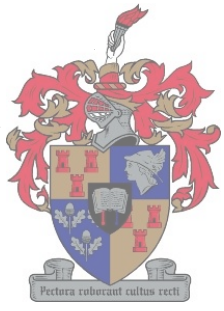
Validation of a Microkernel: a Case Study

A DISSERTATION
SUBMITTED TO THE DEPARTMENT OF COMPUTER SCIENCE
OF THE UNIVERSITY OF STELLENBOSCH
IN PARTIAL FULFILMENT OF THE REQUIREMENTS
FOR THE DEGREE OF
DOCTOR OF PHILOSOPHY



By
PJA de Villiers
November, 1999

Supervised by: Dr GJ Holzmann and Prof AE Krzesinski

Declaration

I the undersigned hereby declare that the work contained in this dissertation is my own original work and has not previously in its entirety or in part been submitted at any university for a degree.

Abstract

This dissertation describes the application of formal methods to the development of operating systems. A related area of software engineering—the development of protocols—has been improved substantially by the application of formal methods. The essential behaviour of a protocol can be checked for correctness by using a verification system. A model of a protocol is developed to capture its most essential characteristics. This model is then checked automatically for correctness properties which are specified as temporal logic formulae.

One of the most successful verification techniques is known as *model checking*. It is a state exploration technique, each state being the values assigned to every variable in a protocol model at a given instant. Although protocols of realistic size generate millions of states, it is possible to check important correctness properties in minutes on a typical workstation.

Broadly speaking, protocols and operating systems are similar in the sense that they are reactive systems. Such systems are designed to interact continually with their environments and usually involve concurrent processing. However, there are important differences between protocols and operating systems. For example, in protocol verification, the focus is on the transmission rules. Data can be represented more abstractly. For operating systems, this is not so. Data structures (such as a scheduler queue) represent the internal state of the system and must be represented in more detail.

To verify a complete operating system is a formidable task. A manageable first step was to select one important component and to investigate the feasibility of applying formal methods to its development. A component that is a basic building block of many modern operating systems is a microkernel. It implements the fundamental abstractions which support the rest of the system.

Instead of using an existing verification system, an experimental verification system

was developed to verify the microkernel. This was done primarily to learn about the techniques involved, but the insight gained about the practical limits of the verification process also helped the the modelling process. Since it was known from the start that the representation of data is important, special care was necessary to store states as compactly as possible.

This case study suggests that the designers of future operating systems can benefit from the use of formal methods. More important than the verification of a specific microkernel is the general approach, which could be used to verify similar systems. The experience gained from this case study is presented as a list of guidelines to reduce the number of states generated. However, many problems remain and these are pointed out as topics for future research.

Opsomming

Hierdie verhandeling beskryf die toepassing van formele metodes om bedryfstelsels te ontwikkel. 'n Verwante gebied—die ontwikkeling van protokolle—is reeds beduidend verbeter deur die gebruik van formele metodes. Die basiese gedrag van 'n protokol kan nagegaan word vir korrektheid deur 'n verikasiestelsel te gebruik. 'n Model van 'n protokol word ontwikkel om die belangrikste gedragspatrone vas te vang. Die model word dan outomaties analiseer om sekere korrektheidseienskappe na te gaan wat gespesifiseer word as formules intemporal logika.

Een van die mees suksesvolle verifikasietegnieke staan bekend as modeltoetsing ("model checking"). Die tegniek behels 'n soektog waar toestande sistematies ondersoek word. Elke toestand is die gesamentlike waardes wat toegewys is aan elke veranderlike van 'n model op 'n gegewe tydstip. Alhoewel protokolle van realistiese grootte miljoene state kan genereer, is dit moontlik om belangrike korrektheidseinskappe na te gaan in enkele minute op 'n tipiese persoonlike rekenaar.

Breedweg is protokolle en bedryfstelsels soortgelyk in die sin dat beide reaktiewe stelsels is. Sulke stelsels word ontwerp vir kontinue interaksie met hulle omgewings en behels gewoonlik gelyklopende verwerking. Nogtans is daar belangrike verskille tussen protokolle en bedryfstelsels. Byvoorbeeld, in die verifikasie van protokolle is die klem op transmissiereëls. Data kan meer abstrak voorgestel word. Vir bedryfstelsels is dit nie so nie. Datastrukture (soos 'n skeduleerdertou) verteenwoordig die interne toestand van die stelsel en moet in meer detail voorgestel word.

Om 'n volledige bedryfstelsel te verifieer is 'n geweldige taak. As 'n hanteerbare eerste stap is een belangrike komponent geselekteer. Die doel was om vas te stel of dit haalbaar sou wees om formele metodes te gebruik om so 'n komponent te ontwikkel. 'n Mikrokern is 'n 'n basiese boublok van baie moderne bedryfstelsels. Dit implementeer die fundamentele abstraksies wat die res van die stelsel ondersteun.

In plaas daarvan om 'n bestaande verifikasiestelsel te gebruik, is 'n eksperimentele stelsel ontwikkel om die mikrokern te verifieer. Dit is gedoen hoofsaaklik om meer te leer van die betrokke tegnieke, maar die insig wat bekom is aangaande die praktiese beperkings van die verifikasieproses het ook gehelp tydens modellering. Omdat dit bekend was dat die voorstelling van data belangrik is, is spesiale aandag geskenk aan die kompakte voorstelling van toestande.

Die gevallestudie toon aan dat die ontwerpers van toekomstige bedryfstelsels voordeel kan trek uit die gebruik van formele metodes. Belangriker as die verifikasie van 'n spesifieke mikrokern, is die algemene benadering wat moontlik aangewend kan word om soortgelyke stelsels te verifieer. Die ondervinding wat opgedoen is tydens die projek word aangebied as 'n lys van riglyne om die aantal toestande wat genereer word te verminder. Heelwat probleme bly nog oor en word uitgewys as onderwerpe vir verdere navorsing.

Acknowledgements

I gladly acknowledge the help of several people who made this project feasible:

- My supervisor, Gerard Holzmann, who kept asking the right questions, always responding immediately when I asked for help.
- My co-supervisor and colleague, Tony Krzesinski, for encouragement and for supporting the project, even though verification is not his field of research.
- Chris Brink, who introduced me to temporal logic.
- Several of my former students, each of whom contributed to the project in some way: Werner Fouché, who implemented the first version of the kernel; Harry Lewis, who added several device drivers to make the kernel industrially viable; Willem Visser, who wrote the first ESML compiler; Pieter Muller, who rewrote the kernel in Oberon and Jaco Geldenhuys who improved the ESML verifier and measured its performance.

Contents

Chapter 1

Introduction

The development of a reliable operating system is a challenging undertaking. A substantial amount of effort is required to design, implement and test such a complex software system. A major problem is that many design errors will only show up later during testing. Even then, testing alone presents no guarantee that all errors have been eliminated. To address this problem, computer-aided verification techniques are being developed to pinpoint errors as early as possible—at the design stage.

A convincing demonstration of the benefits obtainable from formal verification techniques can be found in the field of protocol engineering. Protocols of practical size and complexity can be checked automatically to find subtle design errors. A modelling language is used to develop models that capture the essential characteristics of a protocol and a verification system is then used to confirm that such models satisfy various correctness claims. It is a new challenge to apply such techniques to operating systems. It is my thesis that the ever increasing demands placed on the reliability of operating system software can and will be met by the use of computer-aided formal verification techniques.

To support my thesis, I made the following contributions:

- I demonstrated that the design of a non-trivial microkernel—one of the more complex building blocks of an operating system—could be verified to show that it satisfies formally specified correctness claims.
- I showed how a microkernel can be designed to allow separate verification of its principle components. The challenge was to do this without making the kernel

grossly inefficient. Such a design strategy is essential, because a monolithic design would be beyond the capabilities of currently available verification techniques.

- As part of the experiment, a practical verifier and a suitable modelling language were developed.
- Since the verification of a particular microkernel is not of much general interest, the experience gained during the project is presented as a set of design rules to serve as guidelines in similar application areas.

1.1 Overview

In what follows, I explain in detail how a microkernel was designed and verified. A microkernel is an example of a reactive program. Such programs continually interact with their environments and do not compute a final result on termination. The development of reactive programs is considered to be particularly challenging because such programs must be designed to respond to events that may occur at unpredictable times. Moreover, such programs usually involve concurrency. This means that execution sequences that lead to error states are hard to reproduce, which makes testing difficult. In fact, verification is considered essential because the kind of errors usually found in such designs are often of an extremely subtle nature.

Although research in protocol engineering offers much inspiration and useful guidelines, operating systems present a different and new challenge. It seems feasible to use formal verification techniques to develop fundamental components and then combine these "trusted building blocks" to create reliable systems. Although this strategy seems attractive, a simple-minded combination of reliable components does not necessarily lead to a reliable whole. For example, a system may deadlock even when its components are all deadlock free. A complex design should therefore be studied at various levels of abstraction to analyse all possible patterns of interaction.

Chapter 2 provides some background on microkernels in general and the specific microkernel that was developed as a case study. A microkernel controls the hardware, providing basic memory management, processor management, interrupt handling, interprocess communication and sometimes drivers for peripheral devices. Microkernels can be reused to support different operating systems and this alone seems to justify the investment of extra effort in their development. In addition, a microkernel should

be highly reliable because it forms the foundation that supports everything else.

Chapter 3 presents a selective overview of various protocol verification techniques. Against this background, the problems associated with verifying a microkernel stand out. The main challenge was to devise a compact representation technique to allow verification of the complex data structures encountered in microkernels.

In Chapter 4 the main issues involved in designing a computer-aided verification system are discussed. Two notations are needed: a modelling language to describe systems, and a temporal logic to describe correctness claims about the systems being modelled. An efficient verification algorithm is used to check whether a given correctness claim is satisfied by a model.

Chapter 5 describes the main experimental tool: a verification system for ESML—a modelling language designed with microkernels in mind. The temporal logic CTL was adopted to specify correctness requirements as part of the experiment. It turned out that the logic LTL would have been a better choice because verification algorithms for CTL require a substantial amount of memory. The selection of the specific verification algorithms used in the verifier is motivated and areas are pointed out where the implementation could be improved. Some performance measurements are included to indicate the application limits of the new tool. The motivation for developing a new verifier was mainly to gain experience of the practical limits of different techniques. This provided the kind of insight needed to evaluate the practicality of computer-aided verification techniques.

Chapter 6 contains the major new results: various techniques are proposed to provide a formal development strategy, not only for microkernels, but also for similar operating system components. The main idea is to design complex reactive systems as a set of interacting servers. The advantage is that servers can be isolated and verified separately by providing drivers and stubs to replace their more detailed counterparts. These techniques were applied step by step to develop verified models of a non-trivial microkernel. It is explained how a microkernel can be designed as such a set of interacting servers.

Chapter 7 is retrospective and presents the experience gained during the experiment. General guidelines are given that may help to simplify the development of similar reactive systems. However, some problems remain and these are pointed out as topics for future research.

Chapter 2

Microkernels

This chapter presents a brief introduction to microkernels in general, followed by an overview of the specific microkernel that was developed as a case study. This background is essential to draw up a list of requirements to be met by a suitable verification system.

2.1 What is a microkernel?

A microkernel is an intricate piece of software that implements the most basic functionality of an operating system. It provides only those services that are difficult or expensive to implement otherwise. As explained by Tanenbaum, these services include basic memory management, process management, interprocess communication and sometimes device drivers [66]. The rest of the functionality of a microkernel-based operating system is provided by server processes running in user space.

The advantages of designing operating systems around a suitable microkernel have been illustrated in various research environments [13, 62, 38, 68, 67]. In older monolithic operating systems all services were linked together. This means that services like device drivers cannot be removed or replaced without stopping the system. Some more recent systems (like Linux, FreeBSD and Windows NT) allow device drivers to be loaded at run time. Microkernel-based systems are even more flexible. A service is defined by its interface and new services can be added while the system is running. This allows a system to evolve naturally as older components are systematically replaced by improved versions. By hiding hardware-specific code inside a microkernel, the bulk of

an operating system can be made highly portable. Moreover, a microkernel presents a well-defined model of the hardware to an operating system designer who is encouraged to think in terms of abstract concepts instead of hardware-specific detail. Well-chosen abstractions can be guaranteed to satisfy a set of correctness properties. Abstraction forms the basis of modularisation, without which reliability is almost unattainable. A microkernel therefore creates a framework for the design of reliable operating systems composed of specialised interacting components.

The main abstractions created by a microkernel are *processes* and *communication channels*. Because it supports communication channels between interacting processes, a microkernel has been called a "software backplane" [23]. Unbuffered synchronous (blocking) message passing is the preferred method of interprocess communication because this allows particularly simple and efficient implementations. Debugging can be simplified by keeping each process in a separate address space. Unfortunately this slows down interprocess communication because of the inherent overhead of moving data from one address space to another. Many microkernels therefore allow several processes to share the same address space. This allows data to be exchanged via shared memory. Such "multi-threaded" servers are efficient, but are more difficult to debug.

2.2 A case study: the Gneiss microkernel

A design that is divided into a number of clearly defined subsystems with simple interfaces is simpler to verify than a monolithic design. This is so because an abstract model can be used to study the interaction of the different subsystems and more detailed models can be used to study each subsystem on its own. The reachable state space for a monolithic design that includes a reasonable amount of detail can be enormous.

It was therefore considered essential to have full control over the design of the microkernel to be used as a case study. Attempting to verify some well-known microkernel (without modifying it) was considered too ambitious, although such a project would probably be of more interest to the operating systems community. It was decided to concentrate on correctness and to rely on well-known concepts found in efficient microkernel-based systems such as Amoeba [55, 70, 54] and V [13, 11, 12].

The Gneiss microkernel was selected as a case study. A first version of the kernel was designed by the author [30]. The kernel was later rewritten in Oberon while keeping verification firmly in mind. The original design was retained, however. The basic ideas

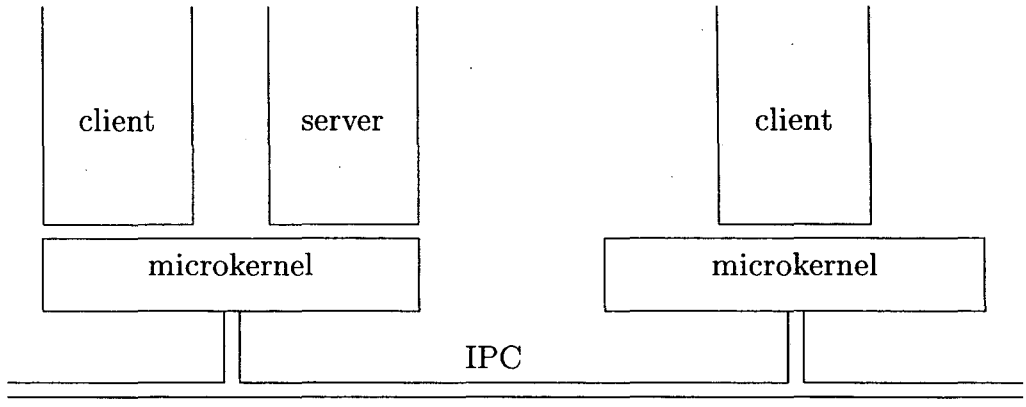


Figure 1: Distributed system based on the Gneiss microkernel

incorporated in the Gneiss kernel are illustrated in Figure 1.

Two physical machines are shown, each running a copy of the kernel. The physical machines are connected via a standard local area network to create a distributed architecture. The key abstractions created and managed by the Gneiss kernel are virtual machines (VMs), processes, and ports. The physical machine on the left of Figure 1 supports two virtual machines (VMs), one implementing a client application and the other a server. The physical machine on the right supports a single VM which runs a client application. Each VM has a separate protected address space and can execute one or more processes that may communicate via ports or shared memory. VMs and processes are scheduled by the kernel. The idea is that each VM will support a single well-defined component of an operating system. A VM may support several cooperating processes, all sharing the same address space. A process can be created about ten times faster than a VM and shared memory techniques may be used when necessary to increase the speed of interprocess communication within the same VM. Processes sharing the same address space are called "light-weight" processes to distinguish them from "heavy-weight" processes found in traditional systems such as Unix. In other systems, light-weight processes are sometimes called "threads".

2.2.1 Design Overview

An informal overview of the Gneiss kernel is presented here. The detail is left until Chapter 6 where models of different kernel components are described. These models are verified to show that they satisfy certain correctness claims. The verified models

can be found in Appendix A.

To help maintain system integrity, the kernel prevents one VM from accessing the address space of another. This makes it impossible for any VM to accidentally (or maliciously) modify the memory area of another. Hardware support for memory protection helps to guarantee this. VMs and processes are represented in the kernel by records that are used to store their current state. VMs that can be activated await their turn in a number of queues, one for each priority level. Each VM record has an associated queue of process records (the VM's process queue) to represent the processes that are ready to run.

The scheduler selects the highest priority VM and activates the first process in its process queue. VMs are time-sliced, but a process is allowed to run until it invokes an interprocess communication (IPC) operation or a higher-priority VM becomes executable. When a specific VM is suspended because something more important demands attention, it is guaranteed that the interrupted process will be resumed immediately when the VM is reactivated later. In other words, processes supported by each VM are executed non-preemptively. The only way to switch from one process to another in the same VM is to execute a kernel call. This ensures that when a process updates a shared data structure, no other process can interfere. No other synchronising primitives such as semaphores or condition variables are needed.

Processes in *different* VMs can only communicate by executing kernel calls. There are only three IPC primitives. A server process executes a *ReceiveRequest* to wait for a request message. A client issues a request by executing a *Transaction* primitive. After receiving a message, a server proceeds to deliver the requested service. The client process is blocked until the server executes a *SendReply* to return the result of the requested operation. Processes are not addressed directly; *ports* are used to provide an extra level of indirection. A timeout mechanism ensures that all transactions terminate eventually—either successfully or not. A status code is returned to indicate the outcome of each request.

Peripheral devices are controlled by kernel-resident servers with specific reserved ports. To issue a request to a peripheral device, a client process executes a *Transaction* operation on the port associated with the appropriate device driver. A request queue may be maintained by a device driver in the form of a linked list of client processes waiting for requests to be completed. On completion of a peripheral operation (when the completion interrupt arrives), the waiting client process is activated by returning

its process record to the scheduler. The next peripheral operation in the request queue (if one is available) is started immediately and then the scheduler is invoked to select a new process to activate.

It can thus be seen that interrupt handling, process scheduling and message passing are intertwined. Fortunately, this need not concern a programmer at the VM level who is presented with a well-defined abstract machine that is much easier to program than the underlying physical machine. To summarise, a VM is an abstract machine with the following properties:

- It has a separate protected address space which encapsulates code and data.
- A VM executes one or more processes that share the same address space.
- Processes interact by executing three primitive IPC operations: *Transaction*, *ReceiveRequest* and *SendReply*. A valid port must be specified with each IPC operation. Processes in the same VM may communicate via shared memory where speed is crucial.
- A VM allows only one process to execute at a time. A process switch may occur only when an IPC primitive is executed.

2.3 Verifying a microkernel

A model of a system is verified by proving that it satisfies certain formally stated correctness criteria. It is desirable (and probably less expensive) to develop models of a system with the goal of detecting errors before any implementation code is produced. However, although many errors of a purely logical nature can be eliminated through verification of a design, the devil can still hide in the details. When a verified model is translated into an implementation, errors may be introduced. For example, certain low-level operations that interact with the hardware are required in an operating system. The nature of such operations is usually simple, but the wrong bit may be specified, for example, or the initialisation sequence for a controller card may be incorrect. In a model the detail of such low-level operations is suppressed by representing then as simple atomic events. In practice each such low-level operation should be tested on its own. Fortunately, it is fairly simple to isolate and correct such coding errors. A systematic approach should be followed though to ensure that all such low-level operations are tested.

To verify a microkernel, two separate formal descriptions are essential: a specification of the kernel's intended behaviour, and a description of various correctness requirements. Two different notations are needed: a modelling language to describe the intended behaviour of the kernel, and a notation that allows various correctness claims to be expressed concisely. A model of a system is an executable abstract program while a correctness claim is usually a formula expressed in a suitable temporal logic.

2.3.1 Behavioural Specifications

A number of modelling languages have been developed that are loosely based on the principles of CSP [41]. The complexity of a verifier is influenced by various properties of the modelling language. Instead of implementing a plethora of specialised concepts, the goal should be to find a small number of general constructs that can be used to model most implementations in a natural way. The following concepts are especially useful:

- **Concurrency.** An intrinsic property of reactive systems is that a number of processes are executed concurrently. A process oriented language therefore forms a natural framework for expressing concurrent execution. The interleaving model of concurrency—where only one process may execute a basic atomic operation at a time—has been accepted widely as a basis for verification systems. A good explanation of how concurrent systems can be modelled by using interleaving can be found in [52]. SPIN is a well-known verification tool that is based on interleaving of the actions of concurrent processes [44]. The choice of which process to select when there are alternatives, raises the issue of fairness, as will be discussed in Chapter 5. Concurrent systems can be designed in two different ways: based on shared memory and based on message passing [2]. Both approaches can be modelled using a process oriented modelling language. Processes are also useful in other ways. No procedures and functions are strictly necessary since these can be modelled as processes. Even objects (in a limited sense) may be modelled as processes.

- **Messages.** It was decided to accept the principles of CSP as a design framework. This means that processes exchange data through unbuffered, synchronous message passing. The management of buffer spaces presents an additional problem to the kernel designer and as illustrated by the V microkernel synchronous message

passing can be quite efficient [11].

- **Nondeterminism.** This is a fundamental concept that is useful in a modelling language. Nondeterminism, especially as part of control structures such as alternative commands, is useful to suppress irrelevant detail.

- **Interfaces.** At the highest level, the structure of a design is determined by the way its various processes interact. This is reflected by the interfaces between processes. Channels are provided by CSP to define interfaces between processes and this should be accommodated by the modelling language. In addition, the messages that may be transmitted over every channel should be defined explicitly. This facility enables a compiler to detect many design errors at an early stage of development.

- **Strong typing.** Compile-time type checks are valuable for the development of reliable software. Serious attention to the definition of fundamental types in a system usually pays off at a later stage by improving clarity. Carelessness in this regard soon leads to awkward type conversions in the code. It therefore seems worthwhile to do it right from the start by insisting that the modelling language should be strongly typed. This allows type conflicts to be detected in the least expensive way: before any implementation code is written.

Above all, the modelling language should be designed to make efficient, automatic verification possible. At the same time, it should be close enough to a modern implementation language to make the implementation task fairly mechanical. In fact, automatic generation of (some) implementations should be possible. The notation should be readable and natural to allow one to focus on concepts rather than distracting detail.

At the start of this project, it was unknown whether a special modelling language and verifier would be required to verify a microkernel or operating systems in general. However, it was clear that the representation of rather complex data would have to be addressed because the behaviour of a microkernel is governed by internal data structures that must be modelled in some detail. The most basic data structuring concept that can be included in a modelling language is arrays—a useful device for modelling tables and other mappings. However, reactive systems like a microkernel include data structures that are awkward to model as arrays. A scheduler queue is a good example. The behaviour of a scheduler depends on a queue of process records. To model queues as arrays requires explicit index manipulation and modulo arithmetic. A more abstract

structuring concept is clearly needed. Sequences form part of specification languages such as Z [64] and can model queues and stacks in a natural way. To model process records (that may contain fields of different types) record structures are needed. It was therefore decided to include arrays, sequences and records. The design of a verifier is affected by these data structuring facilities. Data must be represented as compactly as possible to avoid an explosion in the size of states. Furthermore, such compaction must not be prohibitively expensive in terms of processing time.

Fortunately, microkernels do not need the sophisticated communication techniques supported by some protocol verifiers. Since buffered message passing is not needed, channels are not part of the global system state. This accounts for a significant reduction in state size.

2.3.2 Correctness requirements

Correctness claims to be verified must be expressed in a formal notation. Temporal logic is usually used for this purpose and the properties that can be specified depend on the logic. The verification problem is only tractable for suitably restricted temporal logics and therefore only certain relatively simple properties can be verified efficiently for large models.

Fortunately, even the simplest correctness checks are valuable. Freedom from deadlock is such a property that is straightforward to check. The cause of a deadlock is often a structural clash in the synchronisation skeleton of a design and this should be corrected as soon as possible. Therefore freedom from deadlock should be verified at an early stage. Only a rather abstract model of the entire system is needed to detect deadlocks that may easily go undetected otherwise. It is preferable to start with such an abstract model to arrive at a skeletal design for the entire system that is free from deadlock. The alternative bottom-up approach—to develop detailed subsystems and integrate them later—often leads to structural clash.

Freedom from deadlock is an example of a system invariant—a property that should never be violated. Any reactive system should maintain certain invariants. An invariant property is true after initialisation and is supposed to remain true thereafter. Another example is proper buffer management. During overload conditions buffer space may be depleted and some mechanism should be provided to handle such situations. In general, inspection of a design cannot determine reliably whether such mechanisms

will function as intended. Unfortunately such errors often occur in practice and seldom reveal themselves until a system has been in use for some time. Verification can be used to eliminate such errors as early as possible—at the design stage. The class of so-called *invariant properties* are expressed in the form "it will always be the case that P holds", P being a function that can be evaluated in every state of a model.

The verification models considered in this dissertation consist of two components: a system model and a model of its environment. Another class of correctness properties known as *response properties* is particularly relevant to such models. If the interface between the system model and its environment is defined precisely, it is possible to show that the system model will respond to all external events caused by its environment in the intended way. In the case of the Gneiss microkernel this interface was deliberately designed to be as simple as possible. Three kernel calls and hardware interrupts are the only possible external events. An important property to verify is therefore that the kernel model always responds correctly to kernel calls and interrupts. Such properties are expressed in the form "if event e_1 happens then event e_2 will eventually happen".

A third kind of property that can be verified efficiently is called *precedence properties*. The idea is to prove that event e_1 will always happen before e_2. Precedence properties can be expressed in different ways, with subtle differences, but a useful format is "condition c_1 will hold until condition c_2 becomes true". For example, it may be important to verify that if a message is received it will not be discarded until it has been processed. This can be expressed as a precedence property.

Even if the interface between a microkernel and its environment is simple, a huge number of internal actions are possible, especially if the kernel is designed to be interruptible. Fortunately, a microkernel can be divided into subsystems that can be modelled separately. In the presence of concurrency, the interaction between different components is a common source of subtle design errors. More detailed models of the process management policy, interprocess communication and device drivers are essential to verify these mechanisms.

Process management includes scheduling. A set of rules are encoded in the scheduler model to determine the next process to run. If the rules are wrong the scheduling policy may be violated. This may allow the system to work, although inefficiently. Verification techniques can detect this kind of error when a scheduler is designed. Interprocess communication in a microkernel can be modelled as a protocol. Properties that can be verified are typical for protocols. Device drivers are cyclic routines that interact

closely with the hardware that controls peripheral devices, the interrupt system and the scheduler. It would be valuable to verify that every request is eventually satisfied, that no interrupts are ignored forever and that requests are serviced in the intended order.

2.3.3 Characteristics of a suitable verifier

The kind of verification system needed to verify a microkernel depends on the functionality of the kernel and the kind of correctness claims to be checked. Building a detailed model of a complete microkernel does not make sense for the following reason: a microkernel is a complex piece of software and should be designed by dividing it into functionally coherent components that can be modelled separately. It is natural to develop an abstract representation of the kernel's structure and more detailed models of every subsystem. An abstract model of the complete kernel is useful to determine an efficient structure and the models of subsystems can be detailed enough to serve as blueprints for coding. However, this approach leaves the designer with an important but non-trivial problem to solve. It must be possible to analyse every submodel on its own and then combine the verified submodels, knowing that the sum of the parts will constitute a correctly functioning whole. To ensure that the various submodels fit together, a formal compositional technique is essential, based on a suitable formal structure.

Considering the kind of structures found in a microkernel, the verification system should be designed to cope with the following:

- It must be able to handle complex data structures such as arbitrary combinations of arrays, records and sequences. This means larger states. To avoid using an excessive amount of memory, effective state compaction techniques are required. Compile time compaction techniques should be used wherever possible, because run time compaction is expensive.
- The average number of states processed per second determines the size of model that can be analysed in an acceptable time. Suitably detailed models to serve as guidelines for coding invariably generate huge state spaces. Efficiency is therefore important.
- When a verification run detects an error, a facility is needed to help the designer

to find the problem. Both an interactive simulator and a tool to inspect error trails are useful.

- Some form of fairness should be supported to avoid reporting "artificial" errors caused by cycling continuously through the same states when an alternative execution path exists.

- A verifier is a complex program that can contain errors. There are two possibilities: a model may be rejected erroneously or it may be accepted when it contains an error. The second possibility is more troublesome. Although verifiers are sequential programs similar to compilers or interpreters, current verifiers are not yet as well-structured and reliable as the best compilers. More research is required to compare different ways of structuring a verifier to improve efficiency and reliability.

- Depending on how many states are generated, special techniques may be needed to reduce the state explosion. There are several options: partial order techniques [46, 35, 37] restrict the search by avoiding execution paths that cannot influence the results. For really large models, controlled partial search techniques [43, 44] can be used. Alternatively, techniques based on binary decision diagrams [9] have been reported to yield impressive results for some very large models by storing states implicitly rather than explicitly.

2.4 Summary: project goals

To illustrate that computer-aided verification techniques can be useful for developing operating system software such as a microkernel, the following goals were set:

- The selection or development of a specification language that allows description of the behaviour of a microkernel. It should be possible to model the design of the kernel in sufficient detail to serve as an outline for an efficient implementation.

- Selection or development of a formal notation to specify correctness properties that should be satisfied by the kernel model. A successful notation would be a compromise: it should be expressive enough to specify important kernel properties, but not overly complex, because that would make efficient verification impossible.

- A verifier should be selected or developed that can check automatically whether a design satisfies the stated correctness criteria.
- Models at various levels of abstraction should be developed of a practical microkernel. These models should be verified to satisfy important correctness criteria.
- A running version of the kernel should be produced and shown to conform to the verified models.
- The experience gained during the project should be summarised in the form of a set of guidelines. This may serve as a starting point for other developments in related fields.

Chapter 3

Automated verification

This chapter presents a selected overview of computer-aided verification. The intention is not to cover all literature on the subject, but rather to provide a brief overview of research that led directly to the techniques presented in this dissertation.

Broadly speaking, most computer-aided verification systems are based on either automated theorem proving or state exploration. A theorem prover can be used to check non-trivial correctness properties of system designs. Expert knowledge is needed though. Even the most advanced theorem provers currently available require a considerable amount of user interaction. Non-trivial problems must be divided into subproblems that are manageable and a substantial amount of effort is often required to find the "right approach". Theorem provers also do not assist designers to pinpoint errors in *incorrect* designs. It would therefore be difficult to persuade most designers of reactive systems to use a theorem prover to check designs for correctness.

Fortunately, verification tools based on state exploration are easier to use. These tools require no user interaction and can detect many commonly occurring errors in system designs. Although the size of systems that can be analysed is limited (realistic designs usually generate a huge number of states), substantial progress has been made to reduce the impact of this so-called state explosion problem. Currently a few million states can be analysed in minutes.

The rest of this chapter is about state exploration. Section 3.1 provides a brief overview of reachability analysis techniques. Section 3.2 presents the idea of model checking that was introduced by Clarke and Emerson. By combining techniques from reachability

analysis and model checking, more powerful verification techniques have been developed. A selection of these techniques are described in Section 3.3. Finally, by comparing different options, the strategy that was selected to verify the Gneiss microkernel is motivated in Section 3.4.

3.1 Reachability analysis

Work on automated protocol verification started in the late seventies. A compact historical overview can be found in Chapter 11 (bibliographic notes) of [44]. The original idea was to apply *reachability analysis* to check a protocol against a list of commonly occurring errors. The properties to be verified were formulated as reachability assertions. At first all unique states were stored in memory, but it was soon realised that larger protocols could be analysed by generating states *dynamically* during a *depth-first* search process. A depth-first search is better than a breadth-first search because less memory is required. This is so because the depth of a model's execution tree tends to increase much slower than its width as new states are generated. Unfortunately it is not enough to store only the current path because an excessive amount of unnecessary work will have to be done. To prevent this, more memory is needed to store all unique states as they are generated. The idea is to detect revisited states as will be explained in Chapter 5 (Section 5.1). It follows that the practical limit of a full state space search is determined by the amount of memory that is available.

There are basically two ways to tackle larger models: (1) a limited amount of work can be repeated by discarding some of the states stored, or (2) some execution sequences can be truncated. The first option is to store all unique states in a cache and to discard some states as the cache fills up. Up to a point, this works reasonably well, but it is difficult to decide which states should be discarded to make room for new states. If states that have been discarded are revisited, the runtime increases. This can be tolerated to a certain extent, but to handle really large models the second option is currently the only viable alternative. Obviously, this means that some errors may be missed. A controlled partial search means that we can decide which states to ignore. Various techniques have been investigated, such as setting a depth bound on the stack or favouring transitions that are likely to lead to error states. It is difficult, however, to find guidelines that will work well in general.

3.2 Model checking

In the early eighties Clarke and Emerson first used the term *model checking* [14, 15] to denote an automatic verification technique (based on state exploration) that uses temporal logic to allow greater flexibility in specifying correctness claims. The global state graph of a system (representing a Kripke structure—a model) is explored to check whether it satisfies a given temporal logic formula that represents a correctness claim. The advantage is that a verification system that is based on model checking enables a user to verify any correctness claim expressible by the chosen temporal logic. The *ad hoc* methods previously used in protocol verification systems to express correctness claims were therefore no longer needed, at least in principle. However, the earliest model checking systems were severely limited in their capability to explore large state spaces.

At about the same time Quielle and Sifakis independently developed a similar verification technique, although it was not called model checking [60, 61]. Both these techniques were implemented using CSP to specify behaviour. Both used branching time temporal logic to specify correctness claims. The issue of fairness in the underlying transition systems was treated differently though. Clarke and Emerson defined a fair version of CTL (Computation Tree Logic)—their chosen temporal logic—that restricted the temporal operators of CTL to range over fair paths. Queille and Sifakis took a different approach to fairness by using special temporal operators to indicate conditions that should hold along fair paths. Since these early beginnings, CTL was studied extensively. The logic combines linear time and branching time operators and was designed to express an important class of correctness claims while allowing efficient verification. Linear time operators are quantifiers over states along a given execution path, while branching time operators quantify over paths.

The time required to verify a temporal logic formula depends on the number of reachable states and the formula size (the number of operators contained in the formula). For CTL the time complexity of the model checking algorithm is linear in both the size of the state space and the formula size. For other less restrictive logics, model checking is more expensive in general. For example, the well-known logic LTL (developed from the logic DX described in [58]) requires time linear in the state space size, but exponential in the size of the formula. However, in practice this exponential blowup is not too important because typical correctness claims are expressible as LTL formulae that contain only a few operators. Whether linear time or branching time is best [49, 59, 25, 27] has

not been resolved yet. Emerson and Lei pointed out that the question is not whether linear or branching time is best, but which modalities to use [27]. However, the issue remains controversial. It is often argued that most correctness claims can be expressed as linear time formulae and that path quantifiers are not needed. However, cases exist where a branching time approach has merit. For example, a branching time formula is needed to specify that a reactive system can recover from failures. Such a formula would state that along all paths, in every state, a path exists that leads to the initial state.

3.3 Towards practical verifiers

The early model checkers could not verify systems of practical size, the main problem being that the entire state space was stored in main memory. To address this problem, the better developed techniques of pure reachability analysis were combined with the greater flexibility of model checking. Various new model checking algorithms were developed that generate states dynamically. A depth-first search is executed and only the current execution path is stored in memory. Unfortunately, many paths may lead to the same state, causing unnecessary work to be done. One solution is to use all available memory to store a large cache of states. If a state is revisited and it is in the cache, it can be ignored. Related algorithms based on this idea became known under names such as "on-line model checking" [47] and "on-the-fly verification" [28]. Holzmann developed the first practically useful implementation of a model checker for full LTL that incorporates fairness and efficient on-the-fly state exploration techniques. The system, which is known as SPIN, is described in [44], the first book to explain how computer protocols of realistic size and complexity can be designed by using computer-aided verification.

Research that placed model checking on a firm foundation focused on several important issues: the expressive power of various temporal logics, formalising the model checking problem, fairness, and new algorithms and reduction techniques to combat the fundamental state explosion problem. The logics CTL, CTL* and LTL received much attention. A decision procedure for satisfiability of CTL formulae is given in [24] and a complete axiomatic basis is described in [26]. Manna and Pnueli made a substantial contribution towards the specification of program properties using the logic LTL. In [52] it is described how various modalities can be used to capture different correctness properties and a classification of such properties is given.

The model checking problem was first formalised in model-theoretic terms. The semantics of a temporal logic is usually described in terms of Kripke structures. How this can be done for the logic CTL*, which includes the sublogics LTL and CTL, is shown in [4]. The model checking problem is to determine automatically whether a given finite transition system satisfies a correctness claim—represented by some temporal logic formula—by checking whether the Kripke structure defined by the transition system is a model of the formula. A distinction is sometimes made between *global* and *local* model checking. In global model checking the validity of a formula is computed in all reachable states whereas in local model checking the validity of a formula is computed with respect to a specific state [72]. In local model checking states are generated only if they are needed to compute the truth value of a given formula. In this way much of the unnecessary work performed during global model checking is avoided.

Vardi and Wolper showed that the model checking problem can also be formalised by using automata theory [71]. The idea is to represent the correctness claim as well as the reachable state space as automata. An explanation of this approach can be found in [17]. The synchronous product of the two automata (one representing the *negation* of the correctness claim and the other the reachable state space) is computed on-the-fly. If the product of the two automata is empty, the correctness claim is satisfied. Although explicit construction of automata that represent correctness claims is somewhat counter-intuitive, it is possible to translate linear temporal logic formulae automatically into automata suitable for this kind of analysis.

Various techniques and algorithms were developed, all aiming to reduce the memory requirements and to speed up the state exploration process. A technique that led to a significant improvement in both areas is due to Holzmann [43]. It is basically an ingenious way of keeping track of revisited states. The state value is hashed into an index in a large array of bits. Initially all bits in the array are set to 0 and when a new state is generated, the corresponding bit is set to 1. For a large enough array the probability is low that two different states will be hashed to the same bit. Since such an undetected hash collision will cause some states to be missed, the technique cannot be guaranteed to detect all errors in a model, although it reduces the memory requirements.

Although the global model checking algorithm for CTL was published in the early eighties, the on-the-fly algorithms for CTL were developed much later. The first constructive proof of the correctness of such an algorithm is presented in [72]. The algorithm uses

an efficient depth-first search, but it is not shown how to handle fairness—an essential issue when designing a practically useful verification tool. Another efficient on-the-fly algorithm that can be used to verify CTL and CTL* formulae is given in [4]. This algorithm is quite intricate when compared to the classical model checking algorithm as described in [16], but the time complexity is nonetheless the same. Unfortunately, a substantial amount of memory is needed. Partial graphs must be constructed (and stored in memory) to identify strongly connected components and a number of additional data structures are needed to record the truth values of assertions up to the current point in the exploration process. Similar model checking algorithms exist for other logics. An efficient on-the-fly algorithm for LTL is given in [28]. The algorithm constructs an automaton that represents all execution sequences satisfying a given LTL formula.

A paper published by Burch *et al.* introduced a new model checking technique that became known as *symbolic model checking* [9]. Both the transition relation and the set of visited states can be represented compactly by using binary decision diagrams (BDDs) [7]. In this way, the effect of executing all enabled transitions in every unique state of a model is represented compactly as a graph. The first step is to generate all states that can be reached from the initial state in a single step. This set of states is then repeatedly expanded in a similar way until no new states can be generated. This process defines a breadth-first search that stops when no new states are generated.

BDD-based techniques have some potential advantages. Several case studies demonstrated that large state spaces can be represented more compactly in this way. In addition, since states are stored implicitly, the various operations on BDDs often have the effect of processing several transitions simultaneously. Unfortunately, there is no guarantee that the symbolic representation of a model will be smaller than an explicitly constructed one. The problem is that no efficient BDD-representation exists for some very basic functions [7]. Moreover, variable ordering has a significant effect on the amount of memory that is required. The performance of a verifier that uses the BDD technique is therefore unpredictable in general, although impressive results have been obtained in several case studies, particularly when applied to hardware problems [8].

3.4 Verifying a microkernel

The fact that formal methods are useful to develop correct protocols suggested that similar techniques could work to verify other reactive systems like microkernels. However, the main purpose of most protocols—to transfer data between communicating processes—is but one of the functions of a microkernel. In addition, microkernels also control the basic hardware of a computing node: the processor(s), main memory, and peripheral devices like disks and communication cards. To model the management functions of a microkernel, sophisticated data structures are needed. Furthermore, it is essential to represent these data structures as compactly as possible. The reason is that complex data structures will increase the state size significantly and since huge numbers of states must be stored, larger states will increase the memory requirements dramatically.

In practice, most software is verified by using on-the-fly techniques. Protocols, in particular, have been an exceptionally fruitful application area. On-the-fly verification is quite efficient: memory is used effectively by storing only the current execution path during a depth-first search. In addition, a number of state space reduction strategies are available that allow partial searches without influencing the verification results. For example, the partial order technique described in [35] can improve efficiency in a significant way. The real limit is the amount of memory available to store states. Compaction techniques can pack more states into available memory. Furthermore, when a model is too large to allow a full state space search, Holzmann's technique of hashing without collision detection still allows excellent coverage.

To verify the Gneiss microkernel the choice was between using on-the-fly or BDD-based techniques. Although BDD techniques can produce impressive results, especially in the field of hardware verification, the memory requirements cannot be *guaranteed* to be less than for on-the-fly techniques. Furthermore, the successful verification system SPIN demonstrates that a combination of state compaction, partial order reduction techniques and controlled partial searches makes possible the construction of a verifier that does not reject models that are too large to handle. The ability to resort to partial analysis when necessary is essential in a practical verification system and it was therefore decided to follow the example set by SPIN.

Chapter 4

Modelling, correctness claims and verification

The verifier described in Chapter 5 was designed from scratch and adapted to the task at hand—to verify the microkernel described in Chapter 2. However, many features of the new verifier can be traced back to other people's work. Several features were adopted from Holzmann's verification system SPIN which has proven its worth in practice. SPIN is probably the most widely used system of its kind. The concepts behind it are described in [44] and the source code, which is freely available, provides implementation detail that can seldom be derived from research papers alone. It would be possible to use SPIN to verify a microkernel, but it was decided to implement a new verifier instead.

In retrospect, this was a good decision. Most programs that are intended for a specific purpose can be optimised. For example, an efficient state compaction technique was developed since it was known from the start that the complex data structures in a microkernel would cause the size of states to become a problem. Nonetheless, the main reason for developing a new verifier was to learn about the techniques involved. When modifying a complex program it is easy to ignore the implementation detail of some components. In this way crucial concepts can escape attention. When developing a new verifier, especially when some new techniques are incorporated, the important issues of the various components stand out much clearer. The lessons learned from this experience are summarised at the end of Chapter 7.

A straightforward recoding exercise would have been pointless, of course, and the opportunity was used to experiment with some alternative techniques. The temporal

logic CTL was used instead of LTL, as used in SPIN. The issue of which logic is best is still controversial. The verification algorithm that was used is a modified version of Vergauwen and Lewi's algorithm [72].

Oberon was chosen as implementation language [74, 50]. The language supports modular software development and has excellent safety features that simplified the coding task. Furthermore, Oberon code is highly portable because the same compiler has been modified to produce efficient code for most popular workstations. The following goals were set for the verification tool (points 1, 2, 3, and 4), the modelling language (points 5, 6 and 7), the case study (point 8), and the temporal logic (point 9). These

1. Full state exploration of all microkernel models should be attempted, if feasible, with controlled partial searches (based on the bit vector technique as implemented in SPIN) as an alternative, if necessary.

2. Compact state representation would be needed to support the relatively complex data structures found in a microkernel. Experiments with suitable data compaction techniques were considered essential to restrict states to reasonable sizes.

3. An interactive facility was considered essential to follow execution paths that lead to error states. Without such a tool, many unproductive hours would be spent on finding errors while developing the kernel models.

4. It was decided to experiment with an alternative structure for verifiers of this kind by identifying and separating the major components. The idea was to make it possible to modify one part of the system without having to understand everything in detail. In addition, the separation of independent issues would make the system simpler to describe and therefore easier to understand.

5. Support for parallelism, in the form of concurrently executing processes, would be essential to model a microkernel. Even when only a single processor is supported, peripheral devices still function as separate concurrent entities. CSP was selected as a concurrency framework for the design of the modelling language. This simplified the design of both the kernel and the verification system.

6. A microkernel must manage several queues of waiting processes. Specialised constructs would be needed to simplify the representation of such complex data and it was decided that the modelling language should support arbitrary combinations of finite sequences, records and arrays.

7. There is no guarantee that real systems will perform as required unless we can establish a formal relationship between verified models and implementations derived from them. This is difficult in general, but to increase our confidence in the correctness of real systems, the model should at least reflect the structure of the implementation to be developed from it. For example, several modern programming languages support modules which are invaluable for partitioning a system into manageable components. The modelling language should therefore support modular design.

8. It was expected that a single model of the entire microkernel would be too complex to handle. Submodels with well-defined interfaces were therefore considered essential.

9. A suitable temporal logic was needed to capture the kind of correctness properties mentioned at the end of Chapter 2. However, to keep the project within reasonable bounds, it was decided to ovoid designing a new logic. Some standard logic, adapted if necessary, would have to suffice.

4.1 Computational model

A verifier must explore all execution paths defined by a model. Models are expressed in a suitable high-level notation that can be translated into a low-level form that is convenient for state exploration. The basic formalism used to represent concurrent designs is a transition system. The definition presented in [52, Chapter 1] is adopted here. A transition system is a 4-tuple (V, S, T, I) with the following components:

- V, a finite set of *state variables*. These include data variables which are modified by executing the transition system, and control variables to indicate the current location(s) of activity. The current state of a transition system is determined by the values of the state variables.

- S, a finite set of (unique) states generated by executing the transition system. A state is a unique assignment of values to the variables in V. A state s satisfies an assertion A if and only if the assignment of values to V makes A true, i.e., $s \models A$.

- T is a finite set of transitions. Each transition represents a basic action that can be executed by the system to generate a (potentially) new state. Each transition has

an associated *enabling condition*—a predicate on the current state. A transition is enabled and can be executed if and only if its enabling condition holds.

- I, the initial state of the transition system.

Only finite transition systems (with a finite number of transitions and a finite number of unique states) are considered here because efficient verification techniques exist for such systems. Many practical systems can be modelled as finite transition systems.

A *path* is a sequence of states $s_0, s_1, \ldots$ such that s_0 is the initial state, and for all $i \geq 0$, s_{i+1} can be generated from s_i by executing some transition $t \in T$. Alternatively, a path can be defined as a sequence of *transitions* that can be executed to generate a given sequence of states. This approach is taken in [3, Chapter 2], but the state-based definition is used here. A state s is *reachable* if and only if some path exists that includes s.

Consider two processes P_0 and P_1 in which P_0 can execute a transition t_0 and P_1 can execute a transition t_1. If both t_0 and t_1 are enabled initially, and they cannot disable each other when executed, two execution sequences are possible: t_0, t_1 and t_1, t_0. In transition systems concurrent execution is represented by *nondeterminism*. A transition system is *nondeterministic* if more than one transition is enabled in some state.

Analysis is simplified by representing concurrency as interleaved execution of nondeterministic transition systems. However, such a model may allow execution sequences that are undesirable. Implicit constraints placed on the behaviour of a transition system correspond to different notions of *fairness*, as will be discussed in Section 4.2.3. Without these constraints, a verifier will report errors that will only occur through unfair selection of simultaneously enabled transitions.

4.2 Temporal logic

Temporal logic is useful to specify correctness requirements of reactive systems. Broadly speaking, a temporal logic extends classical propositional logic by adding certain non-truthfunctional temporal operators, such as *always*, *sometimes*, *next* or *until*. Validity in such a logic is governed by the assumptions made about the nature of time. Since it was decided to model reactive systems as nondeterministic transition systems, it

is natural to think of nondeterminism as various different "futures" that are possible at any given time instant. A branching time temporal logic therefore seems like the natural choice. However, linear temporal logic can also be used to specify correctness properties by assuming that any property must hold for all execution sequences. In [52] Manna and Pnueli describe how linear temporal logic can be used to specify many different correctness properties of reactive systems.

The chosen logic not only determines the kind of correctness claims that can be checked; it also has a marked effect on the efficiency of the verification system. For example, some properties expressible in the logic CTL* (which contains CTL) are costly to check while the subset of properties expressible in CTL can be checked efficiently due to carefully chosen restrictions for combining temporal operators. On the other hand, CTL is less expressive than CTL*. For example, CTL cannot express fairness while the logic CTL* can. The linear time logic LTL (contained in CTL*) can also express fairness because similar rules apply for combining the basic modalities.

It has been argued that whether to use branching time or linear time logic is not the basic issue and that the selected temporal operators are more important [27]. However, branching time logics present additional implementation problems that can be avoided with linear time. As a result it took several years before on-the-fly model checking algorithms were devised for CTL and CTL*.

It was decided to use the temporal logic CTL as first defined in [14] to express correctness claims to be verified for the Gneiss kernel models. The logic seemed expressive enough for the task at hand and the existence of efficient model checking algorithms for CTL was considered a bonus. Time is modelled as a *tree* of discrete time instants, each time instant corresponding to a state in the execution of a transition system.

4.2.1 Basic concepts

The language of classical propositional logic is adopted wholesale. As is customary in modal-type logics it is next assumed that there is a set S of *states* such that every atomic proposition is or is not true at a particular state. The states are related to each other by an *accessibility relation*, R. The set S corresponds to the unique states generated by the transition system that represent the reactive system to be analysed. The relation R is determined by the set of transitions of the transition system.

In modal logics the set of states, together with the accessibility relation and the assignment of truth values to atomic propositions in every state, is called a *Kripke structure.* Some properties are usually ascribed to the accessibility relation and depending on what these are, different modal logics arise. Here the accessibility relation structures the set of states into a tree, the branches representing all possible execution sequences from some initial state. The truth value of a compound formula at a given state may depend on the truth values of some of its subformulae at other states further down the tree. To express this, it is necessary to quantify over the *states* in any particular execution sequence, and also over *execution sequences.* For the first purpose the notation "G" is used for "globally", "F" for "finally" and "X" for "next". For the second purpose "A" is used for "for all" and "E" for "there exists". Finally, these different quantifiers are combined to obtain six different modalities: AG, AF, AX, EG, EF and EX. Thus "$AG\alpha$" would say that formula α is true for all states in every execution sequence, "$AF\alpha$" says that in every execution sequence there is some state in which α is true, "$EX\alpha$" says that α is true in some immediate successor state, and so on.

In the more technical Section 4.2.2 below an *until* operator is also introduced to increase the expressiveness of the language. Roughly, "$\alpha \,\mathcal{U}\, \beta$" is the claim that β will be true at some point in the future, and up to the immediately preceding point α will be true. Again, this may be preceded by A or E, indicating respectively that the claim holds for all or only some execution sequences.

In Section 4.1 the basic concepts of transition systems were defined. In the next section some of these concepts are defined from a logical point of view and finally these concepts are brought together to form the basis of a verification system.

4.2.2 The branching time logic CTL

The alphabet of CTL comprises: a set of variables $P = \{p_1, p_2, \ldots, p_n, \ldots\}$; the constants "true" and "false"; the connectives $\neg$ and $\wedge$; the temporal operators A, E, X and $\mathcal{U}$ with parentheses as punctuation symbols.

Any variable by itself is a formula. If α and β are formulae, then so are:

$$\neg\alpha,\ \alpha \wedge \beta,\ AX\alpha,\ EX\alpha,\ A(\alpha \,\mathcal{U}\, \beta),\ E(\alpha \,\mathcal{U}\, \beta),$$

and nothing else is a formula. To define validity, our assumptions concerning time are packed into the formal definition of a *Kripke structure.* It is a triple $K = (S, R, v)$ such

that:

- S is a finite set, the elements of which are called *states*. There is some distinguished state $s_0 \in S$, called the *initial state*.
- R is an *adjacency relation* over S, such that (S, R) is a *tree*, with root node s_0.
- $v : P \times S \to \{0, 1\}$ is an assignment of some truth value (0 *or* 1) to every variable at every state.

A *path* is a branch of the tree (S, R)—an infinite sequence of states $s_0, s_1, \ldots$ starting with the root node s_0 and such that $(\forall i)[(s_i, s_{i+1}) \in R]$. A path *from* a given state s is a branch of the subtree with s as root.

Note that the structure (S, R) denotes the tree whose nodes are occurrences of unique states in S. That is, if a state is revisited, it will occur once for each time it is reached.

Having assumed an assignment of truth values to every atomic formula at every state, a definition can be given of what it means for any compound formula α to be true at some state s. This is indicated in the usual way by "$s \models \alpha$" and the fact that α is *not* true at s by "$s \not\models \alpha$". Inductively then, for any state s: $s \models \mathit{true}$, and $s \not\models \mathit{false}$; for any atomic formula p, and any state s: $s \models p$ iff $v(p, s) = 1$; for any formulae α and β, and any state s:

- $s \models \neg\alpha$ iff $s \not\models \alpha$
- $s \models \alpha \wedge \beta$ iff $s \models \alpha$ and $s \models \beta$
- $s \models AX\alpha$ iff for every state s' such that $(s, s') \in R$ we have $s' \models \alpha$
- $s \models EX\alpha$ iff there exists a state s' such that $(s, s') \in R$ and $s' \models \alpha$
- $s \models A(\alpha \mathcal{U} \beta)$ iff for all paths $s'_0(= s), s'_1, s'_2, \ldots$ from s $(\exists i)[i \geq 0$ and $s'_i \models \beta$ and $(\forall j)[0 \leq j < i$ implies $s'_j \models \alpha]$
- $s \models E(\alpha \mathcal{U} \beta)$ iff there exists a path $s'_0(= s), s'_1, s'_2, \ldots$ from s such that $(\exists i)[i \geq 0$ and $s'_i \models \beta$ and $(\forall j)[0 \leq j < i$ implies $s'_j \models \alpha]$

The other propositional connectives, $\vee$ ("or"), $\Rightarrow$ ("implies") and $\Longleftrightarrow$ ("iff") may be defined from $\neg$ and $\wedge$ in the ordinary textbook way. The remaining four of the six modalities mentioned in Section 4.2.1 can now be introduced by definition:

$AF\alpha$ iff $A(true\ \mathcal{U}\ \alpha)$; $EF\alpha$ iff $E(true\ \mathcal{U}\ \alpha)$; $AG\alpha$ iff $\neg EF\neg\alpha$ and $EG\alpha$ iff $\neg AF\neg\alpha$.

By the inductive definition of $\models$ every CTL formula α is or is not true at any state s in a Kripke structure K. Because of the forward looking nature of the temporal operators it is necessary, for unspecified α, to have full knowledge of the distribution of truth values to atomic formulae at all states in the subtree with root s to deduce the truth value of α at s. Conversely, of course, if we do know that α is true at s we know something about the subtree with root s. The convention is adopted of saying that a subtree with root s has property α if the state s itself has property α, which is to say that α is true at s. In particular then, a Kripke structure K has property α iff α is true at the root node s_0.

4.2.3 Fairness and CTL

As mentioned in Section 4.1, the reduction of concurrency to nondeterminism rises the issue of fairness. The specific constraints placed on resolving nondeterminism determine how faithfully concurrency is modelled by interleaving. In [31, Chapter 7] Francez defines various notions of fairness. Of these, *weak* and *strong* fairness are most relevant here. Manna and Pnueli describe weak fairness and strong fairness in the context of transition systems [52, Chapter 2].

Intuitively, weak fairness requires the following: transitions that are *continually* enabled beyond a certain point may not be ignored indefinitely. In addition to the weak fairness requirements, strong fairness requires that transitions that are not continually enabled, but are nonetheless enabled "repeatedly" (infinitely often) may not be ignored indefinitely. Fairness places additional constraints on execution sequences that are acceptable in a verification system. Without these implicit constraints, a verifier will report errors that should be ascribed to "unfair behaviour". For example, a protocol designed to transmit messages over a lossy channel usually includes some mechanism to retransmit lost messages. We assume that the channel will eventually transmit every message successfully, albeit after an unspecified number of retransmissions. However, unless the model specifies some maximum number of retransmissions explicitly, a verification system without fairness constraints will detect the possibility that a given message may never get through and report this as an "error".

Fairness requires that path quantifiers range only over fair paths in CTL. All unfair

execution sequences are ignored. In Chapter 5 it is explained how fairness was implemented in the experimental verification system for ESML.

4.2.4 Correctness requirements

In [52, Chapter 4] Manna and Pnueli illustrate the use of temporal logic to specify correctness properties of reactive programs. They give two different classifications of correctness properties, first in terms of the structure of the formulae and then the well-known *safety-liveness* classification. This is followed by a selection of examples. In practice, however, three generic formulae cover most cases. In [51] Manna and Pnueli described these three classes of properties as "the majority of properties one would ever wish to verify". The logic used is the linear-time temporal logic LTL, but similar classes of properties can be expressed in CTL, as illustrated below:

- **Invariance properties.** These properties are expressed in CTL by formulae of the form AGp, where p is a state property. A common example of an invariance property is a global invariant that constrains the range of a data variable. For example, if the number of outstanding messages (indicated by the variable *out*) may be at most n to ensure correct functioning of a communication system, this can be captured by the following CTL formula: $AG(out \leq n)$. Deadlock freedom is another example of an invariance property, but it is more efficient to let the verifier check directly for deadlock by detecting terminal states. Mutual exclusion is another common example. For two processes P_0 and P_1 that share a buffer, this property can be expressed by the formula $AG(\neg(cr_0 \wedge cr_1))$, where cr_0 and cr_1 indicate which process is inside its critical region. The formula states that both processes cannot be inside their critical regions simultaneously.

- **Response properties.** These properties are expressed in CTL by formulae of the form $AG(p \Rightarrow AFq)$, where p and q are state properties. It is claimed that for all execution sequences a state where p holds is eventually followed by a state where q holds. For example, if a message is sent, it is guaranteed that it will be received eventually.

- **Precedence properties.** These properties involve three assertions p, q, and r, which are again state properties, and are defined as follows in [51]: Any p-state (a state where p is true) initiates a q-interval (an interval all of whose states satisfy q) which either runs to the end of the computation, or is terminated by an

r-state. This property is defined in terms of a weak until operator. In CTL the strong until operator is used and a related class of precedence properties can be defined as $AG(p \Rightarrow A(q\ \mathcal{U}\ r))$. The essential difference between these two until operators is that for the strong until r must eventually become true, while for the weak until operator this is not required.

A verification system is a combination of a transition system and a suitable temporal logic. A reactive system is represented by a transition system from which an execution tree can be computed. Repetitive sequences of states are discarded according to the rules of fairness. (The kind of fairness that was adopted, the precise rules to be followed, and how these rules were implemented is described in Chapter 5.) This reduces the tree to be finite without losing important information. Simple tests on the variables of the transition system are used to determine the truth value of atomic propositions. The reduced tree may thus be regarded as a Kripke structure, and correctness requirements of the reactive system are expressed as CTL formulae. The CTL formula therefore represents a property the system should have while the (finite) execution tree of the reactive system represents a Kripke structure. The verifier can now determine whether the given reactive system has the specified property by checking whether the formula is true at the root node of the tree.

Although transition systems are theoretically powerful enough to model reactive systems, a higher level modelling language is needed in practice. A compiler is needed to translate models expressed in such a modelling language into functionally equivalent transition systems that can be analysed by a verifier.

4.3 The modelling language ESML

Two languages are used to describe reactive programs: ESML (Extended State Machine Language) [21] to specify the behaviour of a program and the branching time temporal logic CTL (Computation Tree Logic) to specify correctness requirements. CTL and how it can be used to express correctness claims was described in the previous section. An overview of ESML is given here.

The design of ESML was inspired by two languages: the systems programming language Joyce [5] and the modelling language Promela [44]. Joyce provided a design framework for ESML. It is a strongly typed programming language based on CSP [41]. The

run-time environment for ESML was strongly influenced by Promela [44], a modelling language for protocols.

After studying Promela and Joyce and also gaining some experience by attempting to construct "paper" models of typical mechanisms found in microkernels, the following observations were made:

- As explained in Chapter 3 (Section 3.3), unique states generated by a model are stored to allow detection of revisited states; the language should therefore allow states to be represented as compactly as possible.
- Dijkstra's guarded command notation [22] which is also used in Promela in slightly modified form, provides a powerful and elegant control formalism.
- Promela supports synchronous as well as asynchronous message passing. When synchronous message passing is used, a process that sends a message is blocked until another process is willing to receive the message. In asynchronous message passing a process may continue immediately after sending a message. This implies some form of buffering to store the message until another process is willing to receive it. Because buffer management presents additional problems, it was decided to implement only unbuffered synchronous message passing in the Gneiss kernel and in ESML.
- Shared variables were considered unnecessary. To exchange information between the various kernel components, CSP-style message passing was considered to be sufficiently powerful.
- It has been mentioned already that models of microkernels require more sophisticated data structures; a modelling language should allow data structures such as queues and tables to be expressed naturally. The challenge is to achieve this without making verification grossly inefficient. To model the Gneiss microkernel, three data structuring mechanisms were considered essential: *records*—to group related data together, *sequences*—as a natural tool to model queues for example, and *arrays*—to represent tables. It is sometimes necessary to combine these structures to form more sophisticated structures. For example, a queue of process records was used to model the scheduler. Each process record contains data about a specific process (process number, process state) while the queue is a sequence of such process records. Efficient verification dictates finite data structures. It

was therefore necessary to restrict sequences to be finite—they are called lists in ESML. Records and arrays are finite by definition.

In ESML a model of a reactive program consists of a collection of processes. ESML processes are similar to agents in Joyce and processes in Promela. A process may contain typed variables as well as other process definitions. Standard operators are provided to manipulate Boolean and integer variables. Processes take the place of modules in a design; after creation they execute concurrently and communicate over channels.

4.3.1 Type and variable definitions

Data variables form part of the global state. To store states compactly, the ranges of variables are fixed by type definitions. Instead of allocating one or more bytes for a number of standard types (say, 16 bits for an integer), a more economical scheme is used: the smallest number of bits is allocated to support subrange types. For example, a subrange of type integer defined as 0..10 can be represented in 4 bits. This idea would probably be too restrictive in a programming language, but it works surprisingly well for a specification language because it leads to a significant reduction in state size.

The keywords "TYPE" and "VAR" indicate type and variable definitions respectively. Extended BNF notation is used to define the ESML grammar. As usual $[\alpha]$ denotes the sentence α or the empty sentence and $\{\alpha\}$ denotes a finite sequence of sentences α or the empty sentence. Terminal symbols are enclosed in double quotes. The syntax of type definitions is defined as follows:

```
TypeDefinition = TypeName "=" NewType ";".
NewType = NewArrayType | NewListType | NewRecordType |
          NewPortType  | NewSubrange | NewEnumeratedType.
NewSubrange = Numeral ".." Numeral.
NewEnumeratedType = Name {"," Name}.
```

Example:

```
TYPE Number = 0..10; Colour = red, amber, green;
VAR counter1, counter2: Number; TrafficLight: Colour;
```

Brinch Hansen found that the most common errors in Joyce programs were type errors in communication commands. He concluded that "any CSP language must include message declarations which permit complete type checking during compilation". We therefore adopted the idea of a *port type* as defined in Joyce. A port type T defines an *alphabet* which is a set $\{s_0(T_0), s_1(T_1), ..., s_n(T_n)\}$ of symbol classes. The values in each symbol class $s_i(T_i)$ are formed by prefixing each value of type T_i with the name s_i. While each value in T_i represents a different message symbol, classes make it possible to group related messages together conveniently. A special symbol class can be defined with no associated messages. This is called a signal. "EndStream" in the example below is an example of a signal.

```
NewPortType = "{" Alphabet "}".
Alphabet = SymbolClass {"," SymbolClass}.
SymbolClass = SymbolName ["(" MessageType ")" ].
MessageType = TypeName.
```

Example:

```
TYPE Number = 0..10; Stream = {value(Number), EndStream};
```

Processes exchange messages via channels by using two communication commands. Channels are accessed indirectly via ports which must be declared in the parameter list of a process. Ports will be discussed in more detail in Section 4.3.4. For now it is enough to know that a message m which belongs to symbol class S of alphabet A is sent via a port p by the command `p!S(m)`. Similarly a message m which belongs to symbol class S is received via a port p and assigned to a variable x of type T by the command `p?S(x)`. When a message is sent, the sending process waits until a matching receive command is executed by another process. Similarly a process wanting to receive a message waits until a message of the specified type is sent. Here is the syntax of communication commands:

```
Communication = VariableAccess "!" SymbolName [ "(" Expression ")" ] |
                VariableAccess "?" SymbolName [ "(" VariableAccess ")" ].
```

4.3.2 Structured data

Although arrays can be used to simulate sequences and records, this was found to be rather awkward. ESML therefore supports arrays, records and lists (sequences of finite length) as separate types. The maximum length of a list is specified by a constant between square brackets as shown in the example below. Any combination of these structures is allowed.

```
NewArrayType = "ARRAY" "[" Constant "]" "OF" TypeName.
NewListType = "LIST" "[" Constant "]" "OF" TypeName.
NewRecordType = "(" FieldList ")".
FieldList = RecordSection { ";" RecordSection }.
RecordSection = FieldName RecordTail.
RecordTail = "," RecordSection | ":" TypeName.
```

Examples:

```
TYPE Row = ARRAY[5] OF Number;
TYPE ProcessRecord = (ProcessNumber, Priority: Number);
TYPE ProcessQueue = LIST[3] OF ProcessRecord;
```

To manipulate lists, standard operations "APPEND", "PREPEND", "HEAD", "REMOVE", "LEN" and "EMPTY" are available. The operations "APPEND(q, x)" and "PREPEND(q, x)" are used to catenate a single new value x to the beginning or the end of a list q. The function "HEAD(q)" returns the first element of a list q without removing it while REMOVE(q) can be used to remove it. "LEN" returns the number of elements in a list. "EMPTY(q)" is used for initialising a list and sets q to the empty list.

4.3.3 ESML commands

The global state of a model is changed by executing commands. Even the SKIP command, which does not change any data variables, changes the state of a model. This is so because the location variable which points to the next command to be executed is modified. The global state can also be modified by evaluating an expression and assigning the resulting value to a type compatible variable. Range checks are always done before assignments. It is therefore unnecessary to use CTL formulae to specify range checks.

Arithmetic expressions based on the four basic operators associated with integer numbers as well as logical expressions with *and* ("&"), *or* ("OR"), *not* ("~") and the usual relational operators are supported. Control structuring is provided by guarded commands, which are defined as follows:

```
IfConstruct = "IF" GuardedCommandList "END".
DoConstruct = "DO" GuardedCommandList "END".
GuardedCommandList = GuardedCommand { "[]" GuardedCommand }.
GuardedCommand = Guard "->" CommandSeq.
Guard = Expression.
CommandSeq = Command {";" Command}.
Command = AssignmentCommand | IfConstruct |
          DoConstruct | PollConstruct | SKIP |
          ProcCommand | Communication.
```

The semantics of the IF and DO constructs conform to the similarly named constructs in Dijkstra's guarded commands [22]. A guarded command can be executed if and only if the expression constituting its guard evaluates to true. A guarded command is executed by executing its corresponding sequence of commands (*CommandSeq*). When more than one guard is true, any executable guarded command is chosen *nondeterministically* and executed.

For an IF construct, at least one of the guards must be true to avoid a run-time error. For a DO construct any executable guarded command is executed; the DO construct is repeatedly executed until all guards are false.

Examples:

```
IF x >= 0 -> z:= x
[] x <= 0 -> z:= -x
END;
DO counter > 0 -> counter:= counter - 1 END
```

Input and output commands are forbidden in the guards of IF and DO constructs. When guarded communication commands are needed, they must be used within a POLL construct, which is defined as follows:

```
PollConstruct = "POLL " GuardedPollList "END".
GuardedPollList = GuardedPollCommand { "[]" GuardedPollCommand }.
GuardedPollCommand = PollGuard "->" InstructionSeq.
PollGuard = Communication [ "&" Expression ].
```

A guard in a POLL construct is true if the communication constituting the guard can execute and the optional Boolean expression evaluates to true. The Boolean expression may contain variables to be received during communication. A POLL terminates when at least one of its guards becomes true.

Examples:

```
POLL request?value(x) & x>0 -> result!value(x + y)
[]   output!value(y) -> SKIP
END
```

Note that in the first guard of the POLL command above, the message can only be received if $x > 0$. The termination rules of DO and POLL constructs are easy to remember. A DO terminates when *all* its guards are false; a POLL terminates when *at least one* of its guards is true.

4.3.4 Concurrency and communication

A new process is activated (created and started) by executing a process activation command. A model must include at least a main process, but usually a number of other processes are declared as well. Execution starts at the main process, which is the only process that may activate additional processes.

During the activation of a process, storage is allocated for all its variables. A process terminates after the last command in its command sequence has been executed. However, the main process may only terminate once all processes created by it have terminated. This termination rule allows simple, efficient run-time storage management for processes. In the current version of ESML, processes cannot be activated recursively. Although this can be implemented (and was used in an early version), it was not needed. Dynamic process creation was therefore eliminated to improve efficiency. The process activation command has the following syntax:

```
ProcCommand = ProcName [ "(" ActualParameterList ")" ]
```

Two kinds of parameters are supported: *value* parameters and *ports*. Ports are marked by one of the keywords "IN" or "OUT" to indicate the direction of transfer. A process activation command must provide an actual parameter (of matching type) for every

formal parameter defined by the process definition. In the case of value parameters the value of the corresponding actual parameter is assigned to each formal parameter. When a process is activated, its ports are mapped onto the corresponding ports of the creator to create *channels*. A channel is a connection between two processes, but it can only be accessed via ports. A process definition consists of a process name, formal parameters and body:

```
Process = "PROCESS" ProcName Block ProcName
Block = [ "(" FormalParameterList ")" ] ProcBody.
FormalParameterList = ParameterDefinition { ";" ParameterDefinition }.
ParameterDefinition = "IN" VariableGroup  |
                      "OUT" VariableGroup |
                      VariableGroup.
ProcBody = [ ConstantDefinitionPart ] [ TypeDefinitionPart ]
           [ VariableDefinitionPart ] { Process ";"}
           "BEGIN" CommandSeq "END".
```

The use of ESML to model various aspects of the Gneiss microkernel is illustrated in Chapter 6. The models are developed and discussed one step at a time and a list of complete models can be found in the appendix.

4.4 Verification algorithm

A compiler is used to translate ESML models automatically into equivalent transition systems. A practical verification strategy is needed to check a model against a given correctness claim as expressed in CTL.

Vergauwen and Lewi published a goal-directed algorithm that can check CTL formulae in time linear in the size of the state space and the number of operators in the formula [72]. The most attractive property of this algorithm is that the reachability graph is computed on-the-fly by evaluating only those subformulae that are actually needed. This means that the algorithm executes the transition system to generate states as necessary to compute the truth-value of a given formula. However, the space requirements limit the practical value of this algorithm. A large data structure is used to store the value of every subformula in all reachable states. The size of the models that can be analysed is therefore limited by the amount of memory available to store this data structure. Furthermore, the algorithm does not address fairness. Nonetheless, this algorithm is a suitable point of departure to discuss the strategy used in the verifier that is described in the next chapter. The version of Vergauwen and Lewi's

algorithm described here is due to Heljanko [39]. It is similar to Vergauwen and Lewi's algorithm, but the formulation is much clearer and more detailed.

A depth-first search is used to explore the reachable state space, keeping track of the values of all subformulae as they are evaluated in each state. In general, the value of a subformula in a given state can be `true`, `false` or `unknown`. An array `info[s, i]` is used to store the value of each subformula `i` in every state `s`. A function `Info(s, i)` returns the value of subformula `i` in state `s` and procedure `SetInfo(s, i, v)` is used to set the value of subformula `i` to the value `v` in state `s`. While computing the value of each subformula, all states visited are marked (by procedure `Mark`) to make it possible to detect loops. Since the algorithm is quite complex, it will be described as a set of interacting procedures, similar to Heljanko's exposition in [39].

The main procedure `Check` is invoked to compute the value of any CTL formula `i` in a given state `s`, typically the initial state of a model. Pseudocode for procedure `Check` is shown in Figure 2. Note that `Check` does nothing if the value of a formula is already known. This is so because `Check` can be called recursively to evaluate some CTL formulae. The values of the simpler subformulae are computed directly, but two procedures `AU` and `EU` are invoked to compute the value of a formula containing these more complex temporal operators.

On completion, `Check` guarantees that the value of formula `i` has been computed in state `s`. The global array `info` must be initialised before invoking `Check`. Each entry in the array stores the values of every reachable state. All these values are initially set to `unknown`. The order in which states are reached is important and therefore a depth-first search number is allocated to every state. Each entry in `info` contains a field to store this number and all are initialised to 0. The function `ValueProposition` returns the value of a propositional variable in a given state. The functions `Left()` and `Right()` return the left and right operands of an operator respectively. When an operator has only one operand, `Left()` is used to return this operand.

Formulae containing the `AU` operator are handled by the two procedures shown in Figure 3 and Figure 4. Procedure `CheckAU` executes a depth-first search. The goal of procedure `AU` is to preserve a counter example if the formula turns out to be false.

Procedure `CheckAU` returns immediately if the value of formula `i` is already known in state `s`. Otherwise, a depth-first search is started to compute the value by calling `Check` to evaluate the right operand of `AU`. Procedure `Check` enters this value in info.

```
PROCEDURE Check(s: State; i: Formula);
BEGIN
  IF Info(s, i) = unknown THEN
    CASE FormulaType(i)
      atomic: SetInfo(s, i, ValueProposition(s, i))
    | not: SetInfo(s, i, ~ValueProposition(s, i))
    | and:
      Check(s, Left(i));
      IF Info(s, Left(i)) = true THEN
        Check(s, Right(i));
        IF Info(s, Right(i)) = true THEN
          SetInfo(s, i, true); RETURN
        END
      END; SetInfo(s, i, false)
    | ax:
      FOR EACH t IN Successors(s) DO
        Check(t, Left(i));
        IF Info(t, Left(i)) = false THEN
          SetInfo(s, i, false); RETURN
        END
      END;
      SetInfo(s, i, true)
    | au: AU(s, i)
    | eu: EU(s, i)
    END
  END
END Check;
```

Figure 2: Procedure to compute the value of a CTL formula in a given state.

```
PROCEDURE AU(s: State; i: Formula);
BEGIN
  Clear(cstack);
  CheckAU(s, i, cstack);
  WHILE ~Empty(cstack) DO
    (* cstack contains a counter example *)
    t := Pop(cstack);
    Unmark(t, i);
    SetInfo(t, i, false)
  END
END AU;
```

Figure 3: Main routine to check AU. It calls `CheckAU` to conduct a depth-first search and displays a counter example if the formula is false.

```
PROCEDURE CheckAU(s: State; i: Formula; cstack: Stack);
BEGIN
  IF Info(s, i) = unknown THEN
    Check(s, Right(i));
    IF Info(s, Right(i)) = true THEN
      Unmark(s, i); SetInfo(s, i, true)
    ELSE
      Check(s, Left(i));
      Push(s, cstack);
      IF Info(s, Left(i)) = true THEN
        Mark(s, i);
        FOR EACH t IN Successors(s) DO
          IF ~Marked(t, i) THEN CheckAU(t, i, cstack)
          ELSE RETURN
          END
        END
        t := Pop(cstack);
        Unmark(s, i);
        SetInfo(s, i, true)
      END
    END
  END
END CheckAU;
```

Figure 4: Routine to handle the AU operator. If a counter example is found, it is stored in cstack. Nothing is done if the value of the formula is known already.

If it is true, the formula `AU` is true no matter what the value of its left operand may be. If the value of the right operand of `AU` is false, the value of its left operand is needed. As before, procedure `Check` is called to compute this value. The state `s` is pushed onto `cstack` because it may form part of a counter example. If the left operand of `AU` turns out to be false, the formula is false, and procedure `CheckAU` returns. Otherwise, the successors of state `s` must be explored. If the formula containing the `AU` operator turns out to be false in any one of the successors of state `s`, the `AU` operator is false in `s`. Otherwise, the state `s`, which has been pushed onto `cstack`, must be removed, unmarked, and the value of the formula containing the `AU` operator can be set to `true` in state `s`. Termination of the procedure is ensured by the marking of states. This can be seen in the FOR-loop that explores the successors of each state t: if a state is found to be already marked, procedure `CheckAU` returns.

The algorithm for `EU` is more complex. Two consecutive depth-first searches are needed. The first search tries to find a goal state: a state in which the right operand of the `EU` operator is true and that can be reached via states in which the left operand of the `EU` operator is true. Whether a goal state could be found is indicated by a global variable for each formula, which is initialised to -1. The efficiency of the algorithm depends on some extra bookkeeping and this accounts for the complexity of the algorithm. While searching for a goal state, all states visited are marked because, in most cases, the value of the formula `EU` can be derived for those states as well.

The first search is implemented by procedures `EU` and `CheckEU` as shown in Figure 5 and Figure 6 respectively.

Procedure `EU` initialises the goal state for formula `i` to -1 (meaning "unknown") and then calls procedure `CheckEU`. The stack `cstack` is used to store all states that are visited while looking for a goal state. Note that the sequence of states on `cstack` does not necessarily form a path. From the semantics of the `EU` operator it is clear that if no goal state is found, the value of the formula can be set to false in all states that were visited during the search. However, if a goal state could be found, a second search is necessary to determine from which states on `cstack` the goal state can be reached. The formula `EU` is then set to true in these states. The second search is implemented by procedures `LabelSCC` (Figure 7) and `EvalSCCs` (Figure 8). Procedure `LabelSCC` clears `fstack` and then calls procedure `EvalSCCs` which implements the second search, pushing states onto `fstack`. Depth-first search numbers are assigned to states to allow computation of strongly connected components using Tarjan's algorithm [1, Chapter 5].

```
PROCEDURE EU(s: State; i: Formula);
BEGIN
  (* Set goal state for i to unknown *)
  SetGoal(i, -1);
  Clear(cstack);
  CheckEU(s, i, cstack);
  IF Goal(i) = -1 THEN (* no witness found *)
    WHILE ~Empty(cstack) DO
      t := Pop(cstack);
      Unmark(t, i);
      SetInfo(t, i, false)
    END
  END
END EU;
```

Figure 5: Main routine to check EU. It calls `CheckEU` to conduct a depth-first search for a path that makes the formula true.

This standard technique is described in detail in Chapter 5. Essentially, the idea is to find the root node of each SCC. The root node is the node with lowest depth-first search number. If the root node of a SCC is found and the goal state is not reachable from it, the formula i is marked `false` in all states of the SCC.

4.5 Summary

This chapter presents background material needed to understand the design of the verifier that was developed to verify the Gneiss microkernel. Correctness claims are verified by exploring every possible execution path of a model. Models are described in a high-level notation that makes the individual processes and their actions clearly visible. Instead of executing this high-level notation directly, a given model is first translated into an equivalent transition system which is more convenient to interpret. The transition system used as computational model is described.

Correctness claims are represented as temporal logic formulae. The branching time logic CTL was selected to express correctness claims because the best model checking algorithms for it require time linear in the size of the state space and the length of the formula. The best model checking algorithms for LTL (another popular temporal logic that can be used to specify correctness claims) is linear in the size of the state

```
PROCEDURE CheckEU(s: State; i: Formula; cstack: Stack);
BEGIN
  IF Info(s, i) = false THEN RETURN
  ELSIF Info(s, i) = true THEN
    (* state s is the goal *)
    MarkState(s, i);
    SetGoal(i, s);
    RETURN
  ELSE (* value still unknown *)
    Check(s, Right(i));
    IF Info(s, Right(i)) = true THEN
      (* goal state found *)
      MarkState(s, i);
      SetGoal(i, s);
      RETURN
    ELSE
      Check(s, Left(i));
      IF Info(s, Left(i)) = false THEN
        Unmark(s, i);
        SetInfo(s, i, false);
        RETURN
      ELSE
        MarkState(s, i);
        Push(s, cstack);
        FOR EACH t IN Successors(s) DO
          IF ~Marked(t, i) THEN CheckEU(t, i, cstack) END
        END
      END
    END
  END
END CheckEU;
```

Figure 6: Routine to handle the EU operator. The algorithm attempts to find a goal state in which it is first discovered that the EU operator is true. The search stops as soon as a goal state is found or when it is discovered that a goal state cannot be reached. A stack is used to store all states visited.

```
PROCEDURE LabelSCC(s: State; i: Formula; goal: State);
BEGIN
  dfsnr := 0;
  Clear(fstack);
  EvalSCCs(s, i, goal, fstack);
  WHILE ~Empty(fstack) DO
    (* cstack contains states visited *)
    t := Pop(fstack);
    SetSearchNr(t, 0)
  END
END LabelSCC;
```

Figure 7: Routine to label states from which goal can potentially be reached. A subsequent search is required to determine which of these states can actually lead to the goal.

space, but exponential in the size of the formula. In practice, however, rather large data structures are needed by the best known verification algorithms for CTL.

An experimental modelling language, ESML, is described that was designed specifically to model the Gneiss microkernel described in Chapter 2 and similar systems. ESML is loosely based on CSP. The language includes the concept of a process, and communication between processes is based on synchronous message passing. In line with the philosophy behind CSP, global variables are prohibited. This did not hinder the project at all because the microkernel verified as a case study was designed as a hierarchy of clients and servers, each represented as a process with its own local data structures.

Finally, an efficient model checking algorithm for CTL is described. A rather abstract version of this algorithm was published by Vergauwen and Lewi [72], the detailed version described here being developed by Heljanko [39]. The main advantage of this algorithm is that it requires time linear in the size of the state space and the length of the CTL formula. Unfortunately, it needs rather large data structures and therefore a modified version of the algorithm was developed to reduce the memory requirements. The new algorithm, which can check a useful subset of CTL is described in the next chapter.

```
PROCEDURE EvalSCCs(s: State; i: Formula; goal: State; fstack: Stack;
  root: INTEGER);
VAR m: INTEGER;
BEGIN
  Unmark(s, i);
  (* assume s leads to the goal state *)
  SetInfo(s, i, true);
  IF s = goal THEN RETURN END;
  INC(dfsnr);
  SetSearchNr(s, dfsnr);
  min := dfsnr;
  Push(s, fstack);
  FOR EACH t IN Successors(s) DO
    m := SearchNr(t);
    IF (m = 0) & marked(t, i) THEN
      EvalSCCs(s, i, goal, fstack, m)
    END;
    IF (m > 0) & (m < min) THEN
    (* state with smaller dfsnr found *)
    m := min
    END
  END;
  IF SearchNr(s) = min THEN
  (* root state of SCC found and goal not reachable *)
    REPEAT
      t := Pop(fstack);
      SetSearchNr(t, 0);
      SetInfo(t, i, false)
    UNTIL t = s
  END
END EvalSCCs;
```

Figure 8: Routine to determine from which states in fstack the goal state can be reached. In these states the EU formula is true and in all other states on fstack it is false.

Chapter 5

A verifier for reactive systems

As explained in Chapter 4, the efficiency of fast model checking algorithms for CTL depends on the storage of intermediate results in main memory. Vergauwen and Lewi's algorithm uses a large table to store the value of every subformula in all reachable states. They suggest that memory can be saved by storing only the results of subformulae that are expensive to evaluate, but point out that this would destroy the linear time property of the algorithm. The algorithm of Bhat, Cleaveland and Grumberg for CTL* has similar space requirements [4]. It constructs a strongly connected component of the state graph on-the-fly and, in addition, two large data structures are used to store all assertions that have been encountered, and a set of assertions that have been found to be false.

From a practical point of view, there is another problem with both these algorithms: no mechanism is provided to cope with state spaces that are too large for a given memory size. Even one state too many would cause a verification system based on these algorithms to terminate with an "out of memory" message. For a practical verifier this is unacceptable. The goal should be to design a system that can produce useful results even for models that generate state spaces too large to fit into memory.

The challenge, therefore, was to design a verification algorithm whose performance degrades gradually under overload conditions. A cache with replacement of states is used to meet this requirement at the cost of an (often acceptable) increase in run time. The system described here accepts models written in ESML and correctness claims expressed in a subset of CTL. Although the basic strategy is similar to that of Vergauwen and Lewi's algorithm, there are a number of important differences:

- A state cache with replacement has been incorporated to increase the size of models that can be analysed.
- A compaction scheme is used to increase the number of states that can be stored in a cache of a given size.
- The system is designed to handle a useful *subset* of CTL rather than full CTL. This makes it possible to save memory by postponing the evaluation of nested temporal operators without storing all intermediate results.
- Fairness has been incorporated.

5.1 A cache to detect revisited states

Revisited states can be detected by storing all unique states as they are generated. However, models of realistic size generate far too many states for this to be practical. It was decided to address this problem by using a *state space cache*—a technique that was first proposed by Holzmann in [42]. The basic idea is to store as many states as possible and to replace (overwrite) states as memory fills up. A good description of this technique can be found in [28]. This strategy reduces memory requirements at the expense of increased runtime, because the removal of states from the cache will probably cause some work to be repeated.

Experiments show that state space caching makes it possible to handle state spaces that are up to three times larger than can be stored in memory. If necessary, the performance of a state cache can be improved further by reducing the probability of reaching each stored state. This can be done by combining a cache with partial order techniques as was argued in [36]. In addition, a good state compaction scheme makes it possible to fit even more states into memory. However, most compaction schemes require additional processing which lead to a substantial increase in runtime.

5.2 An efficient state compaction scheme

A simple type in ESML defines a finite set of values. Variables of these types can be encoded in $\lceil log_2 n \rceil$ bits, where n is the cardinality of the set defined by the type. For example, type BOOLEAN can be encoded in a single bit, while the subrange type

0..10 will need 4 bits. Complex data types cause a substantial increase in the state size. Since the number of states that can fit into a cache of a given size depend on the state size, a compaction technique is used to represent data types in the smallest possible number of bits. Consider a state containing the following variables:

```
VAR c: 0..4;
    b: BOOLEAN;
    p: (x, y: 0..149);
```

The compacted state is computed as $c+5\cdot(b+2\cdot(p.x+150\cdot p.y))$, which can be written as $1\cdot c+1\cdot 5\cdot b+1\cdot 5\cdot 2\cdot p.x+1\cdot 5\cdot 2\cdot 150\cdot p.y$ to show that each variable z has two associated "mask" factors z_{lo} and z_{hi} that are computed by the ESML compiler. In particular, $c_{lo}=1$, $c_{hi}\doteq 1\cdot 5$, $b_{lo}=1\cdot 5$, $b_{hi}=1\cdot 5\cdot 2$, $p.x_{lo}=1\cdot 5\cdot 2$, $p.x_{hi}=1\cdot 5\cdot 2\cdot 150$, $p.y_{lo}=1\cdot 5\cdot 2\cdot 150$, and $p.y_{hi}=1\cdot 5\cdot 2\cdot 150\cdot 150$. This is analogous to the way a decimal number such as 3425 can be computed as $5+10\cdot(2+10\cdot(4+10\cdot 3))$. The only difference is that in the compaction scheme a variable size "radix" is used which depends on the number of possible values assumed by each variable.

Extracting the value of a given variable z from a compacted state S is simple and requires only two operations: $z=(S\ mod\ z_{hi})\ div\ z_{lo}$. When the value of a variable z changes to z', the updated compacted state $S'=S+z_{lo}\cdot(z'-z)$. Because the factors z_{lo} and z_{hi} are computed at compile time this compaction scheme is simple to implement and efficient. The *mod* and *div* operations that are required at run time represent the only run-time overhead.

We have modified the verifier SPIN to incorporate the same compaction scheme, obtaining similar results [34]. A number of well-known Promela models were used to measure the effectiveness of the compaction technique. Some of the models (leader, pftp, snoopy, and sort) are part of the standard SPIN distribution[1]., while the other models (cambridge and slide) were collected from the Internet

Table 1 shows the memory required (in Mbytes) followed by the time needed to complete each validation run (in seconds)[2]. The first column (marked SPIN) gives the results for the standard SPIN system. The next column (marked SPIN-C) is for the modified SPIN system with compact representation of states. The last column shows the ratio between the two systems for purposes of comparison. Note that the technique works

[1] `http://cm.bell-labs.com/cm/cs/what/spin/Man/Exercises.html`

[2] Tests were executed on a 400MHz Pentium II with 256 Mbytes of memory.

Model	SPIN		SPIN-C		Ratio	
	memory	time	memory	time	memory	time
cambridge	54.1	6.9	23.8	6.8	0.44	0.99
leader	91.7	12.6	33.2	7.7	0.36	0.61
pftp	66.6	8.9	26.5	7.3	0.40	0.83
snoopy	15.2	1.7	10.6	1.4	0.70	0.84
sort	3.7	0.3	4.4	0.2	1.19	0.77
sort(BIG)	130.7	22.6	70.4	14.7	0.54	0.65
slide	33.3	3.2	20.3	3.2	0.61	0.97
Average					0.60	0.81

Table 1: Comparison of standard SPIN against SPIN with compaction (SPIN-C)

best for the larger models. A useful reduction in memory requirements was measured for most models, the average being 40%. More surprising, however, is that execution time was *decreased* on average by 19%. The reason appears to be that the hash function associated with the state cache takes longer to compute for larger states. This seems to be enough to offset the small run-time overhead of compression. As can be expected, the results differ from one model to the next. This is due to differences in the number of variables, the number of read and write accesses, and the structure of the state space.

5.3 Delayed evaluation of subformulae

Formulae with nested temporal operators such as $AG(p \Rightarrow AFq)$ present special problems. Such formulae can be handled in different ways. The original global model checking algorithm given in [15] works bottom-up, starting with the most deeply nested subformulae. This algorithm is straightforward, but unnecessary values are often computed. A more serious problem is that the entire reachability graph must be stored in memory. Therefore the practical value of this approach is limited.

Vergauwen and Lewi's algorithm needs a large array to store the value of every subformula in every state. A subset of CTL can be checked efficiently without storing any intermediate results. Fortunately the kind of CTL formulae that can be checked in this way are those that are often needed in practice. To verify the Gneiss microkernel only two kinds of correctness claims were needed: *invariance* properties and *response* properties. In fact, according to Manna and Pnueli, a slightly larger subset of CTL is adequate to specify most correctness claims needed in practice as described in

Chapter 4. The verification system described here can check invariance and response properties. The verification process proceeds top-down, but the *evaluation of nested temporal subformulae is postponed for as long as possible.* This idea was proposed in [19].

Only the subset of CTL that was needed to verify the Gneiss kernel was actually implemented. This subset will now be described precisely. Formulae of the general form $AG(\alpha)$ were used to describe invariant properties. The argument α of such formulae is required to be a state property. That means, it may not be anything else than an expression composed of the propositional operators $\wedge$, $\vee$, $\neg$, and $\Rightarrow$, with parenthesis used to arbitrary levels for grouping terms together, according to the syntactical definition of full CTL. Formulae of the general form $AG(\alpha \Rightarrow AF\beta)$ were used to express response properties. The arguments α and β of such formulae are restricted as as above: they may not be anything else than state properties.

Consider a CTL formula of the form $AG(\alpha \Rightarrow AF\beta)$. To determine the truth value of this formula in state s the verifier will explore all paths leading from s while checking that in each state along every path the argument to AG is true. Due to the implication, it is unnecessary to determine the truth value of the nested temporal formula $(AF\beta)$ in any state where α is false. If α is true however, the truth value of $AF\beta$ is needed. Instead of computing this truth value immediately, the state s and the subformula are recorded as a *subproblem* to be analysed at a later stage. In the mean time, the nested subformula $AF\beta$ is *assumed* to be true and the search continues. This approach has two advantages: (1) it reduces the memory requirements and (2) the model checking algorithm for the nested formulae that were needed is quite simple.

When nested subformulae are evaluated immediately, the results must be stored in randomly accessible memory so that the verifier can refer to these results. The efficiency of Vergauwen and Lewi's algorithm is based on the availability of these intermediate results. By postponing evaluation, the implicit assumption is that all nested subformulae will eventually work out to be true. This makes it unnecessary to refer to any intermediate results. It is often necessary to evaluate the same nested subformula in many different states. A list of such states is stored and when it becomes too long, it can be written to disk. The technique of postponed evaluation has been implemented and used to check correctness claims for the Gneiss microkernel.

Invariance properties, as expressed in the subset of CTL that was implemented, are simple to check. The entire reachable state space must be explored because of the AG

operator. The condition to be checked in each state is a state property and therefore simple to check. Consequently, the size of the state space determines the cost of checking invariance properties.

The technique of postponing the evaluation of subformulae was only needed to check response properties. First, the entire state space is explored to discover a subset of states in which additional checking must be done. This is the set of states in which some triggering event occurs to which a response is needed. The subsequent checking ensures that each triggering event is always followed by the desired response.

5.4 Strongly connected components and fairness

As mentioned in [15], fairness can be implemented by computing the strongly connected components (SCCs) of the reachability graph. All SCCs of a graph can be computed in linear time[3] by an algorithm due to Tarjan[1, Chapter 5]. The algorithm is shown in Figure 9. Each node or vertex represents a state in a state graph. Nodes are numbered during a depth-first search to preserve the order in which they are visited. These numbers are used to detect the "root" node of each SCC—the node with the lowest search number that belongs to the SCC. Nodes that belong to a SCC are pushed onto a separate stack (the SCC stack). To find the root node of each SCC, each node has a field called *lowlink*. The *lowlink* field of each node is initially simply the depth-first search number of the node. This field must always contain the number of the node with the lowest search number that can reach the current node and the *lowlink* field of new nodes is updated to ensure this. If a node is reached which has a lower *lowlink* value than the current node, this lower value is used to replace the *lowlink* value of the current node. In [1, Chapter 5] it is proved that when the *lowlink* field of a node equals its search number, this node must be the root of the topmost SCC contained in the SCC stack.

Intuitively, the version of fairness that was implemented requires that on a fair path a transition cannot be enabled infinitely often without eventually being executed. This version of fairness for CTL was formally defined by Tuominen in [69]. A path (an infinite sequence of successive states) is fair if for each suffix of the path the following holds: if some state s_i appears infinitely often in the suffix and another state s_j is directly reachable from s_i then s_j appears at least once in the suffix.

[3]Time O(Max(n, e)), where n = number of nodes, e = number of edges

```
PROCEDURE Tarjan(v: Vertex);
VAR w, x: Vertex;
BEGIN
  INC(count); (* count nodes as they are visited *)
  v.dfnr := count;
  v.lowlink := v.dfnr;
  v.marked := TRUE;
  push v on stack;
  FOR each descendant w of v DO
    IF NOT w.marked THEN
      Tarjan(w);
      v.lowlink := Min(v.lowlink, w.lowlink)
    ELSIF (w.dfnr < v.dfnr) & w is on the stack THEN
      v.lowlink := Min(v.lowlink, w.dfnr)
    END
  END;
  IF v.lowlink = v.dfnr THEN (* root found *)
    REPEAT pop x from stack UNTIL x = v
  END
END Tarjan;
```

Figure 9: Tarjan's algorithm to compute the strongly connected components of a graph.

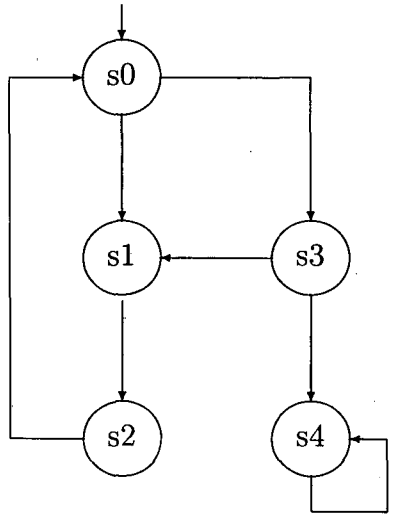


Figure 10: Reachability graph with unfair paths through a strongly connected component

Consider the reachability graph shown in Figure 10. The states s_0, s_1, s_2, s_3 form an SCC rooted at s_0. The root of an SCC is the state that is encountered first during a depth-first search. More than one transition is enabled in states s_0 and s_3. For example, in s_3, one transition leads to state s_1 and the other to s_4. Along some path, state s_3 may occur infinitely often since it is inside the SCC rooted at s_0. Any path from s_0 that continually ignores the transition leading to state s_4 is unfair.

To implement the version of fairness considered here, the path operators of the subset of CTL that was defined earlier must range over fair paths only. Unfair paths must be ignored subject to certain conditions to be discussed shortly. For the specified subset of CTL only the AF operator requires special treatment to implement fairness. The operator AG requires that all paths must be checked and needs no special treatment.

For formulae that contain the AF operator special care is needed when some path leads to an SCC where the formula is false in all states and some state in the SCC has more than one transition enabled.

As an example, consider the state graph shown in Figure 10 again. Suppose the truth value of $AF\alpha$ must be determined in state s_0 and assume that α is false in all states except state s_4. Therefore, for the formula to be true in s_0, each path from state s_0 must lead to a state in which α holds. If a path starting at s_0 contains no state in which α is true, and leads back to s_0, it is necessary to check whether this path is fair or not. For this it is necessary to know whether some other transition from s_0 leads to a state in which α is true. If this is not so, the formula is false in s_0. On the other hand, if a state in which α holds can be reached from s_0 the cycle may be ignored. Therefore, to decide whether $AF\alpha$ is true in s_0, we must know whether a state in which α is true can be reached from s_0. As can be seen in Figure 10 such a path exists. However, depending on the order in which states are generated, this information may not be available when the first cycle (s_0, s_1, s_2, s_0,...) is detected. Therefore, if a cycle leads back to state s_0 for which α is false in all states, it cannot be assumed immediately that the formula $AF\alpha$ is false in s_0. Instead, processing continues after setting a flag in state s_0 to indicate that more information is needed: *before backtracking from s_0, some path from s_0 must lead to a state where α is true.* In Section 5.9 the details of implementing this version of fairness is discussed.

5.5 Structure of the verifier

Two versions of the ESML verifier were implemented. The first version generated a Modula-2 program from each model [73]. This program was then compiled to produce an executable verifier for a specific model. State generation, state storage and state analysis were intertwined in this implementation, which made the system difficult to understand. Clearly, a verifier of this kind can be divided into four relatively independent components:

- **A compiler.** This is a separate program that translates a model into a functionally equivalent transition system.
- **The state generator.** This module is responsible for executing a transition system to explore a model's state space. It also contains the mechanisms needed to support process scheduling and fairness.
- **A state storage module.** This module is needed to detect revisited states by storing as many states as possible. One can experiment with different state storage techniques by replacing this module.
- **The analyser.** The state generation process is controlled by this module. It calls upon the state generator to generate states as needed to verify a given formula. By replacing the analyser module, it should be possible to handle different temporal logics.

A second version of the ESML verifier was developed to reflect this modular structure. The completed system is easier to understand and simpler to modify. One of the advantages of the design is that each module can be described and understood on its own. The ESML compiler is based on standard compilation techniques, the details of which are irrelevant here. For a given ESML model it generates functionally equivalent low level code that represents a transition system. The low level instructions were designed to simplify state generation.

The verification system is divided into three components, each with a well-defined function. (1) Module *StateGenerator* is an interpreter for the abstract machine that handles state generation. It interprets the low level code generated by the compiler to explore the state space of the model. (2) Module *Storage* implements a cache to store as many states as possible to detect revisited states. It is used by module *StateGenerator*.

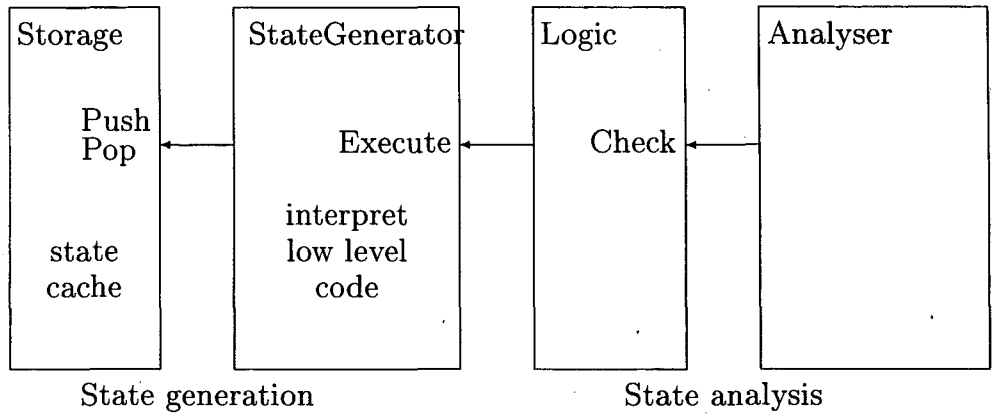


Figure 11: Structure of the verifier

(3) The rest of the verifier consists of two modules: *Logic* and *Analyser*. Module *Logic* contains a procedure *Check* to verify subformulae that contain at most one temporal operator. Procedure *Check* relies on the state generator to generate new states when necessary. Module *Analyser* handles formulae that contain nested subformulae. It contains a procedure *CheckFormula* that calls procedure *Check* to determine the values of subformulae, one at a time. The interaction of the modules that make up the verifier is shown in Figure 11. State generation is completely separated from state analysis. The code generated by the compiler is read in by module *StateGenerator*. This code contains all information about a given model, including the CTL formula to be checked. Specific instructions are executed to check the propositions in the given CTL formula in every state generated. The values of these propositions are transmitted to module *Logic* on request.

5.6 The state generator

The state generator is a separate module. Its implementation contains much detail but it is based on standard techniques. It is basically an interpreter for a stack-based abstract machine designed with state generation in mind. The instructions are classified as (1) typical instructions for stack-based evaluation of expressions such as `PushVariable`, `PushValue`, `Add`, `Greater`, and `Equal`, (2) instructions to handle flow of control: `guards` and `jump`, (3) instructions to manipulate operations on lists, for example `Head`, and `Length`, (4) instructions for communication between processes: `Hook`, `Bang`, and `Poll`, and (5) instructions to create and terminate processes. The memory

Figure 12: Memory organisation of the abstract machine

of the abstract machine is implemented as an array `store` of integers. It contains the abstract code to be interpreted, the variables of each process (the location pointer and data variables of each process are collectively known as a frame), and an expression stack which is used to evaluate expressions. The frame for each process is created by executing the `Activate` instruction. The memory organisation is shown in Figure 12.

Module *StateGenerator* exports the following procedures:

- **Execute.** This procedure executes a new transition if possible to generate a new state. The current state is stored within module *StateGenerator*. Terminal states indicate a deadlock.
- **Evaluate.** This procedure is called to return the value of a given proposition in the current state.
- **Backtrack** The state generator is instructed explicitly to fall back to the current state's immediate predecessor.
- **GetFormula.** This procedure returns the operator (op) and operands (left, right) of the subformula (sf) that is being analysed. Procedure *Getformula* is called by module *Logic*.
- **SetGoodChildren.** This procedure is needed to support fairness. It is a very simple procedure that sets a flag to indicate that a state is reachable from the current state where the argument of the current formula holds. This value is copied to the predecessor of the current state when the verifier falls back.
- **SetState.** This procedure is used to initialise the state generator.

Procedure *Execute* (Figure 13) implements a depth-first search based on a cache with replacement of states. Each transition corresponds to executing some process, the procedure *Reschedule* selecting a new process when no transition is enabled in the current process. One or more transitions may be enabled in every state. The process number determines the order in which transitions are selected for execution. In every state generated, a transition is selected starting from process 0. If no transition of

process 0 is enabled, a transition of the next process (process 1) is selected, and so on. If multiple transitions are enabled in the same process (for example when more than one guard of an IF command is true) the first one will be selected and later, when the verifier falls back to the state where the first transition was executed, the next one will be executed, and so on. This strategy ensures that a depth-first search is performed. All states encountered along the current execution path are stored in a stack. The process number is used to determine whether there are any enabled transitions left in the current state. If no more transitions are enabled in the current state, the function *Backtrack* is called. Procedure *Backtrack* pops the topmost state from the stack and returns either *Backtracked* (thereby falling back to the current state's predecessor) or *Complete* (when the current state was the initial state).

After a state has been generated, procedure *Push* is called, which tries to enter it in the stack. There are three possibilities: (1) if it is a new state, it is entered in the stack and the constant *Inserted* is returned, (2) if the state was already on the stack, the constant *Loop* is returned, and (3) if the state is neither new nor on the stack, it means that it was already encountered along some earlier path; the constant *Revisited* is returned to indicate this. A simple CASE command is used in procedure *Execute* to distinguish between the three possibilities.

It is also possible that no state was generated. This can happen when an attempt to execute a communication command causes the current process to block. It is then necessary to select another process to execute. An error (such as division by 0) can also occur.

Procedure **Execute** handles rescheduling of processes and calls procedure **Step** which attempts to execute a transition from the current process. Procedure **Step** is a typical interpreter for a stack-based computer. Its general structure is shown in Figure 14.

Executing a transition may involve executing more than one instruction. Procedure `Step` sets the variable `transition` to `TRUE` as soon as a transition has been completed. The opcode of the current instruction (stored in the abstract machine's memory at position `loc`) is used to select the appropriate code to interpret each instruction. Some instructions have parameters which follow the opcode. The first of these is referenced as `store[loc+1]`, the second as `store[loc+2]`, etc. During interpretation of each instruction the location pointer `loc` is updated to point to the next instruction. The code to interpret some instructions is straightforward. For example, the following code interprets the `Add` instruction:

```
PROCEDURE Execute*(): INTEGER;
VAR loc, pos, base: INTEGER;
BEGIN
  LOOP
    Reschedule;
    IF AllProcessesTried THEN RETURN Backtrack() END;
    Step(loc, currentproc);
    IF result = Progress THEN (* successful execution *)
      CASE Stack.Push(s) OF
        Inserted: RETURN Forward
      | Revisited: RETURN Revisit
      | Loop: RETURN Cycle
      END
    ELSIF result = NoProgress THEN (* do nothing *)
    ELSIF result = Error THEN RETURN Error
    END
  END
END Execute;
```

Figure 13: Procedure to execute a single transition

```
PROCEDURE Step(VAR loc: INTEGER; process: INTEGER);
(* declarations *)
BEGIN
  transition := FALSE;
  REPEAT
    CASE store[loc] OF
      PushValue: (* code to interpret the PushValue instruction *)
    | Add: (* code to interpret the Add instruction *)
    | ...
    END
  UNTIL transition (* a transition has been completed *)
END Step;
```

Figure 14: General structure of the interpreter for low-level code. The location pointer `loc` points at the current instruction. The code is stored in an array `store`.

```
INC(sp);
store[sp] := store[sp] + store[sp-1];
INC(loc)
```

Before executing an `Add` instruction, the values of its two operands must be pushed onto the stack by executing `PushValue` or `PushVariable` instructions. Executing the `Add` instruction will replace the topmost two values on the stack by their sum. Note that the stack is popped by *incrementing* the stack pointer.

Instructions such as `Add` are used to evaluate expressions. The final result of an expression is stored by updating some data variable in the frame of the current process. This will change the global state of the model and variable `transition` is set to `TRUE` to indicate that a transition has been completed. Note that although the location counter is modified during execution of instructions such as `Add`, this is not reflected as a global state change until a data variable is updated. Each assignment therefore constitutes a single transition.

The unconditional jump instruction has one parameter to specify the location to jump to. This is stored in the next memory location following the opcode. The code to interpret it is equally simple:

```
loc := store[loc+1];
result := Progress;
transition := TRUE
```

Although the `Jump` instruction changes only the location pointer of the current process, it is seen as completing a transition. Therefore a global variable `transition` is set to `TRUE`. This variable is needed because some transitions may involve the execution of several instructions. The procedure *Step* does not return before `transition` is `TRUE`, as can be seen in Figure 14. The global variable `result` is used to indicate whether a completed transition made progress or not.

Guarded commands are used as control structures in ESML. The instructions `jump` and `guards` are used to handle any number of guards. Each guard is represented by a pair of numbers `(m, n)`, where `m` is the address of the code that evaluates the guard expression and `n` is the address of the next guard. A guarded command (IF or DO) is

```
IF x = 0 -> y := 0
[] x # 0 -> y := 1
END

12 guards 32
14 (47, 23)
16 pushValue 2
18 evaluate 44
20 popVariable
21 jump 33
23 (54, 32)
25 pushValue 2
27 evaluate 61
29 popVariable
30 jump 33
32 trap
33 ...
```

Figure 15: Translation of ESML guarded commands

translated into a `guards` instruction followed by any number of guards. The `guards` instruction has a single parameter which is the address of the instruction following the last guard. The guards are evaluated sequentially and the first true one is selected. The corresponding instruction sequence following the selected guard is then executed. The low-level code generated for an IF command is shown in Figure 15. At address 12 is a guards instruction that controls the entire guarded command. Its parameter shows that the address of the instruction following the last guard is 32. At that address is a `trap` instruction, which will be executed to abort execution if all guards of the IF are false, as required by the ESML semantics. The first guard is at address 14. It indicates that the code to evaluate the guard expression is at address 47 and the next guard is at address 23. The code following each guard is executed if the guard is true. Otherwise the next guard is evaluated. The `pushValue` instruction pushes the offset of a variable onto the stack and the `evaluate` instruction executes code indicated by its parameter to leave a result on the stack. This result is assigned to the variable by the `popVariable` instruction. (The address of the variable is on the stack right below the result.) The DO command is handled in a similar way. The only difference is that it ends with a `jump` instruction to transfer control back to the beginning of the loop. The DO command terminates when all its guards are false.

To implement simple communication commands, two instructions **hook** (to receive messages) and **bang** (to send messages) are used. No attempt is made to find a communication partner when a **hook** instruction is executed; the process is simply blocked and another process is scheduled. When a process executes a **bang** instruction, the other processes are probed systematically to find one that is blocked at a **hook** instruction, willing to transmit an acceptable message. If such a communication partner is found, the message is transferred, otherwise the process that executed the **bang** instruction is blocked. It can be seen that the implementation of communication is asymmetric. In the SPIN verifier a simpler, more efficient strategy is used. When a process executes a send or receive operation in SPIN, this process is entered in a queue if no suitable communication partner is available yet. When another process later executes a matching communication command, it is simple to test whether a message is waiting or not. However, the fact that two POLL commands in ESML must be able to synchronise required the more elaborate mechanism described above. In retrospect, it was perhaps a mistake to incorporate such an elaborate mechanism in ESML, since it is seldom used. This mechanism accounts for the extra states generated by the ESML verifier for communication commands when compared to SPIN, as described in Section 5.10.1.

The **hook** instruction has parameters to indicate (1) the channel number, (2) which message is expected, (3) where the data must be stored, (4) the size of the data, and (5) a condition that must hold for the data to be acceptable. The last parameter is set to a predefined constant (MAX(INTEGER)) when no condition was specified. The **bang** instruction has similar parameters, but because data must be transmitted and not received, the third parameter indicates an expression to be evaluated to push the data onto the stack.

The POLL command of ESML is translated into a **poll** instruction followed by a series of **hook** and/or **bang** instructions. The structure of the code generated for POLL commands is similar to that generated for IF commands. However, the **poll** instruction takes the place of the **guards** instruction and **hook** and **bang** instructions take the place of guards. A simple example is shown in Figure 16. The **poll** instruction (line 7) has a single parameter to indicate the address of the first instruction following the entire POLL structure. The first guard (`c1?m`) is translated into a **hook** instruction. Its first two parameters indicate that a message must be received from channel 0, and that message number 0 (`m`) is expected. The next parameter is 0 because no data is associated with the message. The next parameter (data size) is 0 because there are no associated data, and the number 32767 (MAX(INTEGER) indicates that there is

```
POLL c1?m -> SKIP
[] c2!m -> SKIP
END

  7 poll 27
  9 hook 0 0 0 0 32767 (19)
 16 skip
 17 jump 27
 19 bang 1 0 0 0 32767 (27)
 26 skip
 27 ...
```

Figure 16: Instructions generated for a simple POLL command.

no associated Boolean expression either. The number (19) indicates where the next guard is located. For this guard a `bang` instruction was generated, which has similar parameters.

An `activate` instruction is used to activate a new process. It has the following parameters: (1) the identification number of the process, (2) the address of the first instruction of the process, (3) the number of variables declared for this process, (4) the number of parameters, and (5) the number of channels created by this process. Executing an `activate` instruction creates a frame, which reserves space for the local data of the process (including its instruction pointer).

The state generator is initialised by transferring the contents of a specified file that contains compiled code to the lower part of `store`. Variable `framesstart` is initialised to point at a position directly after the code because this is where the first frame will be stored. At this stage variable `framesend` is set to the same position. The first process is then created. This involves executing the first `activate` instruction.

5.7 State storage

The cache is implemented as a separate module *Storage*. A state space caching scheme is used, with a suitable hash function to compute the position of each state in a large array. Closed hashing is used and if a hash collision occurs (a slot is occupied), rehashing with a second hash function is used to search for an open slot. The second hash function provides an "offset" which is added to the current position in the hash table and this

mechanism is used repeatedly, trying to find an open slot. If no open slot can be found during a predefined (small) number of probes, the state currently pointed at is replaced.

To keep track of the current execution path, states are pushed onto a stack as they are generated. Fully analysed states are moved to the cache when they are known to satisfy the current subformula being checked. Instead of storing states that are on the current path (the stack) and states that are in the cache in separate data structures, a single large hash table is used to store all states, whether they are in the cache or on the stack. States that are on the stack are marked. When a state must be moved from the stack to the cache, it is only necessary to change its marking. The stack, which is a separate array, only contains pointers to the state values (as stored in the hash table). The advantage of this scheme is that the same hash computation can be used to look up a state whether it is in the cache or on the stack. A cache that is large enough to store at least 40% of a model's states was found to be needed for acceptable performance. A search path can be terminated when a state is reached that is in the cache. Such a mechanism is really essential to keep verification run times within acceptable limits.

5.8 State analysis

The module *Logic* (which is part of the analyser component) exports a procedure *Check* which is used to evaluate formulae. Formulae evaluated by *Check* may not contain more than one temporal operator. More complex formulae are handled by calling *Check* repeatedly. Here is the code for procedure *Check*:

```
PROCEDURE Check*(sf: INTEGER; VAR result: BOOLEAN);
VAR
  op, left, right, res: INTEGER;
BEGIN
  StateGenerator.GetFormula(sf, op, left, right);
  (* determine truth value in initial state *)
  value := TruthValueFormula(sf);
  IF value = U THEN (* result undetermined *)
    LOOP
      (* generate a new state *)
      res := StateGenerator.Execute();
      (* now analyse the state *)
      CASE res OF
         Forward:
           value := TruthValueFormula(sf);
```

```
            IF value = T THEN
              Trace.SetGoodChildren;
              res := StateGenerator.Backtrack(); AdaptValue(op)
            ELSIF value = F THEN EXIT
            END
        |  Complete:
             value := FinalValue(op); StateGenerator.BackTrack; EXIT
        |  AllChildrenExplored:
             value := FinalValue(op); StateGenerator.BackTrack;
            IF value = F THEN EXIT END
        |  Revisit:
            Path.SetGoodChildren;
            AdaptValue(op)
        END
      END
    END;
    IF value = T THEN result := TRUE
    ELSE result := FALSE
    END
  END Check;
```

The procedure *Setstate*, exported by the state generator module, is used to set up the state generator in the correct initial state before calling *Check*. The initial state is passed to *SetState* in packed (compacted) form. The subformula to be checked is indicated by parameter sf. The procedure *Getformula* returns the current operator since that is needed to determine whether to proceed or not. For example, when checking the formula $AG\alpha$, the verifier can stop as soon as a state is reached where α is false. On the other hand, when checking $AF\alpha$, this is not so and execution should continue. Module *Logic* contains a procedure *TruthValueFormula* which computes the truth value of subformula sf in the current state. The evaluation of propositions will be discussed first because that is quite simple. To determine the truth values of propositions, a procedure *Evaluate*, which is exported by the state generator, is called by *TruthValueFormula*. The state generator has access to the current state and therefore the values of all data variables. It is often impossible to determine the truth value of a subformula that contains a temporal operator by examining a single state. In such a case the value U (undetermined) is returned. Further states are generated until the value T (*true*) or F (*false*) is obtained. The code shown below illustrates how the subset of CTL that was used to verify the Gneiss kernel is handled:

```
PROCEDURE TruthValueFormula(sf: INTEGER): TruthValue;
VAR
  op, left, right: INTEGER; v: TruthValue;
BEGIN
  StateGenerator.GetFormula(sf, op, left, right);
  CASE op OF
  (* Modal operators: *)
      AF:
        v := TruthValueFormula(left);
        IF v = F THEN v := U
        ELSIF v = U THEN (* error: invalid formula *)
        END
  | AG:
      v := TruthValueFormula(left);
      IF v = T THEN v := U
      ELSIF v = U THEN (* error: invalid formula *)
      END

  (* Propositional operators: *)
  | Imp:
      v := TruthValueFormula(left);
      IF v = F THEN v := T
      ELSIF v = T THEN
        v := TruthValueFormula(right);
        IF v = U THEN SubProblem(right); v := T END
      ELSIF v = U THEN (* error: invalid formula *)
      END
  (* other propositional operators, handled similarly *)
  ELSE (* Proposition *)
    v := StateGenerator.Evaluate(op)
  END; RETURN v
END TruthValueFormula;
```

When procedure *Execute* (exported by the state generator) is called to generate a new state, the result will be one of the following: *Forward* (a new state was generated), *Complete* (all states have been explored), *AllChildrenExplored* (all paths leading from the current state have been explored), or *Revisit* (a state was revisited). The appropriate action in each of these cases will be described separately.

- **New state generated.** In this case (res = *Forward*), *TruthValueFormula* is

called to assign the value of the subformula sf in this new state to *value*. Procedure *TruthValueFormula* only returns F if the correctness claim has been violated, meaning that the verification run can be stopped, because the correctness property that is being checked has been violated. However, while the value is U (or T for some formulae), verification must proceed. As an example, consider checking the CTL formula $AF\alpha$ in a state s. A state may be found along the first execution path leading from s where α holds. However, this does not mean that a state where α is true is reachable along all paths that lead from s. As soon as a state is found in which α holds, exploration of that path is stopped and the verifier falls back to the state's predecessor. From this new state—the predecessor of the state in which α holds—there may exist another path to be explored.

It is therefore necessary to adapt the truth value stored in *value* before verification proceeds, depending on the specific temporal operator. The procedure *AdaptValue* is used to accomplish this. Its actions depend on the particular temporal operator. For the formulae considered here (AG and AF), procedure *AdaptValue* sets *value* to *true*.

When all paths leading from the current state have been explored, procedure *FinalValue* is called. This will determine the value of the formula being checked in the current state before the verifier falls back. For AG *FinalValue* returns *true*, and for AF it returns *true* if *GoodChildren* is *true*, otherwise it returns *false*. If *GoodChildren* is *true* in the current state, it means that a state is reachable from it in which the operand of AF is *true*. This is similar to the idea of a *goal* state in Vergauwen and Lewi's algorithm, as described in Chapter 4.

- **Complete**. No further states need be explored. To determine the final value of the formula, the procedure *FinalValue* is called.

- **AllChildrenExplored**. There are no further paths to be explored from the current state. Procedure *FinalValue* is called to determine the value of the subformula in the current state. If *value* is F, the verification run is terminated because the correctness claim has been violated.

- **Revisit**. Because a state was revisited (it is in the cache), the subformula holds for the current path. This is so because the successor states of the state found in the cache have been explored already. For the subformula $AF\alpha$ this means that α is true somewhere on the current path (in some previously visited state) and

procedure *SetGoodChildren* is called to record this fact. (Procedure *SetGoodChildren* is a very simple procedure that basically sets a flag in state records kept by the state generator.) This information is necessary to handle fairness and is used by procedure *FinalValue* mentioned above.

Procedures *Check* and *TruthValueFormula* handle simple CTL formulae that contain at most one temporal operator (limited to the subset of CTL that was defined for verifying the Gneiss kernel). As explained in Section 5.3 more complex formulae are handled by postponing the evaluation of nested temporal operators. This technique may seem to be rather inefficient because many states may be explored more than once. Fortunately, the unnecessary work can be avoided. In Section 5.1 it was explained that a cache is used to avoid a substantial amount of unnecessary work by detecting revisited states. It turns out that the cache can also be used to avoid the unnecessary analysis of many subformulae.

Recall that the algorithm of Vergauwen and Lewi described in Chapter 4 evaluates CTL formulae in a goal-directed fashion, using mutually recursive procedure calls to evaluate nested subformulae as necessary. The algorithm described here uses essentially the same idea, but *postpones the evaluation of nested temporal subformulae* instead of evaluating them immediately. A subformula and the state in which its value is needed is recorded as a *subproblem* to be evaluated at a later stage. To be able to proceed, the value of the subformula is assumed to be *true*. Many subproblems involve the same subformula, but different states. For each subformula, a list of states is built up in which the value of the subformula is needed. The formula is evaluated in each of these states during a subsequent exploration of the state space.

Evaluating the same subformula in different start states may involve unnecessary work that should be avoided. To explain how this can be accomplished, recall that states are only entered in the cache when they are known to satisfy the formula being checked. This means that when a subformula f must be checked in a set of states $S = s_0, s_1, \ldots$ and any state $s_i \in S$ is found in the cache, the subproblem can be skipped. The recursive definition of CTL formulae in terms of the next-state operator explains why this is so. For example, if the value of a formula $AF\alpha$ is needed in a state s, it can be computed by using the equivalent formula $\alpha \vee AXAF\alpha$. During evaluation of $AF\alpha$ only states that satisfy $AF\alpha$ are entered in the cache. If a state s' is found in the cache the search can be terminated because the formula is known to hold in the next state—state s'. The cache therefore helps to avoid unnecessary work. A verification

```
(* evaluate main formula *)
Logic.Check(sf, satisfied);
(* retrieve first subproblem, if any *)
SubProblems.Get(sf, env, new, more);
WHILE satisfied & more DO
  IF new THEN (* clear the cache *) Storage.Reset END;
  IF (* state not in cache *) THEN
    Logic.Check(sf, satisfied)
  END;
  (* select next subproblem; env = environment, sf = subformula *)
  StateGenerator.SetState(env, sf);
  SubProblems.Get(sf, env, new, more)
END
```

Figure 17: Code to handle nested temporal operators as subproblems.

run involves a number of sweeps of the state space, one for each temporal formula. Note, however, that the cache must be cleared before evaluating each new temporal subformula.

The code fragment shown in Figure 17 handles subproblems. Recall that the evaluation of nested subformulae is delayed as was explained in Section 5.3. Procedure *Check* is used repeatedly to check simple subformulae (containing a single temporal operator) as specified by the parameter *sf*. The result is returned in the parameter *satisfied*. Procedure *Get* in module *SubProblems* returns a new subproblem if possible. The argument *sf* indicates the subformula, *env* is the environment of the subproblem (used to set up the initial state of the state generator), *new* is used to indicate whether it is a new subformula (the cache must be cleared then), and *more* is a flag to indicate whether there are more subproblems to check. Module *Storage* implements the cache. The operation *Storage.Reset* is used to clear the cache for each new subformula that is checked to ensure that all states in the cache always satisfy the current subformula. A simple test then saves a substantial amount of unnecessary work: if a state is in the cache the subproblem can be skipped, as long as the same subformula is involved.

In contrast to the strategy described above, Vergauwen and Lewi's algorithm (described in Chapter 4) evaluates subformulae immediately. The advantages and disadvantages of these two strategies can now be discussed. If subformulae are evaluated immediately and intermediate results are stored, it is reasonably straightforward to check full CTL. Nested formulae are only evaluated when necessary, so no unnecessary work is

done. However, a substantial amount of memory is needed to record these intermediate results—the values of nested subformulae in all states visited. This amounts to $|S| \times length(f) \times 2$ bits of memory, where $|S|$ is the size of the state space, and $length(f)$ is the size of the formula. Two bits are required per formula, since its value can be *true*, *false*, or *undetermined*. The size of problems that can be evaluated is therefore limited by the amount of memory available. The structure of the algorithm further requires that the values of all subformulae be available all the time. Therefore, if the array holding these values fills up because the state space is too large, the verification run cannot proceed.

By postponing the evaluation of nested subformulae it becomes unnecessary to store intermediate results, which reduces the memory requirements. More important, however, is the fact that a verification run is effectively divided into a number of smaller verification runs, one for each temporal formula. This means that the cache is used effectively, since it contains only states that are directly related to the temporal operator being evaluated. It would be possible to replace the large array in the algorithm described in Chapter 4 by a hash table. However, if nested subformulae are evaluated immediately (instead of postponing their evaluation), the cache will fill up faster because many states are visited to evaluate nested subformulae. When the cache fills up, states must be replaced, which means that work will have to be repeated. If the evaluation of nested temporal subformulae is postponed, the cache contains only states that are related to the (single) formula being evaluated.

To measure the amount of memory saved by using the delayed evaluation technique would require implementing Vergauwen and Lewi's algorithm on the same platform so that direct comparisons could be made. However, since the focus of this work was on verification of the Gneiss kernel, this was not attempted.

5.9 Implementation of fairness

Tarjan's algorithm [1, Chapter 5] was combined with the depth-first search algorithm to implement fairness. Instead of using a separate stack to store states that belong to a strongly connected component (SCC) as explained in Section 5.4, a single stack is used to store states belonging to the current path and SCC states. To identify states that are on the current path, they are linked together using a special field to indicate the predecessor of each state. Separate pointers indicate the top of the depth-first stack

and the top of the SCC-stack. States are always pushed onto the stack in the order they are encountered. Instead of storing depth-first numbers, the position of a state on the stack serves the same purpose.

Fairness is implemented by two procedures *Push* (Figure 18) and *Pop* (Figure 19) that are exported by the module *Stack*. Procedure *Push* is called by the state generator for each state generated and *Pop* is called by procedure *Backtrack* to fall back to the current state's predecessor.

For efficiency, the stack that stores states on the current path, the stack needed to store SCC states (for Tarjan's algorithm) and the states in the cache are all contained in one large hash table. The global variable *dftop* indicates the top of the depth-first stack (the stack that contains states of the current path). Because all states are stored in a large hash table, it is necessary to first find the state by computing the hash function. If the state could not be found (on the stack or in the cache), it is new and is pushed onto the stack. The top of the stack that stores the SCC states is stored in *top*. A link is included with each stack entry (the link is called *pre*) to link each state to its predecessor. The global variable *top* is is incremented by *Push*. The field *lowlink* contains the lowest depth-first search number of the state that leads to the current state.

Procedure *Pop* manipulates the same variables discussed for procedure *Push* and uses the *lowlink* field to detect the root state of an SCC. When the root state is found, all states in the SCC are inserted in the cache.

5.10 Testing and evaluation of the ESML verifier

Verifiers are programs that are complex enough to contain subtle errors. Since Promela and ESML are similar modelling languages, systematic tests were conducted using SPIN for comparison purposes. Some tests are included here to show how the basic language constructs were tested[4].

[4]The tests shown here were conducted by Jaco Geldenhuys.

```
PROCEDURE Push(s: State): INTEGER;
VAR pos: INTEGER;
BEGIN
  pos := Find(s); (* find position of state in stack *)
  IF pos = -2 THEN (* not found, s is a new state *)
    stack[top].state := s;
    stack[top].lowlink := top;
    stack[top].pre := dftop;
    dftop := top;
    INC(top);
    RETURN Inserted
  ELSIF pos = -1 THEN (* s found in cache *)
    RETURN Revisited
  ELSE (* s is on the stack *)
    stack[dftop].lowlink := Min(stack[dftop].lowlink, pos);
    RETURN Loop
  END
END Push;
```

Figure 18: Procedure Push that attempts to enter each generated state in the stack. It also implements fairness.

```
PROCEDURE Pop();
VAR pre: INTEGER;
BEGIN
  pre := stack[dftop].pre;
  stack[pre].lowlink :=
    Min(stack[pre].lowlink, stack[dftop].lowlink);
  IF stack[dftop].lowlink = dftop THEN (* root of SCC found *)
    WHILE top > dftop DO
      DEC(top);
      Cache.Insert(stack[top].state)
    END
  END;
  dftop := pre
END Pop;
```

Figure 19: Procedure Pop is called to fall back to the current state's predecessor. It also implements fairness.

```
MODEL Ctrl;

PROCESS p;
BEGIN
  DO TRUE -> SKIP END
END p;

BEGIN
  p
END Ctrl;
```

```
/* control structure */

proctype p()
{
  do :: 1 -> /* skip */ od
}

init {
  atomic {run p() }
}
```

# procs	1	2	3	4
# states	1	1	1	1
# trans	2	3	4	5

# procs	1	2	3	4
# states	2	2	2	2
# trans	3	4	5	6

Figure 20: Typical models used to test ESML control structures. The number of unique states generated and transitions executed are shown below each model for different numbers of processes.

5.10.1 Comparisons to SPIN

To test the correctness of the basic control structures (guarded commands) several small models of the form shown in Figure 20 were written in ESML and Promela. Similar models were used to test the correctness of the IF command of ESML. Since the state spaces for such simple models are small, a comparison of the transitions executed and the unique states generated was feasible and could be checked to be the same. Actually, SPIN always generates one state more than the ESML verifier, but this can be explained easily. The ESML verifier views the initial state as the state generated after executing the code for the main process. In SPIN's case, the initial state is the state *before* the code of the main process has been executed. To test the scheduling of several processes, the same models were executed with several instances of the same process running concurrently. The number of unique states generated and the number of transitions executed are shown below the models. (Promela also has a `skip` statement, but when it was used, the number of states and transitions executed escalated.)

The models shown in Figure 21 were used to test a loop with multiple guards. This test was also repeated with several instances of the same process to test the actions of the scheduler.

```
PROCESS p;
BEGIN
  DO TRUE -> SKIP
  [] TRUE -> SKIP
  [] TRUE -> SKIP
  END
END p;
```

```
# procs    1   2   3   4
# states   1   1   1   1
# trans    4   7  10  13
```

```
proctype p()
{
  do :: 1 -> /* skip */
  :: 1 -> /* skip */
  :: 1 -> /* skip */
  od
}
```

```
# procs    1   2   3   4
# states   2   2   2   2
# trans    5   8  11  14
```

Figure 21: Models used to test an ESML loop with more than one simple guard. The number of unique states generated and transitions executed are shown below each model for different numbers of processes.

```
PROCESS p;
TYPE int = 0..2;
VAR x: int;
BEGIN
  DO x = 0 -> x := 1
  [] x = 1 -> x := 2
  [] x = 2 -> x := 0
  END
END p;
```

```
# procs    1    2    3    4
# states   6   36  216 1296
# trans    9   97  865 6913
```

```
proctype p()
{
  int x = 0;

  do :: (x == 0) -> x = 1
  :: (x == 1) -> x = 2
  :: (x == 2) -> x = 0
  od
}
```

```
# procs    1    2    3    4
# states   7   37  217  1297
# trans    8   74  650  5186
```

Figure 22: Models used to test an ESML loop with more than one simple guard. The number of unique states generated and transitions executed are shown below each model for different numbers of processes.

To test more complex guards the models shown in Figure 22 were used. For more than one process the number of states and transitions executed were too numerous to be checked manually. While the number of unique states generated corresponded, the number of transitions executed did not.

Communication commands in ESML and Promela differ enough to make comparison difficult. For example, the POLL command in ESML allows optional Boolean expressions in the guards, a feature that proved useful in some of the kernel models described in the next chapter. However, the same testing technique was used to compare simple models that include communication commands.

A simple one-way channel was tested using the models shown in Figure 23. Several instances of processes p and q were tested. By studying the output of small models, it was possible to trace the internal operation of the verifiers and explain the differences. In this way it was possible to gain some confidence in the correctness of the ESML verifier. The number of states generated and transitions executed are shown to indicate that communication in ESML is less efficient than in Promela. This is caused by the more elaborate implementation strategy required to support synchronisation between different ESML POLL commands.

Progressively more complex models were used to test communication commands as far as differences between the two modelling languages allowed. Some models are easily translated from ESML to Promela, as shown in Figure 24. Other models, especially models that make use of the optional Boolean expression in the guards of POLL commands, need some rewriting. For example, a Promela version of one of the models developed to verify the Gneiss microkernel is shown in Appendix A[5]. The ESML and Promela versions are quite similar. Macro definitions were used to implement the queue operations of ESML and different channels were used to model selective reception of messages in a POLL command.

5.10.2 Modelling style and the number of states generated

The main limitation of model checking is the state explosion problem. It therefore helps to be aware of the influence of different modelling styles on the number of states generated. In particular, it is necessary to know what to avoid. The simple experiments described in this section illustrate the effect of different ESML constructs on the number

[5]The Promela version of the original ESML model was written by Willem Visser.

```
MODEL Comm;

TYPE ch = {a};
VAR C: ch;

PROCESS q(IN i: ch);
BEGIN
  DO TRUE -> i?a END
END q;

PROCESS p(IN o: ch);
BEGIN
  DO TRUE -> o!a END
END p;

BEGIN
  q(C); p(C)

END Comm;
```

```
/* communication */

mtype = {a}

proctype q(chan in)
{
  do :: 1 -> in?a od
}

proctype p(chan out)
{
  do :: 1 -> out!a od
}

init {
  chan C = [0] of {mtype};
  atomic { run q(C); run p(C) }
}
```

	q		
p	1	2	3
1	6	18	54
	9	40	163
2	12	36	108
	25	109	433
3	24	72	216
	65	277	1081

	q		
p	1	2	3
1	5	9	17
	7	18	46
2	9	17	33
	18	50	130
3	17	33	65
	46	130	338

Figure 23: Models used to test simple communication commands. The upper numbers in each row represent states and the lower numbers transitions for different combinations of processes p and q. Note that the ESML verifier generates more states. This is due to a more complex implementation needed to synchronise communication partners.

```
MODEL PC;
CONST Max = 2;
TYPE Int = 0..Max; Ch = msg(Int);
  Buf = ARRAY[Max] OF Int;
VAR A, B: Ch;

PROCESS Consumer(IN i: Ch);

VAR x: Int;
BEGIN
  DO TRUE -> i?msg(x) END
END Consumer;

PROCESS Producer(OUT o; Ch);

VAR x; Int;
BEGIN x := 0;
  DO TRUE ->
    IF x = 0 -> x := 1
    [] x = 1 -> x := 2
    [] x = 2 -> x := 1
    [] x = 2 -> x := 0
    END; o!msg(x)
  END
END Producer;

PROCESS Buffer(IN i: Ch; OUT o: Ch);

VAR buf: Buf; len: int;
BEGIN
  DO len > 0 ->
    len := len - 1; o!msg(buf[len])
  [] len < Max ->
    i?msg(buf[len]); len := len + 1
  END
END Buffer;

BEGIN



  Producer(A);
  Buffer(A, B);
  Consumer(B)
END PC;
```

```
/* producer-consumer */
#define MAX 2


proctype consumer(chan i)
{
  byte x;

  do :: 1 -> i?x od
}

proctype producer(chan o)
{
  byte x = 0;

  do :: 1 ->
    if :: (x == 0) -> x = 1
    :: (x == 1) -> x = 2
    :: (x == 2) -> x = 1
    :: (x == 2) -> x = 0
    fi; i!x
  od
}

proctype buffer(chan i; chan o)
{
  byte buf[MAX]; byte leng = 0;

  do :: leng > 0 ->
    leng = leng - 1; o!buf[leng]
  :: leng < MAX ->
    i?buf[leng]; leng = leng + 1
  od
}

init {
  chan a = [0] of {byte};
  chan b = [0] of {byte};
  atomic {
    run producer(a);
    run buffer(a, b);
    run consumer(b) }
}
```

Figure 24: A model of a producer-consumer system in ESML and Promela.

```
PROCESS P;
VAR x: BOOLEAN;
BEGIN
  DO TRUE -> x := TRUE END
END P;

BEGIN
  P; P; P (* for 3 instances of process P *)
END
```

# procs	# states
1	4
2	16
3	64
4	256

Figure 25: Effect of increasing the number of processes

of states generated.

Perhaps the most important consideration is to keep the number of concurrent processes to an absolute minimum. The effect of increasing the number of concurrent processes is shown in Figure 25.

As can be seen, the number of states generated increases dramatically as more processes are added. This is so because the processes are independent and therefore every state that is reachable for a given process must be matched up with every state that is reachable for all other processes. Partial order techniques are particularly effective to reduce the number of interleavings to be analysed in such situations.

In ESML, communication commands make processes dependent. Dependence among processes has an inhibiting effect on the size of the state space. The models of the Gneiss microkernel described in Chapter 2 show that rather complex models need not generate unmanageable state spaces. This is due in part to the mutual dependence among processes.

A pipeline of processes illustrate the effect of dependence on the number of states generated. Figure 26 shows the effect of adding more instances of process $P2$ to lengthen the pipeline. By comparing Figure 25 to Figure 26 it can be seen that dependencies among processes reduce the state explosion. The number of states generated does not

```
PROCESS P1(OUT out: Msg);
BEGIN
  DO TRUE -> out!signal END
END P1;

PROCESS P2(IN in: Msg; OUT out: Msg);
BEGIN
  DO TRUE -> in?signal; out!signal END
END P2;

PROCESS P3(IN in: Msg);
BEGIN
  DO TRUE -> in?signal END
END P3;
```

# procs	# states
1	18
2	54
3	162
4	486

Figure 26: Effect of dependence among processes

increase as rapidly as when the processes are independent. This is so because processes must wait on one another and therefore many potential state configurations simply cannot occur.

Concurrency has such a marked influence on the size of the state space because several transitions are enabled simultaneously. Another way to have more than one transition enabled at the same time is the use of nondeterministic choice. This is expressible in ESML as guarded commands where more than one guard expression can be true simultaneously. As can be expected, the size of the state space increases as the number of guarded commands is increased. However, the state explosion is less drastic in this case, as can be seen in Figure 27.

In Figure 27 it can be seen how the size of the state space increases when the number of guarded commands in a control structure is increased. Nondeterministic choice is useful to model decision structures where the exact decision mechanism is irrelevant. For example, this technique is useful to model the environment of a control system. Such environments often generate different signals or events to which control logic must repond. It would be possible to model the environment of a control system as a set

```
DO TRUE ->
  IF TRUE -> x := 1
  [] TRUE -> x := 2
  [] TRUE -> x := 3
  END
END
```

```
DO TRUE ->
  DO TRUE -> x := 1
  [] TRUE -> x := 2
  [] TRUE -> x := 3
  END
END
```

# guards	# states
1	6
2	12
3	20
4	30
5	42
6	56

Figure 27: Effect of increasing the number of guards

of concurrent processes, but a nondeterministic control structure is more efficient. In fact, nondeterministic choice should be used instead of concurrency whenever possible. Another example is a model of a lossy channel. The channel can be modelled as a process that selectively discards some messages. On the other hand, it is better to avoid modelling the channel explicitly as a process. A guarded control structure can be used to select events nondeterministically. Even if only one process can be eliminated by using nondeterminism (or any other technique) it can sometimes reduce the size of the state space significantly. Several examples of the use of concurrency and nondeterministic choice will be described in the next chapter.

5.11 Performance of the verifier

The latest version of the verifier (that was used to check the models presented in the next chapter) was studied and optimised by Jaco Geldenhuys. All important mechanisms that can influence the performance of the verifier were measured carefully. Most results confirm what has been reported in the literature but some results are perhaps surprising and will be mentioned here. The interested reader is referred to [33] for details.

ESML models are translated into abstract code that is interpreted. This decision simplified the verifier substantially, but it is difficult to measure the effect of interpretation

directly. Using a profiler, it was found that for the models measured between 49% and 64% of the time was spent in the interpreter. The rest of the time was spent on analysing the states generated. By studying the essential actions and how these could be performed if interpretation was abandoned, it was found that little, if anything, could be gained.

Strong fairness has been implemented by constructing strongly connected components SCCs). The effect on run time was found to be negligible. Building SCCs was found to have a real impact on memory requirements. For many models the effect is negligible, but for others the memory requirements are doubled.

The compaction technique described in Section 5.2 reduces the state size and memory requirements roughly 5 times, and decreases the run time by a factor of between 0.54 and 0.8 for the models measured.

5.12 Summary

A verification system that is powerful enough to verify a typical microkernel was described in this chapter. A subset of CTL was adopted to express correctness claims and ESML, an experimental modelling language, was developed to express behaviour.

A recursive strategy is used in Vergauwen and Lewi's algorithm to handle nested temporal subformulae, as explained in the previous chapter. A different technique was used here, the basic idea being to postpone the evaluation of nested subformulae. Although only a subset of CTL can be checked in this way, it makes it possible to avoid storing intermediate results, which reduces memory requirements. The technique is compatible with (and depends on) the use of a cache which allows replacement of states. This makes it possible to analyse state spaces that are somewhat too large to fit into memory. A state compression technique was also developed that improves the effectiveness of the cache significantly with the surprising result of simultaneously reducing verification run times.

When the verifier detects an error in a model an error trail (the current execution path that leads to the error) is dumped to a file. The error trail can then be inspected interactively by stepping through the model to determine what is wrong. Each process is displayed in a separate window and the currently active command is marked using colour. A similar facility is available in the SPIN verifier.

To gain some confidence in the correctness of the ESML verifier, it was tested systematically by translating ESML models to Promela and using SPIN as a basis for comparison. In addition, the effects of different ESML constructs on the number of states generated were studied. Based on these results, a number of guidelines will be formulated in the next chapter for reducing the impact of the state explosion problem.

Chapter 6

Verifying a microkernel

Current operating systems are not designed to be verified. However, verification seems essential to meet the most stringent requirements. For example, the success of open systems based on widely accepted communication protocol standards created a security problem with regard to stored and transmitted data. Standards for secure distributed systems are emerging. An example is the *Trusted Computer System Evaluation Criteria* published by the American Department of Defense that can be used to classify systems according to different levels of security. The highest (A1) level requires formal verification. For this, the development of verified designs seems like a necessary first step. This chapter presents a verified design for a microkernel—a fundamental component of many current operating systems.

As described in Chapter 2, a microkernel is a typical reactive program that interacts with its environment via kernel calls and a mechanism to field interrupts. Verified models of various components of a microkernel were constructed: a scheduler, interprocess communication, kernel call and interrupt mechanisms, and device drivers.

The microkernel presented here as a case study is a practical program that is used in industry. I was fortunate to be in charge of its development and verification was kept firmly in mind during the design phase. The project was only possible with the help of a number of other people and since the outcome of a software project depends on its history, some background is provided here.

Werner Fouché and I designed the microkernel and Werner implemented the first version in Modula-2 [30, 29]. Harry Lewis extended this program by writing additional

device drivers. Meanwhile, I designed and implemented the first version of the verification tool described in Chapter 5 [18]. Willem Visser and I designed the modelling language ESML [21] and Willem wrote the first ESML compiler [73] to produce a useful verification system. Hans Loedolff, Jaco Geldenhuys and I rewrote the entire verifier to improve its performance and internal structure. Pieter Muller rewrote the microkernel in Oberon, improving some mechanisms where necessary. For example, the object-oriented facilities of the Oberon language made it possible to develop a hierarchy of extensible device drivers [56]. While this improved version of the kernel (called "Gneiss") was implemented, I developed and verified models of various mechanisms. These models are presented here.

6.1 Modelling basic mechanisms

The verified models of the microkernel consist of a number of mechanisms that seem to form part of most reactive systems. The identification of components that are fundamental in some sense, combined with a theory of concurrency, may lead to a practical methodology for the design of concurrent reactive systems.

Various theories of concurrency have been developed of which the process algebraic approach, and specifically CSP, is most relevant here. CSP provides a theoretical framework to describe concurrent mechanisms [41]. The notation is useful to define various components of reactive systems such as pipes, buffers and protocols. However, to model practical reactive systems, a strongly typed modelling language is more suitable. The modelling language ESML—which is based on CSP—is used here to present a number of useful mechanisms.

Pipes, buffers, filters and protocols. A *pipe* is a process with one input channel and one output channel. Two pipes may be chained together by connecting the output channel of one process to the input channel of another to form a more complex pipe. There is one additional constraint: the two processes must be compatible regarding the messages transmitted between them. In ESML that means that the two channels connected together must have the same type.

A *buffer* is a pipe that transmits on its output channel all messages received on its input channel without reordering them. Two or more buffers can be combined to produce a chain of buffers. Buffers introduce extra overhead, but are useful to store messages when they cannot be accepted immediately. Here is an example of a buffer that handles

a stream of numbers ranging from 0 to 10:

```
TYPE INT = 0..10; Stream = {int(INT)};

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

A *filter* is a pipe that modifies its input in some way. Bit stuffing, for example, can be modelled as a filter. Bits received on the input channel appear unchanged on the output channel, except that after a given number of consecutive 1-bits an extra 0 is inserted. In ESML this filter can be modelled as follows:

```
TYPE Binary = {Bool(BOOLEAN)};

PROCESS Filter1(IN in: Binary; OUT out: Binary);
VAR bit: BOOLEAN; count: INT;
BEGIN count := 0;
  DO TRUE -> in?Bool(bit);
    IF bit = 0 -> count := 0
    [] (bit = 1) & (count < 6) -> count := count +1
    [] (bit = 1) & (count = 6) -> out!Bool(0)
    END;
    out!Bool(bit)
  END
END Filter1
```

On the receiver side, the extra 0 bits can be removed by the following filter:

```
TYPE Binary = {Bool(BOOLEAN)};

PROCESS Filter2(IN in: Binary; OUT out: Binary);
VAR bit: BOOLEAN; count: INT;
BEGIN count := 0;
  DO TRUE -> in?Bool(bit);
    IF bit = 0 -> count := 0
    [] (bit = 1) & (count < 6) -> count := count +1
    [] (bit = 1) & (count = 6) -> in?Bool(bit)
    END;
    out!Bool(bit)
  END
END Filter2
```

Filters behave like total functions, and consequently, the effect of a filter can be undone by composing it with its inverse. This explains why a filter composed with its inverse behaves like a buffer. Note that the two filters shown above are not inverses of each other.

A *Protocol* is another fundamental component of reactive systems. Many protocols can be modelled as a number of layers and behave like buffers, since their purpose is to transmit data faithfully without modification. Filters and their inverses play an important role in protocols. For example, when data must be transmitted in encrypted form, encryption and decryption can be modelled as two filters that are inverses of each other. This means that data transmitted over the physical communication medium will be encrypted, while the protocol as a whole behaves like a buffer—messages are transmitted without modification. Apart from correctness, the main advantage of specifying each component of a complex protocol by means of a verified model is that different implementations of the same component are interchangable.

A number of basic mechanisms that are particularly relevant to microkernels will now be described. An introduction to many other mechanisms found in concurrent programs can be found in Andrews [2].

Clients and servers. 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. For example, a synchronous server is useful to model a circular buffer:

```
PROCESS Buffer(IN in: Stream; OUT out: Stream);
CONST number = 5; max = number-1;
TYPE Items = ARRAY number OF INT;
VAR first, last, count: INT; buf: Items
BEGIN
  first := 0; last := 0; count := 0;
  DO TRUE ->
    POLL in?int(buf[last]) & count < number ->
       last := (last + 1) MOD max;
       count := count + 1
    [] out!int(buf[first]) & count > 0 ->
       first := (first + 1) MOD max;
       count := count + 1
    END
  END
END Buffer
```

The buffer only accepts new input if there is some space left and it is always ready to output an item if it is not empty. When the buffer is neither full nor empty, it may choose nondeterministically between an input operation or an output operation.

A more complex pattern of interaction between clients and a synchronous server is sometimes necessary. A client may supply data and wait for a response from the server. The general outline of a such a synchronous server is given below.

```
PROCESS SyncServer(IN in: Request; OUT out: Response);
BEGIN
  DO TRUE ->
    POLL in?requestA -> out!responseA
    [] in? requestB -> out!responseB
    [] ...
    END
  END
END SyncServer
```

It is quite acceptable to block a client process while a short operation is completed. Unfortunately, some operations, like the typical operations executed by peripheral devices, are inherently slow. A slightly more complex kind of server—an *asynchronous* server—is needed to handle such operations. 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. This design can be modelled as follows:

```
PROCESS AsyncServer(IN in: Request; OUT out: Response);
VAR currentOperation: INT;
BEGIN
  DO TRUE ->
    POLL in?request0 -> currentOperation := 0
    [] in?request1 -> currentOperation := 1
    [] ...
    END;
    (* wait for completion signal *)
    in?completionSignal;
    IF currentOperation = 0 -> out!response0
    [] currentOperation = 1 -> out!response1
    [] ...
    END
  END
END AsyncServer
```

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.

```
PROCESS AsyncServer(IN in: Request; OUT out: Response);
TYPE RequestList = LIST[max] OF Request;
VAR currentClient: INT; pending: RequestList;
BEGIN
  DO TRUE ->
    POLL in?request(id) & ~busy ->
       currentClient := id;
       busy := TRUE;
      (* start operation *)
    [] in?request(id) & busy -> pending := pending::<id>
    [] in?completionSignal ->
       (* activate current client *)
       IF LEN(pending) = 0 -> busy := FALSE
       [] LEN(pending) > 0 ->
          currentClient := HEAD(pending); REMOVE(pending);
          (* start operation *)
       END
    END
  END
END AsyncServer
```

Because of their simplicity, synchronous servers are preferable in all situations where operations require little time to complete. An asynchronous server should be used only when operations are inherently slow.

Schedulers. Most reactive systems need some way of allocating shared resources among waiting users. This is known as the scheduling problem. A *monitor* is a well-known example of a scheduler that enables a number of processes to share a resource. Hoare's classical paper [40] introduced the concept and the first implementation is due to Brinch Hansen [6]. Monitors provide mutual exclusion and condition synchronization in a structured way. Wherever data must be protected against concurrent access in a kernel, a monitor should be considered instead of unstructured low-level mechanisms like semaphores, as suggested by Andrews for example [2].

Scheduling is often associated with processor management. Nonetheless, all schedulers are similar, no matter what the nature of the shared resource may be. Basically, a suitable data structure is needed to keep track of waiting clients and a selection policy to choose the next candidate to serve. For reasons of efficiency, many different scheduling algorithms have been devised to address different situations. The following is an example of a scheduler modelled as a synchronous server. It allocates a resource to the longest waiting client.

```
CONST max = 2;
TYPE Client = 0..max;
  Clients = {new(Client), old(Client)};

PROCESS Scheduler(IN in: Clients; OUT out: Clients);
```

```
TYPE Waiting = LIST max  OF Client;
VAR q: Waiting; client: Client;
BEGIN q := <>;
  DO TRUE ->
    POLL out!new(HEAD(q)) & LEN(q) > 0 -> REMOVE(q)
    [] in?old(client) -> q := q::<client>
    END
  END
END Scheduler
```

A queue of clients wait to gain access to a shared resource. Clients are numbered from 0 to 2. When requested, the scheduler selects a new client (the first one in the queue) to be served. Clients that release the resource are entered at the back of the waiting list.

An important requirement for schedulers is *fairness*. Roughly, this means that all requests must be completed eventually. Some scheduling policies are efficient, but unfair and designers should be aware of this. For example, using the shortest seek time as a basis for disk head scheduling is efficient, but unfair. A fair disk head scheduling policy is the elevator policy that is designed to keep disk heads moving in the same direction as long as possible. The direction of movement is only reversed when all requests in the current direction of movement (inwards or outwards) have been serviced. Unfortunately, requests that arrive when the read/write heads have just passed a certain cylinder take longer to complete. For processor allocation similar scheduling policies exist. A simple and fair policy is round-robin scheduling. Various more sophisticated policies, such as priority scheduling, are aimed at providing improved response to short important requests. Ensuring fairness usually involves some overhead and the kind of fairness determines how much. Good descriptions of classical scheduling policies may be found in standard textbooks on operating systems, such as [65].

6.2 General guidelines for modelling and design

It is easier to understand a non-trivial program when it is structured in such a way that its different components can be described and studied separately. The main advantage of abstraction is that the details of the components can be ignored when studying their interaction. Good designs are easy to describe and understand, but should also allow an efficient implementation. Many books have been written about software design. Broadly speaking, there are currently two approaches: functional design and object-oriented design. In functional design, the focus is on *algorithms*, while in object-oriented

design the *data structures* manipulated by a program are placed in the foreground. Both techniques are useful, but the resulting programs may be structured quite differently.

Functional design leads to well-structured, efficient programs. On the down side, such programs are somewhat resistant to change because algorithms and data are closely integrated. Functional design thus works best for programs of small to medium size that must be efficient and will need little after-the-fact modification.

In object-oriented design, reusable abstractions (data and closely associated algorithms) are grouped together and decoupled from other components relying on them. Object-oriented programming languages provide mechanisms to extend basic reusable mechanisms. This allows extra flexibility, but unfortunately the unavoidable extra levels of indirection introduce some inefficiency. Nevertheless, object-oriented techniques are useful where adaptability is mandatory. While some overly enthusiastic supporters of object-oriented design may think otherwise, it should be kept in mind that the technique can only be applied sensibly where complex data structures are present. In [53] Mössenböck presents a balanced introduction to object-oriented design, pointing out the advantages and disadvantages of the technique. Roughly, the choice is between flexibility and efficiency.

Designing and modelling requires creativity. Sometimes a formal specification can suggest a possible design, but it is difficult to derive a good design from a specification in a mechanical way. Although general guidelines can help, design and modelling skills are mostly acquired through experience. No set of rules can be devised that will immediately turn an inexperienced person into an expert. Nonetheless, since it is important to know what to avoid and what to aim for, a number of common mistakes and general guidelines will be pointed out.

Our goal is to develop a *verified* design of a microkernel. For this, we need to combine functional and object-oriented design techniques with formal modelling and verification. In [44, Chapter 2] Holzmann gives general rules for designing protocols that are equally applicable to the design of operating systems. These rules reflect a considerable amount of experience and are therefore adopted here, with minor changes to make them less protocol-specific:

1. *Make sure that the problem is well-defined. All design criteria, requirements and constraints, should be enumerated before a design is started.*

2. *Define the service to be performed at every level of abstraction before deciding*

*which structures should be used to realise these services (*what *comes before* how.*)*

3. *Design* external *functionality before* internal *functionality. First consider the solution as a black-box and decide how it should interact with its environment. Then decide how the black-box can be organised internally. Likely it consists of smaller black-boxes that can be defined in a similar fashion.*

4. *Keep it simple. Fancy designs are buggier than simple ones; they are harder to implement, harder to verify, and often less efficient. Problems that appear complex are often just simple problems huddled together. Our job as designers is to identify the simpler problems, separate them, and then solve them individually.*

5. *Do not connect what is independent. Separate orthogonal concerns.*

6. *Do not introduce what is immaterial. Do not restrict what is irrelevant. A good design is "open-ended," i.e., easily extensible. A good design solves a class of problems rather than a single instance.*

7. *Before implementing a design, build a high-level prototype—a model—and verify that the design criteria are met.*

The first rule stipulates that constraints should be known up front. These are often determined by the implementation environment. It is good practice to select a system model to match the architectural constraints of the implementation environment. Most systems can be structured and modelled in radically different ways. For example, it is important to decide at an early stage whether the design will be centralised or distributed. The choice is usually determined by the hardware architecture at hand. Most reactive systems are made up of several concurrent processes and the popular client-server model is often appropriate, although other models may sometimes be preferable. The advantage of selecting a well-known system model is that properties such as efficiency, reliability and adaptability will be known in advance. Fortunately, it is seldom necessary to start from scratch because tried and tested designs for concurrent systems can be found in standard textbooks such as [2]. An experienced designer knows many different models to choose from.

The warning contained in rules 2 and 3 is to focus on correctness and leave efficiency issues until later. Elegant designs that can be modelled and verified are often discovered by ignoring irrelevant detail. The main goal should be to develop a clean abstract representation that will simplify reasoning about the system. Once this has been achieved,

refinement of a model and its implementation should proceed hand in hand, using one as a check against the other. Design is an iterative process. As problems are uncovered and new insight gained, it may be necessary to rework some ideas until an acceptable structure emerges.

It often helps to model several aspects of a system before attempting a final design. Designing a system is a process of selecting the most suitable from a number of alternative solution strategies. The system specification defines the goal and reasoning leads the way. As design decisions are taken, a final verified design will gradually crystallise out of initial vague ideas. Whether to work in a top-down fashion or bottom-up, is not always clear. A top-down strategy may lead to the discovery that some lower level mechanism cannot be implemented efficiently. On the other hand, working one's way upwards from a set of efficient low level mechanisms can produce awkward designs unless a global picture is kept in mind. The conventional design process is rather error-prone because ideas can only be tried out by implementing them. Prototype implementations help, but errors can still go undetected until everything has been cast in concrete. Modelling and verification offers a better alternative: new ideas are modelled and verified as they occur during the design process, modelling being a more reliable check on correctness than prototyping.

A good strategy is to gain insight by studying simple models. This normally leads to the discovery of subsystems and gradually a series of verified models of increasing sophistication will be produced. Some will be discarded, but what remains will be the "prototype" mentioned in Holzmann's rule number 7.

While developing a model, it is crucial to avoid constructs that are known to cause a state explosion during verification. Various ESML constructs affect the number of states generated, as was discussed in Chapter 5. This suggests a number of additional rules, the rules of modelling:

1. *Include only data structures that can influence the future responses of every component of a model—actions visible across interfaces.*

   The general idea is to analyse intricate concurrent mechanisms—these often contain the most subtle logical design errors. Therefore, we should focus on modelling *control flow* and try to represent data as abstractly as possible. For each component of a model, we are only interested in data that can influence its externally observable actions. Other data should be excluded or represented in simplified

form. For example, when modelling interprocess communication, the contents of the messages can be ignored.

2. *Restrict the number of concurrent processes to the absolute minimum.*

   As was shown in Chapter 5, the number of concurrent processes has a marked effect on the number of states generated. If events are mostly independent, partial order techniques can reduce the state explosion due to different interleavings of events, but in general it is sensible to restrict concurrency as much as possible. When there is more than one instance of the same process, it is often unnecessary to model them all. For example, it may be enough to show that the interaction between a server and one or two client processes is correct. Modelling many client processes may contribute nothing to the analysis and will only increase the number of states generated.

3. *If possible, eliminate buffers by using direct synchronous message passing between processes or restrict the size of buffers to the minimum size that allows correct operation.*

   Buffers are typically used to improve throughput (at the cost of increased response time) by decoupling concurrent processes from each other. However, the number of states generated will also be increased. When large buffers are used in combination with several concurrent processes, the effect is a severe state explosion. Whenever feasible, synchronous, unbuffered communication is preferable to asynchronous communication. This limits concurrency and therefore the number of states generated. Such systems are also simpler to implement.

4. *Beware of seemingly simple processes that generate huge state spaces through manipulation of data variables.*

   Some processes may not interact visibly with other processes, but can generate huge state spaces because each different data value represents a different state. In particular, clocks and counters should be avoided. The following is an extreme example:

```
value := 0;
DO counter < max -> counter := counter + 1
[] counter = max -> counter := 0
END
```

5. *Use nondeterminism to model a decision structure when the alternative actions are important, but the decision mechanism is irrelevant.*

An IF construct with more than one TRUE guard can be used to capture alternative actions when the choice is arbitrary. For example, a server that can display different visible actions, depending on some irrelevant internal mechanism (such as whether a buffer is full or not), can be modelled by the following ESML fragment:

```
IF TRUE -> appropriate action when buffer is full
[] TRUE -> appropriate action when buffer is not full
END
```

It is thus unnecessary to represent the data that drive the decision mechanism explicitly. This technique can lead to a significant state space reduction. The number of states generated increase with the number of guards, but the blow-up is manageable as was shown in Chapter 5. However, it should be kept in mind that nondeterminism is resolved according to the fairness policy supported by the verifier. Special care is needed to ensure correctness in a final implementation. Consider, for example, a model of access control mechanisms to a critical section. Depending on the level of detail in the model, a verifier that implements strong fairness may not show up potential starvation of processes. This is so because selection of any process immediately disables the others. Some transitions are therefore enabled infinitely often, and strong fairness guarantees that they will eventually be selected for execution. If no mechanism is modelled explicitly to prevent starvation, a verifier that implements strong fairness will not find this potential problem.

6. *Divide complex designs into clearly defined subsystems that can be verified separately.*

   A complete design is usually too complex to verify because there are too many patterns of interaction. It helps to find a structure that can be decomposed into a number of subsystems. Depending on the structure of the overall system model, it may be possible to "zoom in" on one subsystem at a time by replacing irrelevant parts of a model by simplified equivalents. This can reduce the number of states generated significantly.

7. *Reuse verified designs that solve important general problems.*

   Reactive systems are expensive to develop, one reason being that new systems are usually designed from scratch. Such expensive *ad hoc* development techniques

may be avoided by adapting successful designs to suit slightly different (but similar) situations. Unfortunately the source code of a non-trivial reactive system is usually too detailed to be reused effectively. Moreover, most reactive systems are not documented well enough to allow effective reuse of successful designs. However, there is a viable alternative: successful mechanisms can be distilled from existing systems and presented as verified models. Such verified models would represent an invaluable "tool kit" of reusable designs for the development of new systems.

6.3 The architecture of a microkernel

Cattel describes a first attempt at verifying a real-time kernel [10]. The verification system SPIN was used and a subtle error was discovered. However, the paper does not describe how the models were developed. What is needed is a design discipline that can be followed to produce provably correct designs. Holzmann wrote the first book on the design and verification of protocols [44]. A tutorial on the use of the verification tool SPIN and the modelling language PROMELA can be found in [32]. It is argued that any good engineering discipline should use prototypes to verify decisions and to predict essential characteristics of products before they are implemented.

The verification of the microkernel presented here should be seen as a first step to develop a similar discipline for designing operating systems. Since the verification of an entire operating system is a formidable task, it seems necessary to concentrate on the verification of single important components first. An overview of the proposed method is presented in [20].

The Gneiss microkernel described here was designed as a set of interacting servers. Without such a formal structuring mechanism, the complexity of the task would be beyond the capabilities of currently available verification techniques. It is significant, however, that the design could still be implemented efficiently without deviating from the verified design. Microkernels present the additional problem that their actions depend on rather complex internal data structures. ESML—the modelling language used to verify the Gneiss microkernel—was designed to address this problem, as was explained in Chapter 5.

The experiment described here had a well-defined goal: to determine to what extent the mechanisms found in a typical microkernel can be verified and described as reusable

designs. Answers were sought to the following questions:

- Can reusable verified designs be developed for mechanisms found in microkernels?
- Can current verification techniques cope with the state explosion encountered when verifying typical microkernel mechanisms?
- Which constructs are needed in a modelling language for this kind of system?
- Is it possible to find a design for a microkernel that can be verified but which will also allow an efficient implementation?
- Is it possible to develop a design discipline that will also work for other operating system components?

It seemed natural to model a microkernel as a collection of interacting servers. The Gneiss microkernel consists of four subsystems: a memory manager, processor manager, interprocess communication subsystem and drivers for peripheral devices. The memory manager is a synchronous server that allocates and deallocates physical memory. The processor manager is modelled as two interacting servers. One controls the currently active process by responding to kernel calls and interrupts. It interacts with another server—the scheduler—to select the next process to activate. Interprocess communication is a simple client-server protocol and device drivers are modelled as asynchronous servers. The various models and the major design decisions are described in the following sections.

A firm decision was taken about the relative importance of some conflicting software properties: simplicity was placed first because complex code is usually unreliable and simple systems are easier to verify. Efficiency was placed second because nobody would like to use an inefficient microkernel. In addition, it would balance the quest for simplicity by ruling out a trivial design. Adaptability was considered least important since it is often in conflict with efficiency. Fortunately, apart from adding new device drivers, microkernels are seldom modified.

Basic memory management, process management and interprocess communication must be as efficient as possible. Functional design was therefore selected as the appropriate method. However, object-oriented techniques were used to design device drivers to simplify the process of adding new drivers to the kernel. Since peripheral devices are

slow compared to processor speeds, the extra overhead imposed by an object-oriented design is negligible in the case of most device drivers [56].

An important decision was the kind of message passing to use. Asynchronous message passing allows a process to proceed directly after sending a message to maximise concurrency. However, the unavoidable added complexity of message buffering would violate our most important requirement—simplicity. Synchronous message passing is simpler to implement and requires no buffering of messages because the sender of a message is blocked until the message has been received. Also, unbuffered synchronous communication reduces the state explosion problem and would thus simplify verification of the kernel.

Designing is a process of reasoning and discovery: reasoning begins with a set of requirements in mind and gradually the required functionality is discovered. The development process, rather than the complete design, is described here. The final verified models are shown in Appendix A.

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. At first it was ignored that the currently active process is associated with some VM. The following model captures the basic 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`. Actually the kernel 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. When modelling the

kernel mechanisms, such detail are not relevant.

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 verification system to detect such errors.

This simple model was easily verified 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. Other user processes could therefore be served while some slow operation on a peripheral device is in progress. To do this, it was necessary to modify **Running** to respond to kernel calls as well as interrupts.

```
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 first 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 creation of user processes was not modelled explicitly though.

```
PROCESS User(id: procid; IN in: Resume; OUT out: KernelCall);
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.

```
PROCESS DeviceDriver(OUT toready: readyrequest;
  IN request: iocommand; OUT todevice: devicecommand;
  OUT touser: continue);
CONST qmax = 2;
VAR curproc, id: procid; deviceidle: BOOLEAN;
BEGIN
  deviceidle := TRUE;
  DO TRUE ->
    POLL request?doio(id) & deviceidle ->
        curproc := id;
```

```
        todevice!startio;
        deviceidle := FALSE
    [] request?iocomplete ->
        touser!resume(curproc);
        toready!enterproc;
        deviceidle := TRUE
    END
   END
END DeviceDriver;
```

The device controller hardware is modelled as a simple synchronous server that accepts IO commands from the device driver. It generates an interrupt for every request:

```
PROCESS Device(OUT torunning: runningrequest;
  IN request: devicecommand);
BEGIN
  DO TRUE ->
    request?startio;
    torunning!int
  END
END Device;
```

Some peripheral devices work more efficiently when a queue of waiting requests is maintained. To exploit this idea, a request queue was added to the device driver. 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. The following model shows how the list construct of ESML was used to model a request queue:

```
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 := HEAD(rq);
            REMOVE(rq);
            todevice!startio
        END
    END
  END
END DeviceDriver;
```

These refinements led to the model of global kernel interaction shown in Appendix A. To make further verification possible and to simplify implementation, the design was divided into subsystems, each consisting of a number of functionally related components. The first of these—the kernel proper—is centered around the currently active process. The key components have already been encountered. These are **Running**, that manages the immediate environment of the currently running process, and the scheduler **Ready** that selects a new process when necessary. The Gneiss scheduler was developed by a process of exhaustive case analysis. The requirements were analysed in a systematic way, the final ESML model corresponding closely to the implementation code. This is described in Section 6.6. Although the memory manager could be modelled as a synchronous server, this was not considered worthwhile because it is rather simple. The memory manager maintains a list of blocks of available memory and this list is searched on request, using a first fit algorithm. More interesting is the subsystem that supports communication between clients and server processes. It manages communication ports and each port may have an associated queue of requests waiting to be processed. Device drivers form a third subsystem. Each device driver is designed as a kernel-resident server that responds to requests from the kernel proper. The refinement of these subsystems will be described next.

6.4 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*.

The communication mechanism used in the Gneiss kernel was modelled based on 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 kernel calls `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 by means of an array `Partner`. The command `toReady!enterProc(s)` is used to enter the server process in the scheduler queue.

```
IF clients OR LEN(pq) = 0 -> (* no server is ready *)
     APPEND(pq, cp); clients := TRUE
[] ~clients & LEN(pq) > 0 -> (* a server is ready *)
     sp := HEAD(pq); REMOVE(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] := HEAD(pq); REMOVE(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 activated. The status code that is returned to the client to reflect the outcome of the operation requested is not modelled explicitly.

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

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. Although this approach is not defensible from a purely logical perspective, it is currently the only reasonable approach where efficiency is important. To illustrate this technique, part of the model that handles the `Transaction` primitive is shown in Figure 28. The corresponding implementation for this part of the model is shown in Figure 29. The code has been simplified somewhat to make it easier to understand, basically by eliminating error handling code. Structurally, however, nothing has been changed.

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 in Figure 29 implement lines 10–11 of the model shown in Figure 28.

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.

```
1  PROCESS Comms(IN in: A0; OUT toReady: A2);
2  CONST NIL = 0;
3  TYPE Map = ARRAY[maxProc+1] OF ProcID;
4  VAR cp, sp: ProcID; pq: ProcQ; clients: BOOLEAN; Partner: Map;
5  BEGIN
6    clients := FALSE;
7    DO TRUE ->
8      POLL
9        in?Trans(cp)-> (* Transaction primitive *)
10         IF clients OR LEN(pq) = 0-> (* no server is ready *)
11           APPEND(pq, cp); clients := TRUE
12       [] ~clients & LEN(pq) > 0 -> (* a server is ready *)
13           sp := HEAD(pq); REMOVE(pq);
14           Partner[cp] := sp;
15           Partner[sp] := cp;
16           toReady!enterProc(sp)
17       END
18   [] in?RecReq(sp) ->
19      ...
```

Figure 28: Part of the model for inter-process communication showing the Transaction primitive

```
1  PROCEDURE LocalTransaction(pid: GPorts.PortID;
2     req, rep: Message; 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
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, server *)
29       sbs.req.len := req.len;  (* number of bytes copied *)
30       GReady.EnterProcess(sp);  sp := NIL (* unblock server *)
31     END;
32
33     (* Phase 3: Finalisation *)
34     IF res # 0 THEN (* error occurred in phase 2, undo *)
35     ELSE (* everything ok, block the client process *)
36       cp := GRunning.PreemptProcess(ContTransaction);
37       cp.bs := cbs;
38     END
39   END; RETURN res
40 END LocalTransaction;
```

Figure 29: Oberon code to show how the `Transaction` primitive was implemented.

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. The complete model of the communication subsystem is shown in Appendix A.

6.5 Device drivers

To add a new device driver to a typical operating system, a programmer must fully understand (1) the device controller hardware and (2) the mechanisms through which device drivers interact with the rest of the system. The former kind of knowledge is needed for every new device. However, the interface that defines the interaction between device drivers and the rest of the system should be designed to hide irrelevant detail. This idea was exploited to develop families of similar device drivers for the Gneiss microkernel [56]. All device drivers have a similar structure. The most intricate mechanisms common to a family of similar devices have been combined and hidden inside low-level modules that are reusable without recompilation.

As an example, a disk driver is presented here. The main ideas behind the driver and the implementation will be discussed before showing how it was modelled. The complete model for the disk driver is shown in Appendix A.

The driver consists of two components: a generic module and a device specific module. The reusable generic component was modelled and verified. The control flow of the device specific component is trivial. It contains device specific procedures that are called by the generic module. True to the spirit of object oriented design, the device specific module extends the generic module by adding detail that differs from one disk controller to the next. The family of disk drivers share common mechanisms that reside in the generic module. Request messages are accepted from clients and handled one at a time. A queue of requests is maintained to improve performance. When the interrupt arrives that signals completion of a peripheral operation, the waiting client process is reactivated and the next peripheral operation (if any) is started. A status code reflects the outcome of each operation.

The generic module `Disk` contains two handler procedures `TransactionHandler` and `InterruptHandler` that are called by mechanisms in the rest of the kernel at the appropriate times. Each time an IO request arrives, procedure `TransactionHandler` in `Disk` is activated. Similarly, an interrupt from the disk controller activates procedure `InterruptHandler` in `Disk`. Interrupts are associated with hardware interrupt numbers

(managed by the kernel component `Running`) and transactions are associated with port numbers (managed by module `Comms`). Modules `Running` and `Comms` each provide a procedure `InstallHandler`. These procedures are called during initialisation to bind the procedure `InterruptHandler` of each device driver to a specific interrupt number and procedure `TransactionHandler` to a specific port number. Further details of interrupt handling and transaction handling are irrelevant here. The binding of specific handlers to specific interrupt numbers and ports was not modelled explicitly, but the rest of the control flow is reflected in the model.

To illustrate the correspondence between the model and implementation of the disk driver, various implementation code fragments (written in Oberon [50]) are included. Here is the (simplified) code of `TransactionHandler` showing two operations: `Read` and `Write`:

```
PROCEDURE TransactionHandler(d: Running.Object;
  req: Comms.Message): LONGINT;
VAR result, time: LONGINT; currentproc: Procs.Process; params: Params;
BEGIN
  WITH d: Controller DO
    CASE Operation(req) OF
      Read, Write:
        currentproc := Running.PreemptProcess(NIL);
        params := Parameters(req); currentproc.bs := params;
        IF d.activeproc = NIL THEN (* idle, start operation *)
          d.activeproc := currentproc;
          result := d.start(d, params, time);
          IF result = ok THEN
            TObjects.SetTimeout(d.activeproc, time, TimeoutHandler)
          ELSE Wakeup(d, result)
          END
        ELSE (* busy, queue request *)
          TObjects.Put(d.waiting, currentproc)
        END
    ELSE result := error
    END
  END
END TransactionHandler;
```

The procedure `Transactionhandler` operates on an object called `d` that is imported from module `Running`. The current process (that requested the IO operation), is preempted, which includes saving its state. If the request queue is empty (`d.activeproc = NIL`) the IO operation is started immediately, but otherwise the request is queued. If the operation has been started successfully, a timeout is set on the process as a guard against lost interrupts due to device malfunction. Otherwise, procedure `Wakeup` is called to enter a status code in the state record of the client process and reactivate it. The duration of the timeout depends on the controller and is therefore supplied by

the device specific module. Process state records are "transient objects" managed by module `TObjects`. The two handler procedures of the disk driver are modelled as a POLL with two guards. Here is a model of the transaction handler:

```
POLL in?IOrequest(currentProc) ->
  IF activeproc = NIL -> (* idle, start operation *)
     activeproc := NIL; toST506!Start
  [] activeproc # NIL -> (* busy, queue request *)
     APPEND(requestQ, currentproc)
  END
[] (* Interrupt *)
END
```

When an IO-completion interrupt arrives, procedure `InterruptHandler` is activated. The process waiting for IO to complete will be activated by calling procedure `Wakeup` as shown below:

```
PROCEDURE InterruptHandler(d: Running.Object);
VAR more: BOOLEAN; result: LONGINT;
BEGIN
  WITH d: Controller DO
    IF d.activeproc # NIL THEN (* a request is active *)
      result := d.transfer(d, d.activeprocs.bs(Params), more);
      IF (result # ok) OR ~more THEN
        TObjects.CancelTimeout(d.activeproc);
        Wakeup(d, result)
      END
    END
  END
END InterruptHandler
```

Procedure `Wakeup` enters the process (and the status code) in the scheduler queue. If the device driver's request queue contains other requests, the next one will be started.

```
PROCEDURE Wakeup(d: Controller; result: LONGINT);
BEGIN
  SetStatusCode(d.activeproc, result);
  Ready.EnterProcess(d.activeproc); d.activeproc := NIL;
  IF ~TObjects.Empty(d.waiting) THEN (* start next request *)
    ... (* similar code in TransactionHandler *)
END Wakeup
```

The time server in Gneiss maintains a queue of future times at which a given activity must be activated. Although timeouts can be modelled in ESML by using nondeterminism, the time server is so simple that it was not modelled explicitly here. In Gneiss, timeouts are only used to prevent device drivers from waiting for ever for an expected

interrupt because of a hardware error. The functions of procedures `InterruptHandler` and `Wakeup` could be combined in the following ESML fragment:

```
POLL (* IO request *)
[] in?Interrupt ->
     toST506!Transfer;
     toClient!EnterProc(activeproc);
     IF LEN(rq) # 0 -> (* handle next request *)
         activeproc := HEAD(rq); REMOVE(rq);
         toST506!Start
     [] LEN(rq) = 0 -> (* no more requests queued *)
         activeproc := NIL
     END
END
```

The disk driver is modelled by the two processes `Disk` and `ST506`. The former represents the generic component, and the latter the device specific component. The complete model of the disk driver is shown in Appendix A.

6.6 Refining the scheduler specification

In the models described so far, VMs played no role. A VM in the Gneiss microkernel has a protected address space that may be shared by a number of (light-weight) processes. This was described in Chapter 2. Processes and VMs are scheduled by the kernel; VMs are time-sliced, while processes are scheduled non-preemptively. In this section the technique used to develop the Gneiss scheduler is explained. The major data structure used by the scheduler is shown in Figure 30.

The scheduler is a state machine whose actions are determined by its current state (such as queues of VMs and processes that are ready to run) and external events (kernel calls and interrupts). A specification was developed by first deciding on a number of scheduling conditions, each a function of the scheduler's state:

- Must the current VM continue to run? This is indicated by the VM's state, which can be "stop" or "proceed". The state of a VM may be modified by events external to the scheduler, like a timer interrupt.
- Is the process queue of the current VM empty? If not, the next process in the queue is selected (unless the state of the current VM is "stop"), but if the process queue is empty, another VM must be selected.

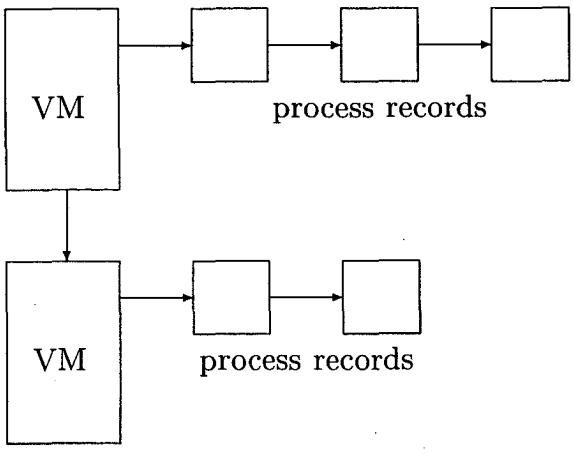


Figure 30: The scheduler queue

- What is the state of the current process? If the current process is not the null process, its state must be saved in such a way that it can be reactivated at a later stage. Different actions are required for interrupts and kernel calls. After a kernel call, the process record is entered at the back of the current VM's process queue. However, when an interrupt occurs, the state of the interrupted process must be saved in such a way that it (and no other process) will be reactivated when the VM is restarted later. This scheduling principle eliminates the need for explicit synchronisation primitives between processes that share the same address space. The process record is thus entered at the *front* of the current VM's process queue. If the current process is the null process, there is no state to save; a new process is simply selected if possible.

The first scheduling condition depends on the state of the current VM. This is stored in `stateCurrentVM` which can take on the values `stop` or `proceed`. The second condition depends on whether the process queue of the current VM is empty or not. This involves testing the length of a list: `LEN(currentVM.ProcQ) = 0`. The third condition can take on three values: `nullProc` if the current process is the null process, `kcall` if the kernel is currently responding to a kernel call or `int` if the kernel is servicing an interrupt.

The scheduler was designed by exhaustive case analysis. This technique can be used to design any decision mechanism that is based on a finite number of variables that can each take on a finite number of values. The following abbreviations are used in the

presentation:

$$
\begin{aligned}
STOP &\equiv \texttt{stateCurrentVM} = \texttt{stop} \\
PROCEED &\equiv \texttt{stateCurrentVM} = \texttt{proceed} \\
PQE &\equiv \texttt{LEN(currentVM.Procq)} = 0 \\
KCALL &\equiv \texttt{stateCurrentProc} = \texttt{kcall} \\
INT &\equiv \texttt{stateCurrentProc} = \texttt{int} \\
NULL &\equiv \texttt{currentProc} = \texttt{nullProc}
\end{aligned}
$$

A combination of two 2-valued conditions and one 3-valued condition leads to 12 cases. Each case is considered separately to specify an appropriate action according to the requirements of scheduling. This technique of exhaustive case analysis guarantees completeness of the specification with respect to the given set of conditions.

1. $STOP \wedge PQE \wedge KCALL \rightarrow$ Action 0

   A new VM must be selected. The current process is at a descheduling point (it is waiting for a kernel call). It is inserted in the (currently empty) process queue of the current VM. The current VM is then inserted at the end of the VM queue. A new VM is selected and the first process in its process queue is selected to be activated.

   (a) Insert current process at back of current VM's process queue

   (b) Insert current VM at back of VM queue

   (c) Select first VM in VM queue; assign to currentVM

   (d) Select first process in currentVM's process queue

2. $STOP \wedge PQE \wedge INT \rightarrow$ Action 1

   A new VM must be selected. When a process is interrupted, it is inserted at the front of the current VM's process queue to ensure that it will be resumed immediately when the VM is activated again. (Because the process queue is empty it makes no difference whether the process is inserted at the back or at the front of the queue.) The current VM can now be inserted at the end of the VM queue. A new VM is then selected and the first process in its process queue is selected to be activated.

   (a) Insert current process at front of current VM's process queue

(b) Insert current VM at back of VM queue

(c) Select first VM in VM queue; assign to currentVM

(d) Select first process in currentVM's process queue

3. $STOP \wedge PQE \wedge NULL \rightarrow$ Action 2

   The process queue is empty and there is no current process. The VM is not inserted in the VM queue, since it cannot be activated before at least one process joins its process queue again. A new VM and process are selected.

   (a) Select first VM in VM queue; assign to currentVM

   (b) Select first process in currentVM's process queue

4. $STOP \wedge \neg PQE \wedge KCALL \rightarrow$ Action 0

   The only difference between this case and case 1 is that the process queue is not empty. This is irrelevant and the current process (which is at a descheduling point, waiting for a kernel call) is simply inserted at the back of the current VM's process queue. The current VM is inserted at the back of the VM queue and a new VM and new process is selected. The action is the same as in case 1.

5. $STOP \wedge \neg PQE \wedge INT \rightarrow$ Action 1

   Another VM must be selected. The current process is inserted at the front of the process queue of the current VM. The action is the same as in case 2. The fact that the process queue is not empty, is irrelevant.

6. $STOP \wedge \neg PQE \wedge NULL \rightarrow$ Action 3

   A new VM must be selected, but since there is no current process, it is only necessary to return the current VM to the VM queue. A new VM is then selected and the first process in its process queue is selected.

   (a) Insert current VM at back of VM queue

   (b) Select first VM in VM queue; assign to currentVM

   (c) Select first process in currentVM's process queue

7. $PROCEED \wedge PQE \wedge KCALL \rightarrow$ Action 4

   The same VM must stay active. There is no other process (the process queue is empty) and the current process (which is waiting for a kernel call) is allowed to proceed, although it will actually just wait until the kernel call has been completed. The scheduler state remains unchanged.

8. $PROCEED \wedge PQE \wedge INT \rightarrow$ Action 4

   Action 4 is indicated. A change of VM is not required and the process queue is empty. The current process was interrupted and must proceed. The scheduler state remains unchanged.

9. $PROCEED \wedge PQE \wedge NULL \rightarrow$ Action 2

   It is not *required* that a new VM be selected. However, the current VM's process queue is empty. The current VM is thus entered in the VM queue and another VM selected. This is the same as Action 2.

10. $PROCEED \wedge \neg PQE \wedge KCALL \rightarrow$ Action 5

    This indicates a descheduling point. The current VM must proceed. Its process queue is not empty and a new process is selected after inserting the current process at the end of the process queue.

    (a) Insert current process at back of current VM's process queue

    (b) Select first process in currentVM's process queue

11. $PROCEED \wedge \neg PQE \wedge INT \rightarrow$ Action 4

    The current process was interrupted. A new process may not be selected and since a change of VM is not called for, the current process is allowed to proceed. This is the same as Action 4. The scheduler state remains unchanged.

12. $PROCEED \wedge \neg PQE \wedge NULL \rightarrow$ Action 6

    The current VM may stay active. However, there is no current process. Since the current VM's process queue is not empty, the first process is selected to run.

    (a) Select first process in currentVM's process queue

Note that the same action is sometimes specified under different circumstances. For example, action 0 is specified regardless of whether the process queue is empty or not. Simplification is possible by combining such conditions that lead to the same action. The first and fourth cases can be combined by forming their disjunction, since the same action (action 0) is indicated. The disjunction of the two conditions in this case is $(STOP \wedge PQE \wedge KCALL) \vee (STOP \wedge \neg PQE \wedge KCALL)$ which simplifies to $STOP \wedge KCALL$. Similar simplifications can be made for action 1, action 2 and action 4. After some manipulation, seven guarded actions remain as represented by the following IF-structure:

```
IF stateCurrentVM = stop & stateCurrentProc = kcall -> Action0
[] stateCurrentVM = stop & stateCurrentProc = int -> Action1
[] LEN(currentVM.ProcQ) = 0 & currentProc = nullProc -> Action2
[] stateCurrentVM = stop & LEN(currentVM.ProcQ) > 0 &
   currentProc = nullProc -> Action3
[] (currentProc # nullProc & stateCurrentProc = kcall &
   LEN(currentVM.ProcQ) = 0 & stateCurrentVM = continue)
   OR
   (currentProc # nullProc & stateCurrentProc = int &
   stateCurrentVM = continue) -> Action4
[] stateCurrentVM = continue & LEN(currentVM.ProcQ) > 0 &
   stateCurrentProc = kcall -> Action5
[] stateCurrentVM = continue & LEN(currentVM.ProcQ) > 0 &
   currentProc = nullProc -> Action6
END
```

The scheduler modifies its state by executing a number of basic queue operations. VMs are inserted at the back of the VM queue by the operation *InsertVM*. The first VM in the VM queue is selected by executing *SelectVM*. Similarly, processes can be selected by *SelectProc*. Processes are inserted at either the back or front of the current VM's process queue by *InsertProcB* and *InsertProcF* respectively. The basic operations to manipulate these queues are shown below:

InsertVM	APPEND(VMq, currentVM)
SelectVM	currentVM := HEAD(VMq); REMOVE(VMq)
SelectProc	currentProc := HEAD(currentVM.ProcQ); REMOVE(currentVM.ProcQ)
InsertProcB	APPEND(currentVM.ProcQ, currentProc)
InsertProcF	PREPEND(currentVM.ProcQ, currentProc)

The scheduler model contains seven unique guarded actions. The informal description of these actions that was given earlier can now be formalised as a sequence of basic operations on queues.

Action0	InsertProcB; InsertVM; SelectVM; SelectProc
Action1	InsertProcF; InsertVM; SelectVM; SelectProc
Action2	SelectVM; SelectProc
Action3	InsertVM; SelectVM; SelectProc
Action4	SKIP
Action5	InsertProcB; SelectProc
Action6	SelectProc

The list structure of ESML was inspired by the scheduler model. Although queues can be modelled as arrays, this is less clear and requires detail (index manipulation and modulo arithmetic) that obscures the design. The complete model of the scheduler is shown in Appendix A. The implementation code for the scheduler was derived directly from this model.

6.7 Correctness requirements

The first correctness property to check is freedom from deadlock. This will eliminate most interaction problems between processes. However, this is not enough. Even when it is known that deadlock is impossible, other more subtle errors may remain. Suitable temporal logic formulae must be used to express these more specific correctness claims. As suggested by Manna and Pnueli, most correctness claims that are important in practice can be expressed as either invariants, response properties, or precedence properties [51]. Unfortunately, this gives no indication of how to derive meaningful correctness claims. A systematic methodology is needed to increase confidence that a given design has been verified properly. In a way, the derivation of correctness claims for designs is similar to the derivation of test cases for implementations. A similar strategy, based on these techniques, is proposed here to derive correctness claims for reactive systems.

One of the standard references on testing methodology is the excellent book by Myers [57]. A more recent presentation, that specifically covers testing of reactive systems, can be found in [48]. These books provide detailed descriptions of various techniques that were found to yield best results in practice. There are basically two complementary approaches to test implementations. These are known as functional testing (black box testing) and structural testing (white box testing). During functional testing, the source code is ignored and test cases are derived from the specification. Various techniques known as equivalence partitioning, boundary-value analysis and cause-effect analysis have been shown to yield good test cases in practice. The disadvantage of functional testing is that errors caused by a specific implementation technique may be missed. For example, specific inputs may cause an error because some table is too small. It is hard to know this without considering the source code, which is the goal of structural testing. The basic idea is to examine the logic of a program and to derive test cases by considering the conditions in alternative commands and loops. These techniques should not be applied without keeping the specification in mind, because incomplete

implementations may then be missed.

Boundary value analysis is a technique that yields good correctness claims to check. The basic idea is to concentrate on extreme values of variables because experience shows that errors are more likely to occur near these boundary conditions. The basic assuption is that the value of only one variable is modified for each different correctness claim. It is useful to let each variable assume the minimum and maximum value of its range. In some cases, correctness claims can be derived by considering other interesting values, but this requires special knowledge about the design that is analysed.

ESML, the modelling language used here, simplifies boundary condition analysis because ranges of variables are checked automatically. Thinking about boundary conditions also helps to identify correctness claims that relate to startup conditions. It can be surprisingly difficult to get the initialisation of a model right. Thinking about boundary conditions helps to identify suitable correctness claims to be checked to verify startup conditions. The guiding principle for boundary condition analysis is to think of all variables as independent entities. Each variable has a specified range and the minimum and maximum values usually provide hints to formulate correctness claims. To keep the strategy simple, the interaction between variables is ignored. Boundary condition analysis is a source of simple invariance claims such as "the number of entries in the buffer will always be smaller than `max`". This can be expressed by a CTL formula of the form $AG(LEN(buf) < max)$

It is often unnecessary to consider the full range of values of a variable. A considerable amount of work can be saved by a technique known as equivalence class analysis. The ranges of variables are divided into a small number of disjoint subranges according to some criterion. The different values in each subrange are similar in some sense and are represented by a single abstract value. This strategy makes it possible to formulate more specific correctness claims because fewer different values are involved. As an example, consider a system that monitors the temperature of a liquid. When the temperature is between a given low and high mark, nothing happens. However, when the temperature rises above the high mark, a cooler must be activated and when the temperature drops below the low mark, a heater must be switched on. Instead of representing the temperature of the liquid by a variable that can take on values between 0 and 100 (say), it is much better to use only three values: low, medium and high. All values below the low mark form one equivalence class. Similarly, all values equal to or higher than the low mark and less than or equal to the high mark form a second

equivalence class, and the values higher than the high mark form a third equivalence class. Because there are seldom more than a few equivalence classes for each variable, it is feasible to formulate correctness claims to check all possible combinations of values. First it should be checked that each possible value can occur. This is often stated negatively as a simple CTL formula of the form $AG(t \# low)$, where t represents the temperature in the example given above. When this claim is violated a designer can be certain that the model can reach a state where the temperature is low. Whether the right action occurs at each temperature can then be checked by verifying a claim expressed as $AG((t = low) => AF(lowact))$, where *lowact* is the appropriate action for a low temperature. Some errors occur because more than one variable take on extreme values simultaneously. A strategy to study this kind of behaviour is called worst case analysis, and is useful to study extreme conditions.

Another successful technique, known as cause-effect analysis, involves the interaction between events. The guiding principle is to identify all events relevant to a given design. These events are then classified as "causes" and "effects" and paired off to consider every possible combination. Decision tables may be used to specify these pairs of events. Conditions should be thought of as "inputs" and effects as "outputs" and these input-output pairs then form the basis of a valuable class of correctness claims. Sometimes more than one condition must be met to cause a certain effect. A systematic approach will ensure that important event sequences are not missed.

Just like structural testing can yield test cases to check specific mechanisms, the logic of a model can be analysed to obtain valuable correctness claims. It should be checked that each alternative of every IF command can be executed. Similarly, each guard in every loop (DO or POLL) should be checked to be satisfiable. The simplest of CTL formulae are needed to check for unreachable fragments in a model, but subtle errors can sometimes be pinned down in this way. If tables are used, check that each entry is used at least once. Also check that every kind of message is actually used and that every message sent is received. Auxiliary variables are sometimes needed to capture the fact that some condition holds or that some event has occurred. Finally, express invariants to be checked for every loop, every process, and the system as a whole.

The following strategy is proposed for designing correctness claims for the verification of reactive systems.

1. Since designs for reactive systems concentrate on flow of control, it makes sense to start by identifying causes and effects. Identify all events that are considered

either a cause or an effect and consider all possible combinations and write temporal logic formulae that can be used to check these claims. Consider both valid and invalid cases. This step should result in a list of response properties to be checked. Look for events in processes that are triggered from other processes. This ensures that process scheduling is involved.

2. Use boundary-value analysis to derive additional correctness claims to check properties not already covered. Specifically check what happens when buffers are empty or full. Also, use equivalence class techniques to limit the values that control variables can take on. Equivalence class techniques make it possible to restrict the number of values that variables can take on. This often makes it feasible to write correctness claims that will explore all possible combinations of these values.

3. Examine the logic of each process and derive claims to check every condition in alternative commands and the terminating conditions of every loop. Include claims to check all possible messages in every channel alphabet. Write formulae to check every entry in every table.

4. Derive meaningful invariants for every loop and process. Invariants can also be defined for some data structures. For example, it may be required that the entries in a queue be ordered. This is expensive to check, but may be feasible in some cases.

5. Study the interaction of different processes and define system invariants that should be maintained.

Of course the proposed strategy does not guarantee that a set of correctness claims will be derived that is complete in any formal sense. However, such a systematic approach must be more reliable than ad hoc techniques. In particular, formal verification offers the following advantages when compared to traditional testing:

- Testing is carried out by using the implementation. A model (prototype) is used for formal verification. Errors are therefore discovered at an earlier stage.

- Coverage is incomplete for testing because it is impractical to check all execution sequences. For the same reason traditional debuggers are also rather limited as a means of finding errors. In formal verification the complete relevant state space

is checked for every correctness claim. A test case shows that an implementation is correct for a single specific execution sequence. A correctness claim shows that a design is correct for all possible execution sequences.

- It can be checked that invariants are maintained. Without formal verification the best that can be done is to include certain checks in the code to check invariants at run time. This is a source of inefficiency that can only be afforded for the simplest of cases such as checking array indices. Formal verification can sometimes be used to check for and eliminate errors such as buffer overflow. In some situations it then becomes unnecessary to use run-time checks in implementations.

Despite the advantages of formal verification the list of correctness claims may still be incomplete and therefore a design can only be claimed to be correct with respect to a given set of correctness claims.

Validated models are developed interactively. Temporal logic is used as a kind of "query language" to increase a designer's confidence in the correctness of a model. In this way, correctness claims are formulated, verified and discarded all the time. The final list of verified correctness claims therefore represent only a small subset of all the temporal logic formulae that were used to gradually convince a designer that a given model is correct.

To illustrate this development technique, a number of correctness claims that were used to verify the Gneiss kernel models will be described here. The correctness claims verified for the global kernel model (shown in Appendix A) will be discussed first. After checking for deadlock to eliminate general interaction problems, every process was studied on its own. For example, process `user` contains two interesting events: when a resume signal is received, a kernel call is executed in response. By studying each process on its own and following the control logic of a model, interesting execution sequences can be identified and checked by a verification system. A systematic approach will ensure that all interesting cause-effect sequences are identified. To increase the chances of showing up errors, it helps to study execution paths of reasonable length to include as many intermediate actions as possible. This shows a fundamental difference between testing and formal verification. Modules are tested one at a time during structural and functional testing to keep execution sequences short. That is the only way to force execution of specific paths. However, with formal verification it is unnecessary to force the execution of specific paths. All paths relevant to a given correctness claim are executed automatically. If an error can occur, it will be detected.

The reception of a resume signal by process `user` should be seen as an "effect" that is "caused" by some event in the rest of the model. Process `running` sends the resume signal after requesting a new process from the scheduler. A cause-effect relationship therefore exists between the processes `user` and `running`. As another example, consider the event `!kcall(id)` in process `user`. By following the logic it can be seen that this event should cause the event `!doio` in running. In turn, this should eventually cause the device to start an io operation, as indicated by setting the flag deviceidle to false. This sequence of events can be checked by the CTL formula $AG(Blocked(p_i) => AF\neg(Idle(device))$. Predicates like *Blocked* depend on the values of variables in a model. Sometimes auxiliary variables must be used because the information needed is not recorded explicitly by the model. For example, a variable `blocked` was added to record whether process `user` is blocked or not. Auxiliary variables may be added provided that they do not influence the control logic of a model in any way.

Other cause-effect sequences could be identified by analysing the other processes in the model one by one. The following formulae capture interesting correctness claims for the global kernel model:

- $AG(Entered(p_i) => AF(Selected(p_i))$, which means that if process p_i is entered in the scheduler queue, process p_i will eventually be selected and activated again.
- $AG(Waiting(p_i)$ & $Signal => AF(Entered(p_i))$, which means that if process p_i is waiting for IO and an IO completion signal arrives, process p_i will eventually be entered in the scheduler queue.

The only two interesting data structures in the model are the scheduler queue in process `ready` and the request queue in `devicedriver`. Boundary value techniques suggest that empty queues should be investigated. The following correctness claims could be verified:

- $AG(Idle(device) => Empty(rq))$, which means that when the peripheral device is idle the request queue in `devicedriver` must be empty.
- $AG(Empty(q) => EF(\neg Idle(device))$, which means that the peripheral device can be busy processing an IO request while the scheduler queue is empty. This is possible when there is only one user process in the system.

The model `GComms` represents interprocess communication in more detail. Because the three different kernel calls supported by the Gneiss kernel were modelled explicitly, it was possible to distinguish between client and server processes. The processes `Client`, `Server`, `Ready`, and `Running` are simple. Similar processes have been encountered in the global kernel model. The process `Comms` is the interesting one to concentrate on in this model. A POLL command is used to select any one of the three kernel calls. If process p_i executes a transaction and process p_j responds to it by executing a `ReceiveRequest` operation, the kernel must keep track of this. An array `Partner[]` is used to keep track of communication partners. When a server process executes a `SendReply` operation, this array is used to determine which processes to activate.

The proposed technique of cause-effect analysis was used to derive the following correctness claims:

- Client and server processes can be paired off as communication partners. However, clients cannot communicate with clients and servers cannot communicate with servers. Several CTL formulae of the form $AG(Comms.Partner[m] \# n)$ were used to check this.
- When several client processes are sending requests to a server by executing `Transaction` operations, it should be possible to have more than one process waiting in the request queue. A server process and three clients were used to check this. The formula $AG(LEN(Comms.pq < 2))$ is violated when a second process is entered in the request queue.
- $AG(Trans => AF(RecReq))$ When a client process executes a `Transaction` kernel call the server will eventually execute a `ReceiveRequest` kernel call. A single client and server were used to check this.

The ESML model of the Gneiss scheduler contains more detail than any of the other kernel models. However, the structure of the model is simple. It consists mainly of a rather complex IF command, each alternative guarding execution of a single action. The actions manipulate the scheduler queue by removing and entering processes or VMs. As was explained in Section 6.6, the conditions were derived in a systematic way by using exhaustive case analysis and the model was therefore expected to be correct. The structure of the model suggested cause-effect analysis, which was extremely simple in this case. Each condition in the IF command represented a different cause, the corresponding effect being the action taken. A set of simple CTL formulae of the form

$AG(Ready.action \# n)$ quickly established that all actions could be executed, except action number five. The error was tracked down to an incorrectly coded condition in the controlling IF structure. Formal analysis of a scheduler is particularly important because errors sometimes do not cause the system to stop, but may lead to inefficiency.

Many of the correctness claims described above were violated at first. This will not be surprising to anyone who has used formal verification in practice. Deadlock is often encountered during the initial stages of developing a model. In fact, considerable patience is usually required to make the initial deadlocks disappear. Of course, a deadlock free model does not mean that everything is fine. Surprisingly subtle errors can sometimes be found by checking invariants that should hold or by checking specific patterns of interaction. Experience shows that the most valuable properties to check are simple. This is good news, because complex properties are also the most expensive ones to check. For example, formulae that contain nested temporal operators require much more time than simple assertions and invariant formulae. These simpler properties should therefore be checked first because when an error is detected the model must be modified and all previous verification runs must be repeated.

6.8 Summary

The verification of an entire operating system is a formidable task. A reasonable first step involves the verification of a single interesting component. This was shown to be possible. A detailed formal verification study of a microkernel as described in this chapter has not been published before. Holzmann has published similar studies for protocols [44, 32], but operating systems present a different challenge. The approach taken was to divide the design into components that could be verified independently. This was done by structuring the kernel as a set of interacting servers. Some mechanisms such as buffers and schedulers are fundamental building blocks. It was shown how such components could be modelled in ESML, the modelling language described in Chapter 4. Such verified designs may be reusable in different application areas.

Experience has shown that certain constructs in a modelling language can simplify models significantly. For example, the list construct of ESML was particularly useful to model queues. Without it the rather complex scheduler model would contain even more distracting detail. Other language constructs such as dynamic process creation and buffered communication channels, were not needed at all. This indicates that

different kinds of modelling languages may be required for different application areas.

It was shown how the implementation of the Gneiss kernel corresponds to the verified design. The microkernel is efficient, despite being structured as a set of interacting servers. It is significant that this was not merely an academic project. In fact, the Gneiss kernel is used to support a commercial digital signal processing system.

Whether the same verification techniques will also work for other operating system components remains to be seen. However, it seems likely that most servers that incorporate concurrent algorithms will benefit from the verification techniques described here.

Chapter 7

Evaluation and Conclusions

Probably the most significant contribution of this work is its demonstration that automated verification techniques can be applied to the development of operating system software. The project showed that:

- Validated designs can be developed for typical mechanisms that are used in microkernels. The design of the *Gneiss* microkernel was verified against various correctness criteria. The level of detail captured by the various models differ, but in two cases—the scheduler and interprocess communication—the models and implementations correspond closely. In such cases, it should be possible to generate the implementation code automatically from the verified design, although this was not attempted. The *Gneiss* microkernel is a useful tool. It is used in two commercial applications: a distributed control system for assembly lines in factories and a dedicated digital signal processing system. Although *Gneiss* is a relatively small microkernel, it contains many mechanisms found in well-known microkernels such as Amoeba, Mach and V. Whether some of these verified designs are fundamental in some sense remains to be seen.

- An efficient microkernel can be designed as a collection of interacting servers, each simple enough to be verified in considerable detail. Suitable structuring principles are essential, however. Servers must conform to definite rules to allow them to be verified separately and combined into a correctly functioning kernel. Without this structuring technique it would probably be difficult to verify a microkernel in sufficient detail. A top-down approach was used. A high-level model was first designed to study the interaction between the major components

of the kernel. More detailed models were then constructed of the scheduler, interprocess communication and device drivers. The Gneiss device drivers all share the same object-oriented design: a more complex, but reusable part that interacts with the rest of the kernel, and a simpler device-specific part [56]. The model of a disk driver described in Chapter 6 reflects this structure. The remote communication mechanism used in Gneiss is a standard protocol that was verified using a similar approach [63].

- A modelling language with special features is needed to verify a microkernel. The list construct of ESML was especially useful when modelling the scheduler and device drivers. However, complex structured data lead to larger states and an effective state compression technique was developed to cope with this problem.

The experimental verifier described in Chapter 5 was used to verify the kernel models. This verifier is based on standard on-the-fly verification techniques. However, a new technique was developed to handle nested subformulae. This technique, which involves delayed evaluation of subformulae was preferred, although there are different on-the-fly algorithms that can handle full CTL [72, 4]. The reason is that these general algorithms require too much memory in practice.

The verifier was divided into three components, each with a well defined function. These are the state generator, state storage and the evaluator. This modular structure makes it easier to experiment by replacing some components with alternative ones. Despite this modular structure, the resulting verifier was still quite efficient. States are analysed at roughly the same rate as SPIN [44], which is widely regarded as an efficient verification system. It should be kept in mind, however, that a detailed comparison with SPIN was not attempted because there are too many differences to make this meaningful. Efficient compaction techniques were found to be essential. Even though some processor time is spent on compacting states before they are stored in the cache, this is amply repaid because the hash functions used to access the cache are simpler to compute when states are smaller.

The most serious limitation of state based verification techniques is the huge number of states generated. Much progress has been made and current verifiers can cope with models that generate millions of states. Guidelines are crucial to avoid a state explosion. The guidelines given in Chapter 6 are intended to simplify the development of verified models of operating system software. By following these guidelines, the number of states generated were found to be reduced significantly.

Although models of the most important mechanisms in the Gneiss microkernel were developed and verified, some problems remain. Currently only relatively simple correctness properties of finite state models can be verified efficiently. Although more expressive logics exist that can capture more advanced correctness properties, efficient verification algorithms are known only for simple logics like CTL and LTL. To verify more elaborate correctness claims, a combination of model checking and theorem proving may be useful.

A problem with the approach taken in verifying the Gneiss microkernel is that there is no rigorous connection between models and implementations. Since models are abstract programs, it seems feasible to generate efficient implementations automatically from verified models. This has been done for protocols. For microkernels, some mechanisms are too low level and machine dependent and require careful coding for acceptable performance. At best, the verified models can serve as guidelines for coding. A strategy that worked well, was to include various correctness checks in the implementation code to detect errors at run-time. Validated models can often suggest such run-time checks.

The strict adherence to CSP as a design framework worked out well. For example, global variables shared among processes were never missed. However, although ESML served its purpose well, some concepts should be reconsidered. Among these are the use of a single construct—a process—for expressing both concurrency and encapsulation, which are two different concepts. Two different constructs such as a module for encapsulation and a process for concurrency would probably be better. Also, a complex model that contains a large number of channels is difficult to understand. Bidirectional channels may be used to reduce the number of channels. In addition, the POLL construct should be simplified to allow a more efficient implementation like the strategy used in SPIN.

A fundamental problem is that a verifier, like any complex program, may contain subtle errors. When a model is rejected erroneously, an "error trail" will be produced that should be enough to detect the problem. On the other hand, a model that contains an error may be accepted. This is more serious because such errors may go undetected. Fortunately, verification systems are based on sound theory. Formal logic and automata theory present two such theoretical frameworks that can guide the design of verification systems. However, just how to translate sound theoretical ideas into functional verification systems that use computing resources effectively, is quite another matter. The design of the verifier described in Chapter 5 is meant to improve reliability by

dividing a complex program into simpler interacting modules. The verifier has three components: a state generator, a separate component to evaluate logical formulae, and a compiler for the modelling language. Each component has a well-defined interface and as long as this is preserved, internal mechanisms can be modified without affecting other components. This design and the choice of a safe implementation language (Oberon) should improve reliability.

7.1 Summary of what was learnt from the project

The experience gained during this project is summarised here because it may be useful to others.

7.1.1 Verifiers

- LTL seems preferable to CTL for expressing correctness claims in practice although this is a controversial issue[49, 59, 25, 27]. Although CTL can be checked in linear time and the time complexity of checking LTL is exponential, it should be remembered that the exponential blow-up for LTL is caused by the number of operators in the formula, which is usually small in practice. More important is that large data structures are needed to store intermediate results in a CTL model checker. The CTL model checking algorithm of Vergauwen and Lewi needs a large array for storing the values of every subformula in every state [72]. The algorithm by Bhat, Cleaveland and Grumberg for CTL* (which includes CTL) has similar space requirements [4]. Two large data structures are needed to store all assertions that have been encountered and a set of assertions that have been found to be false. In the verifier described in this dissertation (Chapter 5), the memory requirements could be reduced by handling only a subset of CTL. In addition, it is hard to find a really useful formula that can be expressed in CTL, but not in LTL. However, there are useful properties that can be expressed in LTL, but not in CTL. For example, it is possible to express strong fairness in LTL, but not in CTL. A classification of useful LTL properties to be checked for concurrent systems is described in [52].

- A model checker should be designed to check directly for the simplest properties. For example, to check for deadlock freedom only requires a single full search of the reachable state space. This can be done more efficiently if no temporal logic

formula is specified. It is simply necessary to check for invalid end states. This approach is also used in the SPIN verifier [44]. Such simple correctness properties seem to be the most useful to check in practice and should therefore be handled as efficiently as possible.

- Within reason, model checking algorithms should be optimised for space, not time. Reachable states must be stored in main memory, since secondary storage is too slow to be practical. One way to fit more states into main memory is to use compaction techniques. While the simple compaction technique described in Section 5.2 can actually speed up state comparison operations, the amount of compaction obtained in this way is limited (about 40%). More effective compaction techniques seem to slow down the model checker as was shown in [45]. However, since processor speeds are increasing all the time, more sophisticated state compaction techniques may be feasible in the near future.

- Computing strongly connected components is probably not a good idea for implementing fairness. It was found to double the memory requirements in some cases, although the effect on run time is negligible. The effect of computing strongly connected components in the ESML model checker is reported in [34].

- A model checker can be structured as two components: a state generator and a state analyser. The ESML model checker is structured in this way as described in Section 5.5. The state analyser requires only the values of propositions in the current state. The details of how states are represented are hidden inside the state generator. This approach makes the model checker easier to understand.

- Using interpretation to generate states simplifies the model checker. This was found to have little measurable effect on the execution speed as reported in [34]. The reason is that the overhead of interpretation is small compared to the other tasks to be performed by a model checker.

7.1.2 Verification of operating systems

- The most important contribution of this project is perhaps the illustration that parts of an operating system can be verified. In Chapter 6 three different models of parts of a microkernel are discussed: a model of a scheduler, disk driver, and inter-process communication. A more abstract model is also presented to

study the main flow of control through the system as it interacts with user-level processes. Two different approaches are recommended: (1) Use modelling as a design technique while developing intricate mechanisms. The completed verified model can then be used as a guideline for developing an implementation. (2) Derive models from existing implementation code. This is only feasible for well-structured code. If an error is detected in a model, it should be possible to determine whether a similar error could occur in the implementation. This is a valuable check on the correctness of intricate mechanisms, especially when concurrency is involved.

- Do not try to model the complete system because the state space may be too large to include enough detail to find subtle errors. As described in Chapter 6 the Gneiss kernel was modelled as interacting client and server processes. In this way it is possible to replace a model of a server (for example, the scheduler) by an abstract version when its internal details are irrelevant. This technique is illustrated by the "global interaction model" shown in the appendix (A.1).

- It helps to focus on the most intricate mechanisms where concurrency is involved. For example, the model of a disk driver described in Chapter 6 was designed to study the asynchronous nature of code that drives peripheral devices. The disk driver can receive requests from user processes even when the disk controller is busy. For this a queue of requests is maintained and interrupts signal the end of each IO operation. Even rather abstract models of such mechanisms are useful. Logical errors are difficult to find during testing and these are exactly the kind of errors that can be found through verification.

- For operating systems it is probably not feasible to generate implementation code from verified models. A good programmer can certainly produce more efficient code than can be generated automatically. At best, it may be possible to translate certain verified mechanisms like a scheduler into efficient code. Although this was not attempted, the process of translating the scheduler model described in Section 6.6 into implementation code was rather mechanical.

- A serious attempt to find suitable abstractions for low-level operations that can be represented as atomic transitions in a model can improve the implementation significantly.

- When the implementation is modified as the system evolves, always remember to modify the models too and repeat the verification runs. It is all too easy to

introduce subtle bugs by seemingly trivial modifications to the implementation code.

Appendix A

The Gneiss kernel models

A.1 Global interaction model

```
MODEL Kernel;

  CONST
    nullproc = 0; qmax = 2;

  TYPE
    procid = 0..2;
    readyrequest = {selectproc(procid), enterproc(procid)};
    runningrequest = {newproc(procid), kcall(procid), int, tick};
    iocommand = {doio(procid), iocomplete};
    devicecommand = {startio};
    continue = {resume(procid)};
    procqueue = LIST[qmax] OF procid;

  VAR ch0: readyrequest;
    ch1: runningrequest;
    ch2: continue;
    ch3: iocommand;
    ch4: devicecommand;

  PROCESS user(id: procid; OUT torunning: runningrequest;
```

```
    OUT toready: readyrequest;
    IN request: continue);
    (* use value parameter to identify different processes (procid) *)
  VAR p: procid;
  BEGIN
    (* proc registers itself *)
    toready ! enterproc(id);
    DO TRUE ->
      (* wait for the right resume message (procid 0 or 1) *)
      POLL request?resume(p) & p = id -> SKIP END;
      torunning!kcall(id)
    END
  END user;

  PROCESS ready(IN request: readyrequest;
    OUT response: runningrequest);
  VAR p: procid; q: procqueue;
  BEGIN
    request?enterproc(p); APPEND(q, p); (* initialize ready queue *)
    DO TRUE ->
      POLL
         request ? selectproc(p) ->
           IF LEN(q) > 0 ->
             p := HEAD(q); REMOVE(q); response!newproc(p)
           [] LEN(q) = 0 ->
             response!newproc(nullproc)
           END
      [] request?enterproc(p) ->
         IF p # nullproc -> APPEND(q, p)
         [] p = nullproc -> SKIP
         END
      END
    END
  END ready;

  PROCESS running(OUT toready: readyrequest;
```

```
  IN request: runningrequest;
  OUT todevicedriver: iocommand; OUT touser: continue);
VAR curproc: procid;
BEGIN
  DO TRUE ->
    POLL toready!selectproc(curproc) ->
      request?newproc(curproc);
      IF curproc # nullproc -> touser ! resume(curproc)
      [] curproc = nullproc -> SKIP
      END
    [] request?kcall(curproc) -> todevicedriver!doio(curproc)
    [] request?int ->
      todevicedriver!iocomplete
    END
  END
END running;

PROCESS devicedriver(OUT toready: readyrequest;
  IN request: iocommand; OUT todevice: devicecommand);
  VAR curproc, id: procid; rq: procqueue; 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 ->
        toready!enterproc(curproc);
        IF LEN(rq) = 0 -> deviceidle := TRUE
        [] LEN(rq) > 0 ->
            curproc := HEAD(rq);
            REMOVE(rq);
            todevice!startio
```

```
          END
      END
    END
  END devicedriver;

  PROCESS device(OUT torunning: runningrequest;
    IN request: devicecommand);
    VAR idle: BOOLEAN;
  BEGIN
    idle := TRUE;
    DO TRUE ->
      POLL request?startio & idle ->
        idle := FALSE
      END;
      idle := TRUE;
      torunning!int
    END
  END device;

BEGIN
  ready(ch0, ch1); running(ch0, ch1, ch3, ch2);
  devicedriver(ch0, ch3, ch4); device(ch1, ch4);
  user(1, ch1, ch0, ch2); user(2, ch1, ch0, ch2)
END Kernel;
```

A.2 Interprocess Communication

```
MODEL GComms;
(* Gneiss microkernel: interprocess communication *)

  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(HEAD(q)) & LEN(q) > 0 ->  REMOVE(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 := HEAD(pq); REMOVE(pq);
                Partner[cp] := sp;
                Partner[sp] := cp;
                toReady!enterProc(sp)
            END
        [] in?RecReq(sp) ->
            IF clients & LEN(pq) > 0 -> (* a client is ready *)
                cp := HEAD(pq); REMOVE(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 GComms;
```

A.3 Device drivers

```
MODEL GDisk;

  CONST
```

```
  nullproc = 0; qmax = 2;

TYPE
  ProcID = 0..2;
  ProcQueue = LIST[qmax] OF ProcID;
  ContrReq = {IO};
  DiskReq = {IOReq(ProcID), Interrupt};
  ClientReq = {EnterProc(ProcID)};
  ST506Req = {Start, Transfer};

VAR ch0: DiskReq;
  ch1: ST506Req;
  ch2: ContrReq;
  ch3: DiskReq;
  ch4: ClientReq;

PROCESS Client(IN req: ClientReq; OUT toDisk: DiskReq);
VAR rq: ProcQueue; id: ProcID; next: ProcID;
BEGIN
  EMPTY(rq); APPEND(rq, 1); APPEND(rq, 2);
  next := HEAD(rq); REMOVE(rq);
  DO TRUE ->
    (*
    POLL
      toDisk!IOReq(HEAD(rq)) & (LEN(rq) > 0) -> REMOVE(rq)
    [] req?EnterProc(id) -> APPEND(rq, id)
    END;

    next := HEAD(rq); REMOVE(rq);
    toDisk!IOReq(next);
    req?EnterProc(id);
    APPEND(rq, id);
    *)
    toDisk!IOReq(1);
    req?EnterProc(id)
  END
```

```
END Client;

PROCESS Disk(IN req: DiskReq; OUT toST506:ST506Req;
  OUT toClient: ClientReq);
VAR activeproc, currentproc: ProcID;
  rq: ProcQueue; (* queue of requests *)
BEGIN
  EMPTY(rq);  activeproc := nullproc;
  DO TRUE ->
    POLL req?IOReq(currentproc) ->
      IF activeproc = nullproc -> (* idle, start operation *)
        activeproc := currentproc;
        toST506!Start
      [] activeproc # nullproc -> (* busy, queue operation *)
        APPEND(rq, currentproc)
      END
    [] req ? Interrupt ->
      toST506!Transfer;
      (*Wakeup:*)
      toClient!EnterProc(activeproc);
      IF LEN(rq) # 0 -> (* handle next request *)
        activeproc := HEAD(rq); REMOVE(rq);
        toST506!Start
      [] LEN(rq) = 0 -> (* no more requests queued *)
        activeproc := nullproc
      END
    END
  END
END Disk;

PROCESS ST506(IN req: ST506Req; OUT toContr: ContrReq);
BEGIN
  DO TRUE ->
    POLL req ? Start -> toContr ! IO
    [] req?Transfer -> SKIP
    END
```

```
    END
  END ST506;

  PROCESS Contr(IN req: ContrReq; OUT toDisk: DiskReq);
  BEGIN
    DO TRUE ->
      req?IO;
      toDisk!Interrupt
     END
  END Contr;

BEGIN
  Client(ch4,ch0);
  Disk(ch0,ch1,ch4); ST506(ch1,ch2); Contr(ch2,ch0)
END GDisk;
```

A.4 Scheduler

```
MODEL GScheduler;

CONST
  NrVMs = 2; NrProcs = 3; NIL = NrProcs; (* NIL is highest proc nr *)
TYPE
  INT = 0..NrProcs;
  ProcState = active, int, kcall;
  ProcInfo = (id: INT; state: ProcState);
  A0 = {selectproc(ProcInfo), enterproc(INT)};
  A1 = {newproc(INT)};
  A2 = {doio(INT)};
VAR ch0: A0; ch1: A1; ch2 : A2;

PROCESS Ready(IN in : A0; OUT response : A1);
TYPE
  VMQ = LIST[NrVMs]  OF INT;
  ProcQ = LIST[NrProcs]  OF INT;
```

```
  ProcQarray = ARRAY[NrVMs] OF ProcQ; (* proc queues *)
  ProcToVM = ARRAY[NrProcs] OF INT; (* maps procs to vms *)
  Action = 0..6;
VAR cp: ProcInfo; p: INT; cvm: INT; action: Action;
  stop: BOOLEAN; q: ProcQarray; vmq: VMQ; vm: ProcToVM;
BEGIN (* initialise *)
  EMPTY(q[0]); APPEND(q[0], 0);
  EMPTY(q[1]); APPEND(q[1], 1); APPEND(q[1], 2);
  vm[0] := 0; (* proc 0 belongs to vm 0 *)
  vm[1] := 1; (* proc 1 belongs to vm 1 *)
  vm[2] := 1; (* proc 2 belongs to vm 1 *)
  EMPTY(vmq); APPEND(vmq, 1);
  cvm := 0;
  DO TRUE ->
    POLL
      in?selectproc(cp) ->
      (* choose whether vm will stop or not *)
      IF TRUE -> stop := TRUE [] TRUE -> stop := FALSE END;
      (* now select a new process *)
      IF stop & cp.id # NIL & cp.state = kcall ->
        (* action 0: *) action := 0;
        APPEND(q[cvm], cp.id); APPEND(vmq, cvm);
        cvm := HEAD(vmq); REMOVE(vmq);
        p := HEAD(q[cvm]); REMOVE(q[cvm])
      [] stop & cp.id # NIL & cp.state = int ->
        (* action 1: *) action := 1;
        PREPEND(q[cvm], cp.id); APPEND(vmq, cvm);
        cvm := HEAD(vmq); REMOVE(vmq);
        p := HEAD(q[cvm]); REMOVE(q[cvm])
      [] (cvm = NIL OR LEN(q[cvm]) = 0) & cp.id = NIL ->
        (* action 2: *) action := 2;
        cvm := HEAD(vmq); REMOVE(vmq);
        p := HEAD(q[cvm]); REMOVE(q[cvm])
      [] stop & cvm # NIL & LEN(q[cvm]) > 0 & cp.id = NIL ->
        (* action 3: *) action := 3;
        APPEND(vmq, cvm); cvm := HEAD(vmq); REMOVE(vmq);
```

```
        p := HEAD(q[cvm]); REMOVE(q[cvm])
      [] (cvm # NIL & LEN(q[cvm]) = 0 &
          cp.id # NIL & cp.state = kcall) OR
        (cp.id # NIL & cp.state = int & ~stop) ->
        (* action 4: *) action := 4; p := cp.id
      [] ~stop & cvm # NIL & LEN(q[cvm]) > 0 & cp.id # NIL &
        cp.state = kcall ->
        (* action 5: *) action := 5;
        APPEND(q[cvm], cp.id);
        p := HEAD(q[cvm]); REMOVE(q[cvm])
      [] ~stop & cvm # NIL & LEN(q[cvm]) > 0 & cp.id = NIL ->
        (* action 6: *) action := 6;
        p := HEAD(q[cvm]); REMOVE(q[cvm])
      END; response!newproc(p)

    [] in?enterproc(p) -> APPEND(q[vm[p]], p)
    END
  END
END Ready;

PROCESS Running(OUT toReady: A0; IN in: A1;
  OUT toDeviceDriver: A2);
VAR p: INT; cp: ProcInfo;
BEGIN cp.id := NIL;
  DO TRUE ->
    toReady!selectproc(cp); in?newproc(p);
    cp.id := p;
    IF TRUE -> cp.state := kcall;
       toDeviceDriver!doio(cp.id); cp.id := NIL
    [] TRUE -> cp.state := kcall (* non-io kcall *)
    [] TRUE -> cp.state := int (* process was interrupted *)
    END
  END
END Running;

PROCESS DeviceDriver(IN in: A2; OUT toReady: A0);
```

```
VAR p: INT;
BEGIN
  p := NIL;
  DO TRUE -> in?doio(p); toReady!enterproc(p) END
END DeviceDriver;

BEGIN
  Ready(ch0, ch1); Running(ch0, ch1, ch2);
  DeviceDriver(ch2, ch0)
END GScheduler;
```

A.5 The global interaction model in Promela

```
#define nullproc 0
#define qmax     3
#define FALSE    0
#define TRUE     1

#define Append(q,qlength,p) \
  d_step { q[qlength] = p; qlength = qlength+1; }
#define HEAD(q) q[0]
#define REMOVE(q,qlength) \
  d_step { \
   i = 0; \
   do \
   :: i <= qlength-1 -> \
      q[i] = q[i+1]; \
      i++ \
   :: i > qlength-1 -> break \
   od;\
   qlength = qlength-1 \
  }
#define LENGTH(qlength) qlength

mtype = {createproc, selectproc, enterproc, newproc, kcall, inter,
```

```
        tick, startio, resume, iocomplete, doio}

chan readyrequest          = [0] of {byte,byte}
chan runningrequest        = [0] of {byte,byte}
chan continue_chan         = [0] of {byte,byte}
chan iocommand             = [0] of {byte,byte};
chan runningtodevicedriver[2]        = [0] of {byte,byte};
chan device_iocommand      = [0] of {byte,byte};

hidden byte dummy;

#define p1 (the_proc==1)
#define q1 (the_proc==2)

proctype ready(chan request, response)
{
 byte p,i;
 byte q[qmax];
 byte qlength=0;

 do :: TRUE ->
   if
   :: request?selectproc(p) ->
      if
      :: qlength > 0 ->  p = HEAD(q); REMOVE(q,qlength);
                         response!newproc(p);
      :: qlength == 0 -> response!newproc(nullproc)
      fi
   :: request?enterproc(p) ->
      if
      :: p != nullproc -> Append(q,qlength,p);
         assert(qlength < qmax);
      :: p == nullproc -> skip
      fi
   fi
 od
```

```
}

proctype running(chan toready, request, todevicedriver, touser)
{
 byte curproc;

 do :: TRUE ->
   if
   :: toready!selectproc(curproc) ->
      request?newproc(curproc);
      if
      :: curproc != nullproc -> touser!resume(curproc);
      :: curproc == nullproc -> /* skip */
      fi;
   :: request?kcall(curproc) ->
      runningtodevicedriver[0]!doio(curproc)
   :: request?inter(dummy)   ->
      todevicedriver!iocomplete(dummy)
   fi
 od
}

proctype devicedriver(chan toready, request, todevice)
{
 byte curproc,id,i;
 byte rq[qmax];
 byte rq_length=0;
 bool device_idle=TRUE;

 do :: TRUE ->
    if
    :: runningtodevicedriver[1-device_idle]?doio(id) ->
       curproc = id;
       todevice!startio(dummy);
       device_idle = FALSE
    :: runningtodevicedriver[device_idle]?doio(id) ->
```

```
        Append(rq,rq_length,id)
    :: request?iocomplete(dummy) ->
        toready!enterproc(curproc)
    if
    :: rq_length == 0 -> device_idle = TRUE
    :: rq_length > 0  -> curproc = HEAD(rq);
            REMOVE(rq,rq_length);
            todevice!startio(dummy)
    fi
    fi
 od
}

proctype device(chan torunning,
         request)
{
  do :: TRUE ->
     request?startio(dummy);
     torunning!inter(dummy)
  od
}

proctype user(byte id; chan torunning, toready, request)
{
  toready!enterproc(id);
  do :: TRUE ->
    if ::(id == 1) -> request?resume,1
       ::(id == 2) -> request?resume,2
       ::(id == 3) -> request?resume,3
    fi;
    torunning!kcall(id)
  od
}

init {
 atomic {
```

```
    run ready(readyrequest,runningrequest);
    run running(readyrequest,runningrequest,iocommand,continue_chan);
    run devicedriver(readyrequest,iocommand,device_iocommand);
    run device(runningrequest,device_iocommand);
    run user(1,runningrequest,readyrequest,continue_chan);
    run user(2,runningrequest,readyrequest,continue_chan)
    }
}
```

Bibliography

[1] A. Aho, J. Hopcroft, and D. Ullman. *The Design and Analysis of Computer Algorithms*. Addison-Wesley, 1974.

[2] G. R. Andrews. *Concurrent Programming*. The Benjamin/Cummings Publishing Company, Inc., 1991.

[3] A. Arnold. *Finite Transition Systems*. International Series in Computer Science. Prentice Hall, 1994.

[4] G. Bhat, R. Cleaveland, and O. Grumberg. Efficient On-the-Fly Model Checking for CTL*. In *Proceedings 10th Symposium on Logic in Computer Science*, San Diego, CA, 1995.

[5] P. Brinch Hansen. Joyce—A Programming Language for Distributed Systems. *Software—Practice and Experience*, 17(1):29–50, January 1987.

[6] Per Brinch Hansen. The programming language Concurrent Pascal. *IEEE Transactions on Software Engineering*, 1(2):199–206, June 1975.

[7] R.E. Bryant. Graph-Based Algorithms for Boolean Function Manipulation. *IEEE Transactions on Computers*, C-35(8):677–691, August 1986.

[8] J.R. Burch, E.M. Clarke, D. Long, K.L McMillan, and D.L. Dill. Symbolic Model Checking for Sequential Circuit Verification. *IEEE Transactions on Computer Aided Design*, 13(4):401–424, 1994.

[9] J.R. Burch, E.M. Clarke, K.L McMillan, D.L. Dill, and L.J. Hwang. Symbolic Model Checking: 10^{20} States and Beyond. In *Proceedings of the 5-th IEEE Symposium on Logic in Computer Science*, pages 428–439, Philadelphia, June 1990.

[10] 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.

[11] D. R. Cheriton. An Experiment using Registers for Fast Message-Based Interprocess Communication. *ACM Operating Systems Review*, 18(4):12–20, October 1984.

[12] D. R. Cheriton. The V Distributed System. *Communications of the ACM*, 31(3):314–333, March 1988.

[13] D. R. Cheriton and W. Zwaenepoel. The Distributed V Kernel and its Performance for Diskless Workstations. *Proceedings of the 9th ACM Symposium on Operating System Principles*, pages 129–140, 1983.

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

[15] E.M. Clarke, E.A. Emerson, and A.P. Sistla. Automatic Verification of Finite State Concurrent Systems Using Temporal Logic Specifications: A Practical Approach. *Proceedings 10th ACM Symposium on Principles of Programming Languages*, pages 117–126, 1983.

[16] E.M. Clarke, E.A. Emerson, and A.P. Sistla. Automatic Verification of Finite-State Concurrent Systems Using Temporal Logic Specifications. *ACM Transactions on Programming Languages and Systems*, 8(2):244–263, April 1986.

[17] C. Courcoubetis, M. Vardi, P. Wolper, and M. Vannakakis. Memory-Efficient Algorithms for the Verification of Temporal Properties. *Formal Methods in System Design*, 1(2/3):275–288, October 1992.

[18] P.J.A. de Villiers. A Model Checker for Transition Systems. In *6-th Southern African Computer Symposium*, pages 262–275, July 1991.

[19] P.J.A. de Villiers. A Model Checker for Transition Systems. *South African Computer Journal*, (8):24–31, November 1992.

[20] P.J.A. de Villiers. Using Formal Validation Techniques to Develop a Microkernel. In *Proceedings: IFIP International Workshop on Dependable Computing and its Applications*, January, 12–14 1998.

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

[22] E.W. Dijkstra. *A Discipline of Programming.* Prentice-Hall, Englewood Cliffs, New Jersey, 1976.

[23] D.R. Cheriton. The V Distributed System. *Communications of the ACM*, 31(3):314–323, March 1988.

[24] E A Emerson and E M Clarke. Using branching time temporal logic to synthesize synchronization skeletons. *Science of Computer Programming*, 2:241–266, 1982.

[25] E A Emerson and J Y Halpern. 'Sometimes' and 'not never' revisited: on branching versus linear time temporal logic. *Journal of the ACM*, 33(1):151–178, January 1986.

[26] E. Allen Emerson and J.Y. Halpern. Decision Procedures and Expressiveness in the Temporal Logic of Branching Time. *Journal of Computer and System Sciences*, (30):1–24, 1985.

[27] E.A. Emerson and C. Lei. Modalities for Model Checking: Branching Time Strikes Back. *Science of Computer Programming*, (8):275–306, 1987.

[28] J. Fernandez, L. Mournier, C. Jard, and T. Jéron. On-the-fly Verification of Finite Transition Systems. *Formal Methods in System Design*, 1(2/3):251–273, October 1992.

[29] W. Fouché. An Efficient Kernel to Support the Client-Server Model. Master's thesis, Department of Computer Science, University of Stellenbosch, Stellenbosch 7600, South Africa, December 1991.

[30] W. Fouché and P.J.A. 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.

[31] N. Francez. *Fairness.* Springer-Verlag, Inc., New York, 1986.

[32] G. J. Holzmann. Design and validation of protocols: a tutorial. *Computer Networks and ISDN Systems*, (25):981–1017, 1993.

[33] J Geldenhuys. An Efficient Model Checker for CTL. Master's thesis, Department of Computer Science, University of Stellenbosch, Stellenbosch 7600, South Africa, December 1999.

[34] J. Geldenhuys and P.J.A. de Villiers. Runtime efficient state compaction in SPIN. In *Proceedings: 5th and 6th International SPIN Workshops, LNCS 1680*, pages 12–21, 1999.

[35] R. Gerth, R. Kuiper, D. Peled, and W. Penczek. A partial order approach to branching time model checking. In *Proceedings of the 3rd Israel Symposium on the Theory of Computing and Systems.*, pages 130–139. IEEE Computer Society Press, 1995.

[36] P. Godefroid, G. Holzmann, and D. Pirottin. State Space Caching Revisited. In *Proceedings: Computer Aided Verification*, pages 175–186, July 1992.

[37] P. Godefroid and P. Wolper. A Partial Approach to Model Checking. In *6-th IEEE Symposium on Logic in Computer Science*, pages 406–414, Amsterdam, 15-18 July 1991.

[38] M. Guillemont. The Chorus Distributed Operating System: Design and Implementation. In P. C. Ravasio, G. Hopkins, and N. Naffah, editors, *Local Computer Networks*, pages 207–223. Proceedings of the IFIP TC 6 International In-Depth Symposium on Local Computer Networks, Florence, Italy, (19-21 April), North-Holland Publishing Company, 1982.

[39] Keijo Heljanko. Model checking the branching time logic CTL. Research Report No 45, Helsinki University of Technology, Digital Systems Laboratory, Department of Computer Science, Helsinki University of Technology, Otaniemi, Finland, May 1997.

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

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

[42] G.J. Holzmann. Tracing Protocols. *AT&T Technical Journal*, 64:2413–2434, December 1985.

[43] G.J. Holzmann. An Improved Reachability Analysis Technique. *Software Practice and Experience*, 18(2):137–161, February 1988.

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

[45] 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.

[46] G.J. Holzmann and D. Peled. An Improvement in Formal Verification. In *Proceedings: FORTE 1994*, pages 177–191, Berne, Switzerland, October 1994.

[47] C. Jard and T. Jeron. On-Line Model Checking for Finite Linear Temporal Logic Specifications. In *International Workshop on Automatic Verification Methods for Finite State Systems*, pages 189–196, Grenoble, France, June 12-14 1989.

[48] P.C. Jorgensen. *Software Testing: A Craftsman's Approach.* CRC Press, Inc., 1995.

[49] L. Lamport. "Sometime" is sometimes "not never"—On the temporal logic of programs. In *Proceedings of the 7th ACM Symposium on Principles of Programming Languages*, pages 174–185, Las Vegas, Jan 1980.

[50] M. Reiser and N. Wirth. *Programming in Oberon: Steps beyond Pascal and Modula.* Addison-Wesley, 1992.

[51] Z Manna and A Pnueli. Tools and Rules for the Practicing Verifier. Technical Report STAN-CS-90-1321, Dept of Computer Science, Stanford University, July 1990.

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

[53] H-P Mössenböck. *Object-Oriented Programming in Oberon-2.* Springer-Verlag, 1993.

[54] S. J. Mullender et al. Amoeba: A Distributed Operating System for the 1990s. *IEEE Computer*, 23(5):44–53, May 1990.

[55] S. J. Mullender and A. S. Tanenbaum. The Design of a Capability-Based Distributed Operating System. *The Computer Journal*, 29(4):289–299, 1986.

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

[57] G.J. Myers. *The Art of Software Testing.* John Wiley & Sons, 1979.

[58] A Pnueli. The temporal semantics of concurrent programs. *Theoretical Computer Science*, 13:45–60, 1981.

[59] A. Pnueli. Linear and branching structures in the semantics and logics of reactive systems. In *Proceedings 12th ICALP*, pages 15–32, 1985.

[60] J P Queille and J Sifakis. Specification and verification of concurrent systems in CESAR. In *Proceedings 5th International Symposium in Programming*, pages 337–351. LNCS 137, 1981.

[61] J P Queille and J Sifakis. Fairness and Related Properties in Transition Systems— A Temporal Logic to Deal with Fairness. *Acta Informatica*, pages 195–220, 1983.

[62] R. F. Rashid. From RIG to Accent to Mach: The Evolution of a Network Operating System. *Proceedings of the ACM/IEEE Computer Society Fall Joint Conference*, pages 1128–1137, November 1986.

[63] H.N. Roux and P.J.A. de Villiers. A Validation Model of the VMTP Transport Level Protocol. In *Proceedings: SAICSIT'96, National Research and Development Conference*, Durban, South Africa, September, 26–27 1996.

[64] J.M. Spivey. *The Z Notation, A Reference Manual.* International Series in Computer Science. Prentice Hall, 2nd edition, 1992.

[65] A. S. Tanenbaum. *OPERATING SYSTEMS: Design and Implementation.* Prentice-Hall International, Inc., 1987.

[66] A. S. Tanenbaum. *Modern Operating Systems.* Prentice-Hall, Englewood Cliffs, New Jersey, 1992.

[67] A. S. Tanenbaum et al. Experiences with the Amoeba Distributed Operating System. *Communications of the ACM*, 33(12):46–63, December 1990.

[68] A. S. Tanenbaum and S. J. Mullender. An Overview of the Amoeba Distributed Operating System. *ACM Operating Systems Review*, 15(3):51–64, July 1981.

[69] Heikki Tuominen. Logic in Petri Net Analysis. Research Report 5, Helsinki University of Technology, Department of Computer Science, Digital Systems Laboratory, Otaniemi, Otakaari 5 A SF-02150 ESPOO, FINLAND, January 1988.

[70] R. Van Renesse, H. Van Staveren, and A. S. Tanenbaum. The Performance of the Amoeba Distributed Operating System. *Software—Practice and Experience*, 19(3):223–234, March 1989.

[71] M. Vardi and P. Wolper. An automata-theoretic approach to automatic program verification. In *LICS'86*, pages 332–344, Cambridge, Massachusetts, June 1986.

[72] B. Vergauwen and J. Lewi. A Linear Local Model Checking Algorithm for CTL. In *Proceedings CONCUR 93, LNCS 715*, pages 447–461, June 12-14 1993.

[73] W.C. Visser. A Run-Time Environment for a Validation Language. Master's thesis, Department of Computer Science, University of Stellenbosch, Stellenbosch 7600, South Africa, December 1993.

[74] N. Wirth and J. Gutknecht. *Project Oberon: The Design of an Operating System and Compiler*. Addison-Wesley, 1992.