, but it is currently the only reasonable approach where efficiency is important. The Oberon code that implements the *Transaction* primitive is included here to illustrate this technique. The code has been simplified somewhat to make it easier to understand, mostly by eliminating error handling code. Structurally, however, nothing has been changed. Local transactions are handled by the following procedure:

```
 1  PROCEDURE LocalTransaction(pid: GPorts.PortID;  req, rep: Message;
 2     timeout: LONGINT): LONGINT;
 3  VAR
 4    cp, sp: GProcs.Process; (* client, server processes *)
 5    ob: GObjects.Object; (* used to store a process record *)
 6    res: LONGINT; (* result of transaction *)
 7    p: GPorts.Port;
 8    cbs, sbs: BS; (* client, server blocked state record *)
 9  BEGIN
10    (* Phase 1:  code deleted to allocate cbs and look up the port
11    and assign it to p *)
12
13    (* Phase 2:  Communication *)
14    cp := GRunning.proc; (* client process *)
15    sp := NIL; (* server process *)
16    IF p.clients OR ~GObjects.GetP(p.queue, ob) THEN (* no servers *)
17      cbs.req := req^;  cbs.mptr := rep;  cbs.rep := rep^;
18      GObjects.Put(p.queue, cp);  p.clients := TRUE;
19      res := 0 (* successful, so far *)
20    ELSE  (* a server is available *)
21      cbs.req := nullmsg;  cbs.mptr := rep;  cbs.rep := rep^;
22      sp := ob(GProcs.Process); (* get the server process *)
23      sbs := sp.bs(BS); (* get message descriptor *)
24      (* copy request msg to server *)
25      res := GVMs.CopyMessage(req, sbs.req);
26      IF res = 0 THEN (* all seems ok *)
27        sp.state.EAX := 0;
28        cp.server := sp; sp.client := cp; (* link client to server *)
29        sbs.req.len := req.len;  (* number of bytes copied *)
30        GReady.EnterProcess(sp);  sp := NIL (* unblock the server *)
31      END;
32
33      (* Phase 3: Finalisation *)
34      IF res # 0 THEN (* error occurred in phase 2, undo everything *)
35      ELSE (* everything ok, block the client process *)
36        cp := GRunning.PreemptProcess(ContTransaction); cp.bs := cbs;
37      END
38    END; RETURN res
39  END LocalTransaction;
```

The detail of handling different ports could be ignored in the model, but not in the implementation. Queues of processes and other objects are manipulated by module GObjects. If the specified queue is non-empty, procedure *GetP* returns TRUE and removes the first object from the queue, assigning it to a specified variable of type GObjects.Object. If the queue is empty, *GetP* returns FALSE. Procedure *Put* enters an object in a specified queue. The correspondence between the model and implementation should be obvious. For example, lines 16–19 of the code shown above implement the following two lines from the model:

```
IF clients OR LEN(pq) = 0 ->
   APPEND(pq, cp); clients := TRUE
```

The code illustrates that the kind of detail needed for an efficient implementation can seldom be afforded in a model. For example, copying of data is ignored in the model. In the implementation a data structure cbs ("client blocked state") is used to store pointers to the data to be copied. Manipulation of cbs is shown in lines 17, 21 and 36.

6 Correctness Properties

The focus of this paper is on the development of validation models because that is the real hard work. The description of correctness claims is relatively simple. Temporal logic as a notation for specifying correctness claims is well-known. For example, a classification of many important correctness properties of reactive systems can be found in [8]. The temporal logic CTL was used to describe correctness properties of the Gneiss microkernel. CTL is a well-known notation for describing correctness properties of reactive systems and was first defined in [2].

Typical correctness claims were validated for the Gneiss kernel. The first property that was checked for every model is deadlock freedom. This is a straighforward property to check and essentially amounts to an exhaustive state exploration. Various other safety properties were validated, such as checking for buffer overflow. This can be expressed by a CTL formula of the form $AG(LEN(buf) \leq max)$. Intuitively, the formula means that the number of items in the buffer will always be at most max. Liveness properties such as responsiveness were also validated. For example, to check that the kernel will eventually respond to all kernel calls, a CTL formula of the following form was useful: $AG(\alpha \Rightarrow AF\beta)$. The property α means that a kernel call occurred and β means that the kernel call has been completed.

7 Experience and Conclusions

Modelling and formal validation have been used to develop a design for a microkernel. The final validation models were detailed enough to correspond closely to the implementation code. The kernel is efficient and reliable and is currently used in industry to support dedicated computer systems. It is difficult to measure the results of such a project, but some comments may be helpful to indicate the benefits obtained.

An important advantage is that modelling seems to simplify designs. Validated models have been developed for various kernel mechanisms and these models proved valuable to develop efficient implementations. Some mechanisms, like a scheduler, can be modelled in sufficient detail to make the mechanical generation of efficient code feasible. In most cases, however, too much detail is required to allow efficient validation. In these cases a validated model should be used as a guideline for coding.

The techniques described in this paper could be applied to other reactive systems as well. For example, the mechanisms that were validated successfully—schedulers, device drivers and interprocess communication—are also used in most embedded systems. However, the problem of formalising the connection between validated models and implementations remains.

The experience gained during this project can best be summarised as a set of guidelines that may help to simplify the development of other reactive systems. The state explosion problem remains the most important limitation of computer-aided validation. The best way to reduce the size of the generated state space is to know what to avoid. In [6, Chapter 2] Holzmann describes several general rules of design that reflect a considerable amount of experience in the design of computer protocol models. In addition, the following more specific modelling guidelines can be added:

- Keep the number of concurrent processes to the absolute minimum. The size of the reachable state space increases dramatically (roughly exponentially) as the number of concurrent processes is increased.
- Include only data structures that can influence the future responses of a model. Actions that are not visible across interfaces should be eliminated to reduce the number of state generated.
- Eliminate buffers if possible or keep their size to the absolute minimum required for correct operation.
- Beware of seemingly simple processes that generate huge state spaces through manipulation of data variables. A clock or counter that increments a variable is the typical example to be avoided.
- Use nondeterminism to model a decision structure when the alternative actions are important but the decision mechanism is irrelevant.
- Divide complex designs into clearly defined subsystems that can be validated separately.

It can be concluded that computer-aided validation as a technique is sufficiently well developed to be applied in industry to check designs of reactive systems for subtle errors. However, several areas are still subject to research. One idea is to try and extend the techniques to handle systems with an infinite number of states. Another important research topic is to find more effective state compaction techniques to increase the number of states that can be stored in a given amount of memory. In this respect recent work done by Holzmann shows a rich potential to be explored [7].

8 Acknowledgement

I thank Werner Fouché and Pieter Muller for writing most of the code to implement the microkernel.

A Model of Communication Subsystem

```
MODEL SSComms;
  CONST nullproc = 0; maxProc = 2;
  TYPE ProcID = 0..maxProc; ProcQ = LIST[maxProc] OF ProcID;
    A0 = {Trans(ProcID), RecReq(ProcID), SendRep(ProcID),
          newProc(ProcID)};
    A1 = {trans, recReq, sendRep, newProc(ProcID)};
    A2 = {enterProc(ProcID), selectProc(ProcID)};
    A3 = {resume(ProcID)};
  VAR ch0: A0; ch1: A1; ch2: A2; ch3: A3;
```

```
PROCESS Client(id: ProcID; IN in: A3; OUT toRunning: A1);
VAR p: ProcID;
BEGIN
  DO TRUE ->
    POLL in?resume(p) & p = id -> SKIP END; toRunning!trans
  END
END Client;

PROCESS Server(id: ProcID; IN in: A3; OUT toRunning: A1);
VAR p: ProcID;
BEGIN
  DO TRUE ->
    POLL in?resume(p) & p = id -> SKIP END;
    toRunning!recReq;
    POLL in?resume(p) & p = id -> SKIP END;
    toRunning!sendRep
  END
END Server;

PROCESS Ready(IN in: A2; OUT toRunning: A1);
VAR p: ProcID; q: ProcQ;
BEGIN
  EMPTY(q); APPEND(q,1); APPEND(q, 2); p := nullproc;
  DO TRUE ->
    POLL in!selectProc(HD(q)) & LEN(q) > 0 ->  REMOVEHD(q)
    [] in!selectProc(nullproc) & LEN(q) = 0 ->  SKIP
    [] in?enterProc(p) ->
        IF p # nullproc -> APPEND(q, p)
        [] p = nullproc -> SKIP
        END
    END
  END
END Ready;

PROCESS Running(IN in: A1; OUT toReady: A2; OUT toUser: A3;
                OUT toComms: A0);
VAR cp: ProcID; (* current process *)
BEGIN
  DO TRUE ->
    toReady?selectProc(cp);
    IF cp # nullproc -> toUser!resume(cp);
      POLL in?trans -> toComms!Trans(cp)
      [] in?recReq -> toComms!RecReq(cp)
      [] in?sendRep -> toComms!SendRep(cp)
      END
    [] cp = nullproc -> SKIP
    END
  END
END Running;

PROCESS Comms(IN in: A0; OUT toReady: A2);
CONST NIL = 0;
TYPE Map = ARRAY[maxProc+1] OF ProcID;
VAR cp, sp: ProcID; pq: ProcQ; clients: BOOLEAN; Partner: Map;
BEGIN clients := FALSE;
  DO TRUE ->
    POLL in?Trans(cp)->
      IF clients OR LEN(pq) = 0-> (* no server is ready *)
          APPEND(pq, cp); clients := TRUE
      [] ~clients & LEN(pq) > 0 -> (* a server is ready *)
          sp := HD(pq); REMOVEHD(pq);
          Partner[cp] := sp;
          Partner[sp] := cp;
          toReady!enterProc(sp)
      END
    [] in?RecReq(sp) ->
      IF clients & LEN(pq) > 0 -> (* a client is ready *)
         cp := HD(pq); REMOVEHD(pq);
         Partner[cp] := sp;
```

```
          Partner[sp] := cp;
          toReady!enterProc(sp)
        [] ~clients OR (LEN(pq) = 0) -> (* no client is ready *)
          APPEND(pq, sp); clients := FALSE
        END
      [] in?SendRep(sp) ->
        cp := Partner[sp]; Partner[sp] := NIL; Partner[cp] := NIL;
        toReady!enterProc(cp); toReady!enterProc(sp)
      END
    END
  END Comms;

BEGIN
  Client(1, ch3, ch1); Server(2, ch3, ch1); Ready(ch2, ch1);
  Running(ch1, ch2, ch3, ch0); Comms(ch0, ch2)
END SSComms;
```

References

[1] T. Cattel. Modelization and Verification of a Multiprocessor Realtime OS Kernel. In *Proceedings 7th International Conference on Formal Description Techniques*, pages 35–50, Bern, Switzerland, October 1994.

[2] E.M. Clarke and E.A. Emerson. Design and Synthesis of Synchronization Skeletons using Branching Time Temporal Logic. In D. Kozen, editor, *Proceedings of IBM Workshop on Logic of Programs*, pages 52–71. Lecture Notes in Computer Science, 131, 1981.

[3] P. de Villiers and W. Visser. ESML—A Validation Language for Concurrent Systems. In Judy Bishop, editor, *7-th Southern African Computer Symposium*, pages 59–64, July 1992.

[4] W. Fouché and P. de Villiers. A Reusable Kernel for the Development of Control Software. In M. Linck, editor, *6-th Southern African Computer Symposium*, pages 83–94, July 1991.

[5] C.A.R. Hoare. *Communicating Sequential Processes.* Prentice-Hall, Englewood Cliffs, New Jersey, 1985.

[6] G.J. Holzmann. *Design and Validation of Computer Protocols.* Prentice-Hall, Englewood Cliffs, New Jersey, 1991.

[7] G.J. Holzmann. State Compression in SPIN: Recursive Indexing and Compression Training Runs. In *Proceedings: SPIN97 Workshop*, Enschede, The Netherlands, April 1997. University of Twente.

[8] Z Manna and A Pnueli. *The Temporal Logic of Reactive and Concurrent Systems.* Springer-Verlag, 1992.

[9] P.J. Muller and P.J.A. de Villiers. Using Oberon to Design a Hierarchy of Extensible Device Drivers. In P. Schulthess, editor, *Proceedings of the Joint Modular Languages Conference, University of Ulm, Germany*, pages 147–158, September 1994.