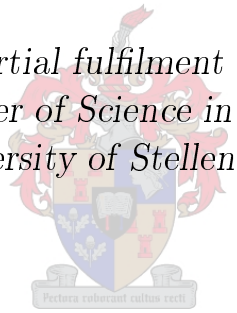
Developing a generic request-processor for systems with limited request processing resources

by

H. Venter

Thesis presented in partial fulfilment of the requirements for the degree of Master of Science in Engineering at the University of Stellenbosch



Department Electric and Electronic Engineering
Stellenbosch University
Private Bag X1, 7602 Matieland

Supervisor: Dr. R. Wolhuter

March 2008

Declaration

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

Signature: .....
H. Venter

Date: .....

Abstract

This thesis describes the design, modelling and implementation of a prototype request-processing software system, which can be used as the basis for a request processing framework for systems with limited request processing resources. Due to design constraints, the request-processor system described here consists of multiple processes.

It is problematic to prove that a multiple process design satisfies the conditions of a set of predefined requirements. One way to verify that such a multiple process design works as intended, is to use modelchecking tools.

The system was verified for correctness and translated into a working prototype software system.

Uittreksel

Hierdie tesis beskryf die ontwerp, modellering en implementering van 'n prototipe versoekverwerking-sagtewarestelsel. Die stelsel kan gebruik word om 'n versoekverwerkingsraamwerk te ontwerp vir stelsels met beperkte versoekverwerkingshulpbronne. Die versoekverwerkingsstelsel bestaan uit veelvoudige prosesse. Die veelvoudige proses-ontwerp was die direkte gevolg van stelselbeperkings.

Dit is problematies om te bewys dat 'n multi-proses-ontwerp korrek funksioneer. *Modelchecking*-sagteware kan gebruik word om te verifieer of 'n stelsel korrek funksioneer.

Die korrektheid van die stelsel is geverifieer voordat die finale prototipe geïmplementeer is.

Acknowledgements

I would like to express my sincere gratitude to the following people and organisations who have contributed to making this work possible:

- Dr. Riaan Wolhuter for his guidance as my supervisor. His comments and expertise helped me to look at the project from different angles. I would also like to thank him for all his support during the past two years. It helped me to keep the thesis on track through testing times.
- Prof. Sias Mostert for his involvement during the early stages of my post graduate studies.
- Mcelory, for all his support, motivational speeches and friendship.
- My parents and the rest of my family for all their love and support.
- All my friends for just being there.

Contents

List of Figures

List of Tables

Listings

Nomenclature

Acronyms

PROMELA	Process Meta-language
SPIN	Simple PROMELA Interpreter
POSIX	Portable Operating System Interface
LTL	Linear Time Temporal Logic
GUI	Graphical User Interface

Chapter 1

Introduction

1.1 Problem statement

Concurrent systems are an integral part of our lives. Large Internet services for example, deal with concurrency on an unprecedented scale. These services process various concurrent data requests. Designing such systems however, is a challenging task.

Threads are generally used for expressing concurrency. The Jetty [1] web server for instance, uses threads to process incoming client connections. The server starts a new thread for every new connection.

Using a thread based design for a request processing system, has its advantages:

- It is relatively reliable; and
- it is easy to maintain and debug.

But it also has two major disadvantages:

- It is resource intensive; and
- it has scaling limits.

In a threaded concurrent system design, each thread uses it own share of the system resources. This means that the system's size limit is proportional to the resources that are available.

The resource availability problem is specially problematic in systems that require a request processing system, but the system only has a limited set of resources available to

[1] An open-source, standards-based, full-featured web server implemented entirely in Java

process incoming requests. Designing request processors for such systems, is a challenging task.

1.2 The scope of the conducted work

The scope of the conducted work for this thesis entailed the design and implementation of a prototype request-processor software system. This design can be the basis of a request processor framework for systems with limited request processing resources.

The imaging-satellite, Sumbandila, was selected for the design and implementation of the prototype described in this thesis.[1]. An imaging-satellite is an example of a system with limited resources. Apart from other possible procedures, an imaging satellite can also make use of a request-processor software system in order to process image processing requests.

Designing and implementing a request-processor system to function as part of a limited resource based system, poses unique challenges, which differ from system to system. In the case of the Sumbandila satellite, the most important challenges were:

- The operational environment of the satellite; and
- the software used on board the satellite.

The operational environment immediately placed a robustness requirement on the design of the request-processor, because singular event upsets, caused by radiation, may have a major impact on the software on board the satellite. The operating system used on board the Sumbandila satellite, QNX, does not allow for process threads. This fact alone aided in the decision to use a multiple process-based design for the request-processor system. In order for the design to work, an interprocess communication scheme had to be devised.

Designing a concurrent system with these properties was a challenging task. The request-processor system had to be as reliable and responsive as possible. Like most concurrent systems, the request-processor also makes use of shared resources. It is problematic to design and implement a concurrent system which has a proper resource sharing scheme while at the same time being reliable and responsive. If not designed correctly, the system will suffer from all sorts of concurrency related problems such as data inconsistency and deadlock. Keeping this in mind, the development of the concurrent request-processor system was divided into four main phases:

1. Modelling the design of the system using the *PROMELA* modelling language.
2. Verifying the logical correctness of the model.
3. Translating the *PROMELA* model into a set of C programs.
4. Testing the final system.

The idea of modelling and verifying the system design originates from the realm of *modelchecking*. Modelchecking enables the designer of a concurrent or distributed software system to verify the correctness of the system. It is commonly used in the design of live-critical systems such as aircraft control systems. In the case of the request-processor system it was used to verify the responsiveness and correctness of the system. It was also used to ensure that the system is free of deadlock.

1.3 Thesis outline

The thesis outline is as follows:

- Chapter 2 describes the design of a request-processor for a system with limited request processing resources. The chapter also discuss the user requirements of such a system.
- Chapter 3 presents a brief overview of modelchecking. The intention was not to provide an in depth discussion on this matter. The chapter primarily states the importance of modelchecking in concurrent or distributed system design and how it can help to prove the correctness of such a design. It is not intended to be a detailed, technical discussion of modelchecking theory. It consists of a short introduction to the *PROMELA* modelling language and the *SPIN* verification system.
- Chapter 4 describes how the request-processor system design described in chapter 2, was modelled in *PROMELA*. The system design had to be modelled so that the *SPIN* verifier could verify logical correctness properties of the request-processor system design.
- Chapter 5 includes the correctness properties that were verified for the request-processor system design. The chapter also provides the results of the *SPIN* verification runs.

- Chapter 6 describes how the verified *PROMELA* model of the request-processor system was translated into the final C-based prototype software system.
- Chapter 7 entails the project conclusions and recommendations for future project development.

1.4 Summary of work completed

The work completed for this thesis resulted in a prototype for a reliable and responsive request processing software system, which can be used as the basis for a request processing framework for systems with limited request processing resources. The designed prototype was verified through formal means to prove its correctness and reliability as far as possible.

Chapter 2

System design overview

2.1 Introduction

Designing a proper working concurrent system was a challenging task. In this chapter, some of the aspects involved in designing a concurrent, generic request-processor for a system with a limited set of request processing resources are introduced. These aspects include a description of the effect of the user requirements and the design constraints on such a system. To assist in this discussion an overview of a request-processor design for an imaging satellite is given. The ideas contained in this design can be used as the basis for designing a generic request-processing framework for systems that have a limited set of request processing resources.

2.2 The effect of user requirements and constraints on the design of a request-processor software system

To successfully design and implement a request-processor software system, the design must be based on a set of user requirements. The most basic user requirements for such a design with limited request processing resources are:

1. The system must be able to accept and respond to client requests.

2. It must be responsive.

3. In order to maximize system performance, the system must be able to process more than one request at a time. From a user perspective, the system must exhibit

concurrent behaviour.

In general, a basic concurrent client-server software system can be used to meet all the requirements as outlined above. The first requirement can be met seeing that the requirement does not state what kind of client requests the system must accept and respond to. The specification of the requests is left to the software designer. The second and third requirements can be met by making use of program threads.

In some cases, considering a system's user requirements for a specific design is not enough. For specialized systems, such as an imaging satellite, environmental constraints must also be taken into account. These constraints can effect decisions made during the design phase of a software system.

The request-processor design described in this chapter used an imaging satellite as a base system. This placed the following additional constraints on the request-processor design:

1. The operating environment of the system
2. The operating system used on board the satellite

The operating environment constraint is severe. A satellite operates in an extreme hostile environment where radiation damages satellite hardware, which has a detrimental effect on satellite software. A request-processor for an imaging satellite must therefore be as robust as possible.

It was mentioned in chapter 1 that the request-processor software design described in this thesis, used the Sumbandila satellite as the base system. This meant that all the above mentioned requirements needed to be taken into account. The first design excluded the operating environment constraint. Software threads, generally found in concurrent server systems, were used. A threaded design is not adequate if the environment constraint is considered. A singular event upset, caused by radiation, can cause threads entering a deadlock state or a system crash. Since tracking which thread processes which request is not reliable, the request may be lost if the thread dies. Furthermore, the operating system used on board the Sumbandila satellite (QNX) does not have thread support.

2.2.1 Limited resources

In order to successfully design and implement request-processor software for a system with limited request processing resources, the limited resources for request-processing must be known. In the case of the Sumbandila satellite, the following resources were chosen:

- the mass memory on board the satellite used to store raw satellite images;
- the imaging schedule on board the satellite;
- the ground communication system on board the satellite; and
- the image processor on board the satellite.

2.2.2 Possible requests

The limited resource constraint placed a limit on the type of requests that the designed request-processor system could accept. If it is assumed that the satellite imaging payload is only able to process certain kinds of image processing requests, this had to be incorporated into the system design. It was decided that the designed request-processor system should be able to process the following requests made by a client:

- a *RAW*-image request
- a process-image request
- a schedule request

A *RAW*-image request makes it possible to request a *raw*, unprocessed satellite image from the satellite based on a provided set of coordinates. It was decided that the system would be able to service this request by making use of two resources on board the satellite: Mass storage and the satellite image acquirement schedule. The flowchart of the servicing procedure of a *RAW*-image request, used in the final request-processor design, is shown in Figure 2.1. The system first checks whether or not the request can be satisfied using an image from mass storage. If no raw-image can be found in mass storage that can satisfy the request, the system checks whether it can make the raw-image request part of the satellite imaging schedule. If this fails, the request is logged so that it can form part of a future imaging schedule.

A process-image request makes it possible to request a processed image from the satellite. It was decided that the system would be able to process requests that either contains a user-defined image, or a set of coordinates which enables the system to fetch a raw-image before processing it. The procedure used to fetch the raw-image is exactly the same as the one used when processing a raw-image request.

The last type of request that the system would be able to process, is a schedule request. The schedule request is unique, in that it does not require any kind of image processing

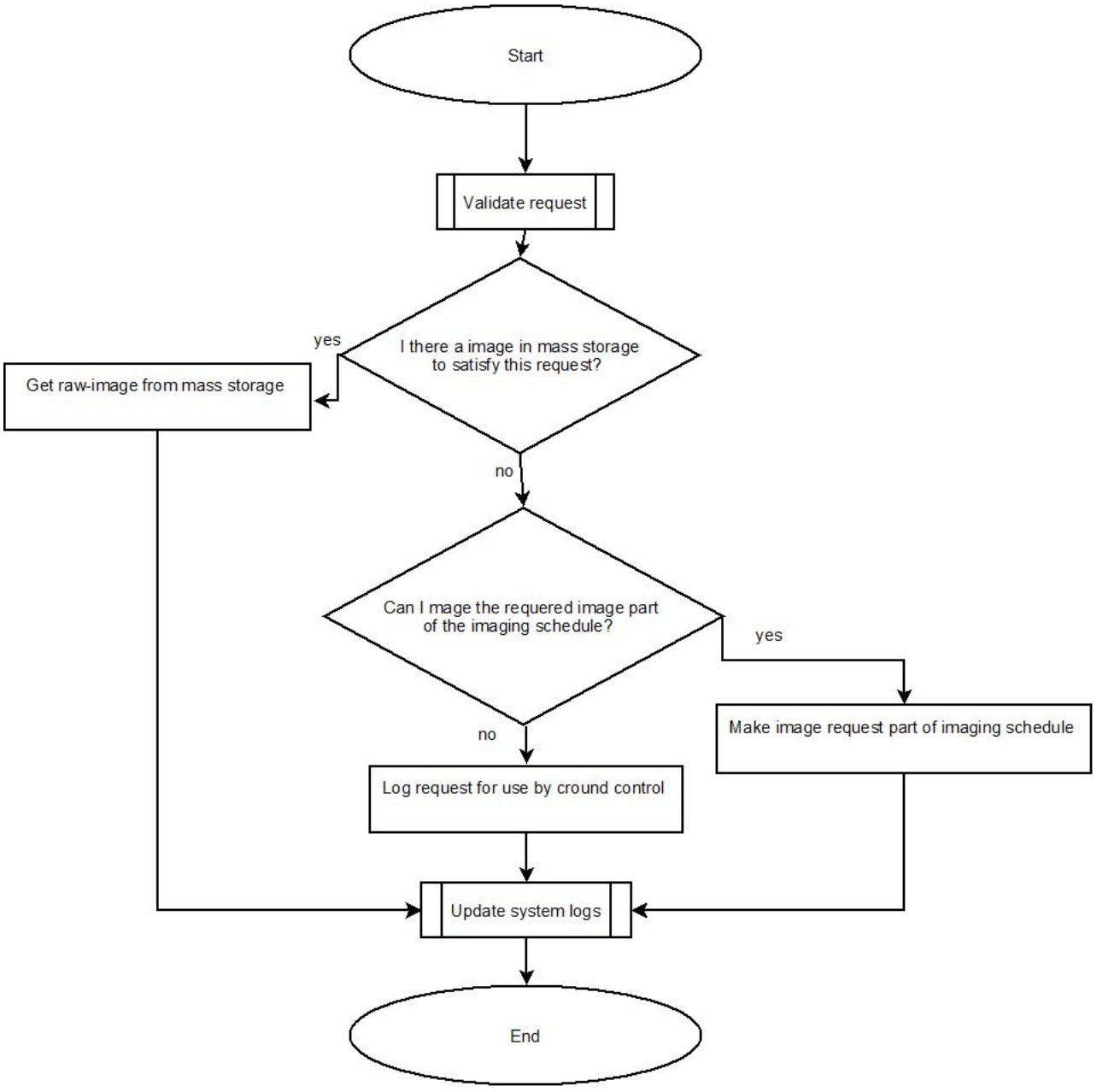


Figure 2.1: Flowchart for handling *RAW*-image requests.

or image acquirement. The request has only one purpose: To enable a client to add an image-request to the imaging schedule of the satellite.

2.3 Multiple process design

In the previous section some design requirements and constraints for a satellite request-processor system were highlighted. It was stated that a standard concurrent sever design could not be used to satisfy the responsiveness and concurrent behaviour requirements of the system. To solve this problem, a multiple process design was used. The idea behind the multiple process approach is that not only will it satisfy the system requirements, but it will also make the system easier to maintain. A multiple process design has an added advantage, namely *a separation of concerns*. This means that a failure of one process will not severely affect the other processes in the system. Suppose an image processing process fails, the multiple process design enables the system to still accept and process other types of requests that do not involve the image processing process. It is also easier in a multiple process design to add functionality that can keep track of requests in the system.

The multiple process design that was used for the satellite request-processor prototype, is depicted in Figure 2.2. A high-level description of the functionality of each process is given in the following sections:

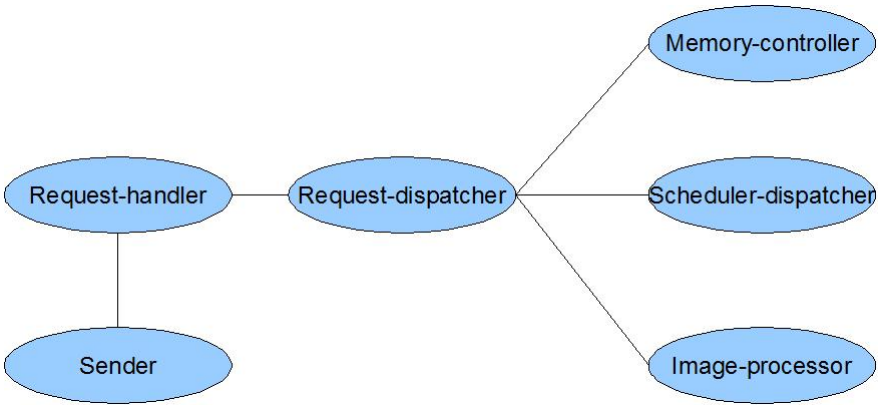


Figure 2.2: The processes used to build the request-processor.

The request-handler

The purpose of the request-handler is to accept client requests and to pass it on to the request-dispatcher. The acceptance of a request can include:

- Validating the request. Validation includes aspects such as verifying the consistency of the request payload and the validity of the coordinates contained in the request.
- Running a security check.

The request-handler further accepts processing results from the request-dispatcher and is also responsible for sending any responses required from the system to the client by using the sender-process. The functionality of the request-handler does not seem complex. When looking at the process diagram in Figure 2.2, one might consider incorporating this functionality into the request-dispatcher process. This, however, does not follow the *separation of concerns* design pattern. By having a separate process for accepting the requests, the system is able to accept requests even if the rest of the system is not available. The request-handler process can also be used as a buffer for client requests that need to be processed by the request-processor system.

The request-dispatcher

The request-dispatcher is the most important process in the request-processor system. The main function of the request-dispatcher process is to control the allocation and use of resources that are needed to successfully process a client request. In order to understand the controlling role of the request-dispatcher, consider the example depicted in Figure 2.3.

In this example the request-dispatcher must process an image-processing and raw-image request. In order to process the requests the following resources are needed:

- **Image-processing request:** The memory-controller to fetch the raw-image required for processing, and the image-processor to perform the required raw-image processing.
- **RAW-image request:** The memory-controller to fetch the required raw-image from memory.

It was already stated earlier that program threads could not be used in the designed system. This means that internally, requests can only be handled in a synchronous manner. This implies that the memory-controller can only be used to satisfy one request at

a time. The request-dispatcher starts off by sending a raw-image request (part of the image-processing request processing sequence) to the memory-controller. It now notes that the memory-controller is in use. It is fair to assume that the response time of the memory-controller is not instantaneous. The request-dispatcher process cannot wait for a response from the memory-controller. This is partly due to the responsiveness requirement imposed on the designed request-processor system. The request-dispatcher must now try to start the processing of the waiting raw-image request. In the example, however, the request-dispatcher has no other option than to wait for the memory-controller to return with a response seeing that the memory-controller is needed to successfully process the raw-image request. If the waiting request was a schedule request, the request-dispatcher would have been able to start the processing of the request. The request-dispatcher would have been able to start a schedule request, because the only resource needed to complete the request, namely the scheduler-dispatcher, is not in use.

After receiving the raw-image from the memory-controller, the request-dispatcher sends the raw-image to the image-processor. It also starts processing the waiting raw-image request by sending it to the memory-controller. The example shows that the request-dispatcher's resource controlling nature makes it an important and complex process in the request-processor system. It also shows that the request-dispatcher is able to maximally utilize the available system's request processing resources.

The system resources

The sender, memory-controller, scheduler-dispatcher and image-processor processes that are part of the multiple process system design (see Figure 2.2), are abstract representations

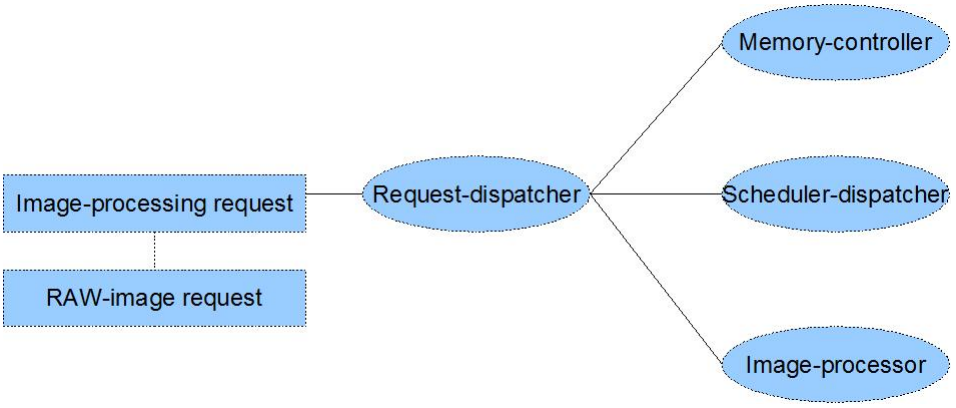


Figure 2.3: Request-dispatcher control example.

of actual satellite resources. Each process abstracts the following satellite resources:

- **Memory-controller** The software and hardware subsystem that controls and accesses the mass memory on board the satellite that is intended for raw-image storage.
- **Scheduler-dispatcher:** The software subsystem that controls the imaging schedule on board the satellite.
- **Image-processor:** The hardware and software subsystem controlling the image processor payload on board the satellite.
- **Sender:** The software and hardware subsystem that enables the satellite to communicate with ground stations.

The above-mentioned resource abstractions correspond to the request processing resources of the satellite system for which the request-processor was designed (see section 2.2.1).

The following assumptions about the satellite resources were made in order to design and implement a working prototype of the request-processor system described in this thesis:

1. Each resource payload has its own control software.
2. The control software provides a simple API, so that other processes can use the corresponding resource.
3. The control software is similar to a normal device driver. It can accept and respond to processing requests received from processes in the system.

These assumptions have one major advantage: It makes it possible to design and implement the request processing resources and the request-processor separately. The separate systems will be able to successfully interact with each other as long as they use the same API specification. This design strategy has been successfully used in object orientated software systems and in a variety of engineering projects.

2.4 Interprocess communication

The multiple process design described in the previous section is a solution that can satisfy all the requirements and constraints imposed on the satellite request-processor system.

The design, however, has its own set of design challenges that needs to be solved. One of the most prominent of these design challenges, is interprocess communication.

To achieve interprocess communication between two simple processes that do not share any resources, is not problematic. There are solutions for this problem. If more than one process and shared resources are involved, the problem is more complex. The request-processor system is such a multiple process, resource sharing system. In this section the solution to the interprocess communication problem for the request-processor is discussed. This section also describes some failed interprocess communication designs that were conceived during the design phase of the request-processor system.

2.4.1 Using asynchronous queue processes

The design

The idea of this design was to use separate processes for each communication channel that was needed for interprocess communication. The idea is depicted in Figure 2.4. Message

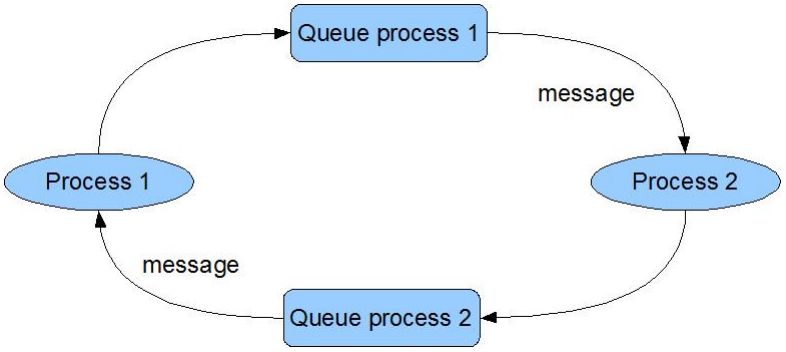


Figure 2.4: Interprocess communication using queue processes.

passing between processes is achieved by pushing messages to, and pulling messages from a queue process. With this interprocess communication design, the queues involved can be used as message buffers. Each queue process was designed to be synchronous in order to ensure message consistency. This meant that only one process at a time could access the queue. The queue also had to guarantee some form of fairness. Without this it becomes possible that one process can continually push messages to the queue which makes it impossible for another process to pull any message from the queue. One of the first queue process designs is listed in Listing 2.1.

Listing 2.1: Pseudo-code of an asynchronous queue process

```
process queue(chan readerIn,readerOut,writerIn,writerOut){
int lp =1;
Begin Repeat
  If (in State 1)-> /*In state 1 only service read requests */
    If (recieve token on readerIn)->
      Read message from queue;
      send message out on readerOut;
      lp=2;
    If (nothing received on readerIn)->
      lp=2;

    Go to start of loop;

  If (in State 2)-> /*In state 2 only service write requests */
    If (receive message on writerIn)->
      Update message queue
      lp=1;
    If (nothing received on writerIn)->
      lp=1;

    Go to start of loop
End Repeat
}
```

By studying the queue process design in Listing 2.1, one can see how the queue process ensures fairness. It uses the concept of states. In the first state it only accepts requests wanting to read a message from the internal message buffer. After receiving such a read request it does the required processing and moves to the second state. In the second state, the queue process will only accept a write request. If it receives a write request, it does the required processing and moves back to the previous state. If the queue process does not receive a request in any given state, it times out and moves to the next state. This design ensures that every process using the queue will always eventually have exclusive access to the internal message buffer of the queue process.

Problems with the design

Theoretically, the asynchronous queue processes interprocess communication design provides an effective means for the processes in the request-processor system to send messages to each other. The design, however, reflects shortcomings. In order to provide communi-

cation for all the processes in the request-processor system, at least twelve queue-processes (two for each process in the system) are needed. This also means that the request-processor would consist of at least eighteen separate processes. Even if the number of processes was not a problem, the asynchronous queue processes design would have added an unnecessarily high level of complexity to the request-processor system, making maintenance and testing problematic.

2.4.2 Round-robin message passing

The design

The idea of the round-robin design was to use one token which is passed between all the processes. The idea is depicted in Figure 2.5. The round-robin scheduling policy is a known and tested network scheduling policy which is most commonly found in token-ring network topologies. It has also successfully been used over the years to provide interprocess communication for multiple process systems. During the development phase of the interprocess communication scheme for the request-processor system, a round-robin based design was tested. In the design all the request-processor processes were organized in a classic ring formation as depicted in figure 2.5. This layout has one major advantage: It makes it possible to create an interprocess communication design that can cater for absolute fairness in the sequence of message delivery in the system. In the case of the request-processor system, it helps to ensure sole usage of a system resource at any given time.

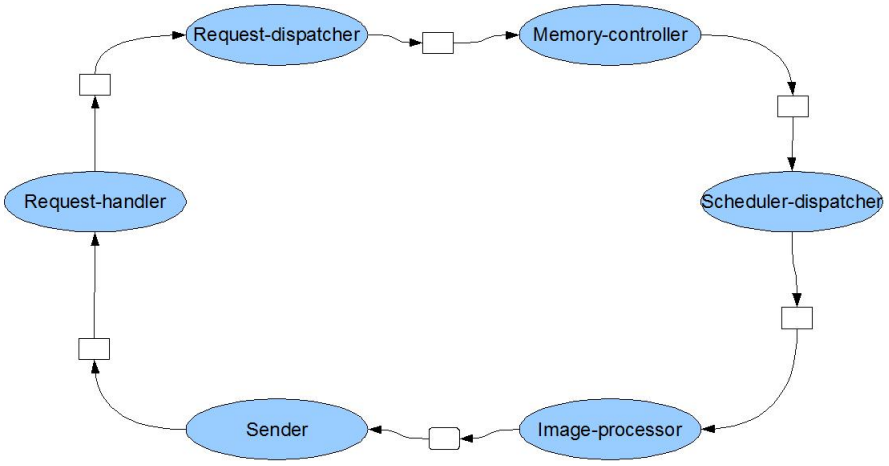


Figure 2.5: Interprocess communication using one token and round-robin scheduling.

For clarity of this statement, consider a scenario where the image-processor needs to write an image to mass memory, and the request-dispatcher needs to read an image from mass memory. If no *dirty-read* operations are allowed in the system, both the image-processor and request-dispatcher must obtain exclusive access to mass memory before performing the read or write operation. Now lets assume that the image processor request has been in the system longer than the request dispatcher request. In order for the image-processor request to be processed first, it will have to have the smaller timestamp of the two requests. When the token gets to the image-processor, it inspects the token and finds that the token is *free*, it tries to acquire the token by placing a *token-request* message on it before passing it on to the sender process. It is worth noting that in the designed round-robin message passing scheme, a process can only try to place something on the token if the token is marked as *free* or if the token has a *token-request* on it. The scheme was designed to first process the *token-request* with the smallest timestamp in the system.

After passing on its *token-request* message, the image-processor waits for the token to come around again. If the image-processor receives the token, and it contains the image processor's own *token-request* message, it knows it can place its memory write request on the token and pass it on to the sender process. When the token gets to the memory-controller, the memory-controller processes the request and places the response on the token before passing the token to the scheduler-dispatcher. When the token containing the memory-controller response reaches the image-processor, it removes the response and marks the token as *free* before sending it to the sender process. Any process can now try to acquire the token for processing. The fairness of the system is thus ensured.

Problems with this design

Two problems surfaced in the testing of the round-robin design. The first problem with the design is that, not only does the design share the strengths of the networking related token-ring topology and round-robin scheduling policy, it also shares its weaknesses. Some of these weaknesses are:

- The ring can be broken, which disables the message passing capability of the protocol.
- The token can become corrupted, which disables the message passing capability of the protocol.

The second problem is related to the complexity of the round-robin design. Due to the nature of the design, every process needs to be part of the ring. This means that

every process needs to know, and is dependent on, every other process that forms part of the token-ring. This additional process awareness requirement increases the complexity of each process involved in the token-ring, which in turn makes system maintainability a problematic and error-prone affair. It was mainly because of this reason that it was decided not to use the round-robin design to solve the interprocess communication problem of the request-processor system.

2.4.3 The final design

The final interprocess communication design used for the request-processor system, incorporated all the positive aspects of the mentioned interprocess communication designs. The final design is depicted in Figure 2.6

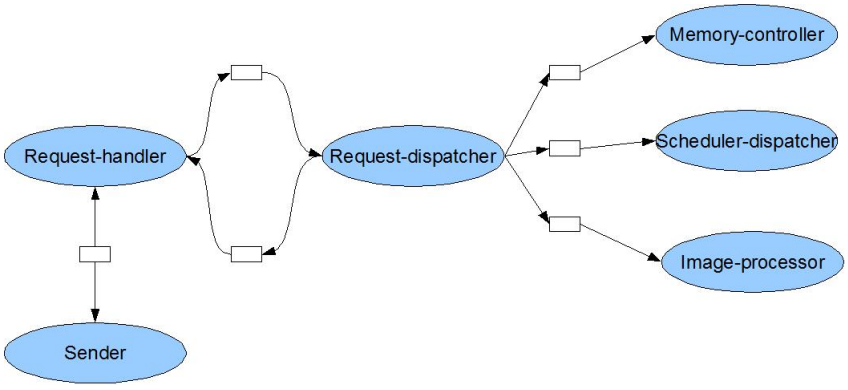


Figure 2.6: The interprocess communication design of the request-processor system.

Common process design features

All the processes in the interprocess communication design depicted in Figure 2.6 share a state-based design similar to the one used by the queue processes of the asynchronous queue processes communication design described earlier. The reason why all the processes share this common state-based design is because it ensures a level of fairness. A simplified design of the request-dispatcher process is listed in Listing 2.2.

The mapping between the states of the request-dispatcher, and the different processes it communicates with, becomes clear when one considers Figure 2.6 and Listing 2.2. According to figure 2.6, the request-dispatcher must communicate with four processes within the request-processor system, namely: the request-handler, the memory-controller, the

Listing 2.2: Common processes design for fair interprocess communication

```
/*High level Pseudo-code design of the request dispatcher,
depicting its state-based nature*/
process requestDispatcher()
Begin Repeat
  /*This process is a service provider,
   it must send the initial token in state 1*/
   If (in State 1)->/*send init token to request handler*/
     RDRHTokenRingOut!tokenInit(NONE,NONE);
     lp=2;
     Go to start of loop;
   If (in State 2)->
     /*In this state only service and accept requests
     from the request handler*/
     lp=3;
     Go to start of loop;
   If (in State 3)->
     /*In this state only service and accept requests
     from the scheduler dispatcher*/
     lp=4;
     Go to start of loop;
   If (in State 4)->
     /*In this state only service and accept requests
     from the image processor*/
     lp=5;
     Go to start of loop;
   If (in State 2)->
     /*In this state only service and accept requests
     from the memory controller*/
     lp=2;
     Go to start of loop;
End Repeat
```

scheduler-dispatcher and the image-processor. These communication channels account for states two, three, four and five present in listing 2.2.

State 1 shown in Listing 2.2, is a special initialization state. A similar initialization state is present in every process in the request-processor system that provides a service to other processes in the system. In the request-processor system the following are considered to be service-providing processes:

- **The request-dispatcher:** The request-dispatcher process provides a service to the request-handler process by processing and responding to requests it receives from the request-handler.
- **The resource processes:** The memory-controller, scheduler-dispatcher and image-processor processes provide a service to the request-dispatcher. The processes accept and respond to request-dispatcher requests. The sender process provides the same service to the request-handler process.

Each service-providing process has a initialization state in which the initial communication channels between the service-providing process and its clients, are initialized. Each process in the system is also responsible for the buffering of messages it wants to send to another process. The request-dispatcher process is, for example, responsible for buffering three different types of internal requests: memory-controller requests, scheduler-dispatcher requests and image-processor requests.

The message buffering and state-based design of the processes of the system, enables it to exhibit concurrent behaviour. For example, if the request-dispatcher reaches the state where it can only communicate with the image-processor and it wants to send a request to the image-processor, but the image-processor is unable to accept the request, the request-dispatcher can simply queue the request and continue to its next state. The state-based design of the request-processor processes have one added advantage: It is simple to extend a process in order for it to communicate with a new process. All that needs to be done is to add a state to the process, in which the process can only accept and respond to requests from the new process.

Communication between the request-handler and request-dispatcher

The communication protocol between the request-handler and request-dispatcher is based on the round-robin interprocess communication design as described earlier. The round-robin design was used, because the request-dispatcher and request- handler processes need to be aware of each other.

The fact that there are only two processes, the request-handler and the request-dispatcher, means that this specific round-robin design does not suffer from a long round-trip time and implementation complexity.

To understand how the request-handler and request-dispatcher processes communicate with each other, consider the example depicted in Figure 2.7.

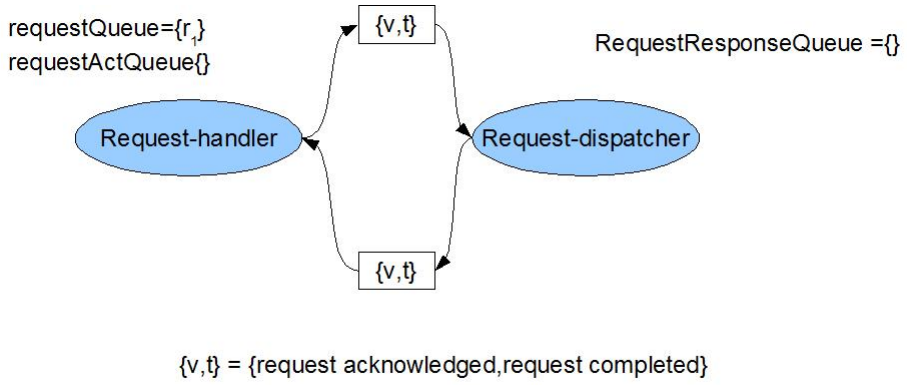


Figure 2.7: Interprocess communication between the request handler and request dispatcher.

The token in the example is a double v, t, where v represents a request that is acknowledged by the request-dispatcher and t represents a request that was completed by the request-dispatcher. In the case of the request-handler, v represents a request that needs to be processed by the request-dispatcher. It is worth noting that each completed request contains additional response information. Part of the response information can for example, be a processed image which the request-handler must send back to the client who was responsible for the initial image-processing request.

The request-handler process maintains a couple of internal message queues. Two of these queues are shown in the example: The request queue and the request acknowledgement queue. The request-handler uses the request queue to store unprocessed requests it receives from a client (mobile ground stations). It uses the request acknowledgement queue to store requests that are acknowledged by the request-dispatcher. If the request-dispatcher acknowledges a request, it implies that the request has been queued for processing. It does not mean that the request has been processed by the system. Every request that the request-handler sends to the request-dispatcher, will always first be acknowledged before it will be processed.

The request-dispatcher process maintains the following internal queues:

- A queue for every different type of client request.

- A queue containing requests for the memory-controller process.
- A queue containing requests for the scheduler-dispatcher process.
- A queue containing requests for the image-processor process.
- A queue containing the processed requests to be sent back to the request-handler process.

The example in Figure 2.7 only depicts the request-response queue, which contains the processed requests that need to be sent back to the request-handler. Now consider the example in Figure 2.7: The request-handler cannot send any request to the request-dispatcher without having access to the ring-token. It waits until it receives the ring-token from the request-dispatcher and then checks to see whether or not the request-dispatcher has acknowledged a request or sent back a completed request. A completed request is a request that has been completely processed by the request-dispatcher. In the example, neither an acknowledgement nor response is present on the token. It now places the first request in the request queue, r_1, onto the token and sends it to the request-dispatcher. Note that the request handler did not remove the request from the request queue.

When the request-dispatcher receives the token containing request r_1, it inspects the request type to see whether or not there is space for the request in the appropriate internal request queue. If the request-dispatcher finds that it is not able to accept the request, it removes it from the token, places a completed request (if any) onto the token and sends it back to the request-handler. Let's assume that the request-dispatcher is indeed able to accept request r_1. In this case the request-dispatcher removes the request from the token and places it into the appropriate internal request queue. It then sends back a token to the request-handler which contains an acknowledgement of request r_1 and a completed request (if any).

On receiving the token from the request-dispatcher, the request-handler again inspects the token for a request acknowledgement and a completed request. This time it finds the acknowledgement of request r_1. It removes request r_1 from its internal request queue and places it in the acknowledge request queue. It now checks to see if there are any more requests that need to be processed by the request-dispatcher. If there is, it sends a token containing the request to the request-dispatcher. Otherwise, the request-handler sends an empty token back to the request-dispatcher.

Now, consider the scenario where the request-dispatcher has completely processed request r_1. The request-dispatcher places the completed request onto the token and passes

it to the request-handler. It also updates its request response queue. On receiving the token, the request-handler runs through its normal token inspection procedure. This time it finds that the request-dispatcher has completed the processing associated with request r_1, and removes the response from the token. It also removes the corresponding request from its internal request acknowledge queue and places it together with the response in the sender queue so that the sender process can send the response to the client.

When studying the example, it is worth noting that neither the request-handler nor the request dispatcher keep the token to themselves. Each process inspects the token, updates the token and passes it back to the other process immediately. This immediate token passing scheme ensures fairness between the two processes.

Communication between the system resource processes and the request-dispatcher

The communication scheme between the request-dispatcher and the system resource processes is similar to the round-robin scheme used to achieve communication between the request-handler and request-dispatcher processes, in that it is token based. A process can only send a message to another process if it has a token. To understand the protocol, consider the example depicted in Figure 2.8.

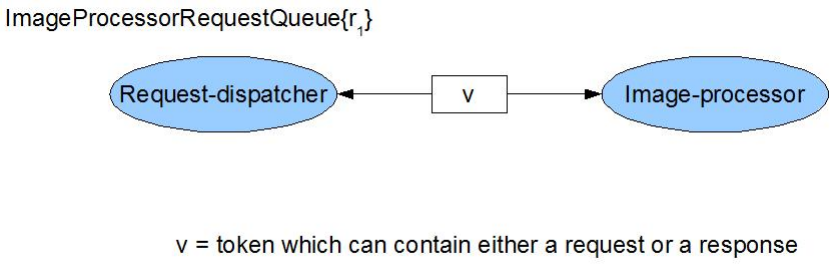


Figure 2.8: Interprocess communication between the request-dispatcher and image processor.

The first thing that is immediately apparent from Figure 2.8, is that the communication between the request-dispatcher and the image-processor is bidirectional. The big difference between the round-robin token passing scheme used by the request-dispatcher ↔ request-handler, and the token passing scheme used by the request-dispatcher ↔ system resource process, is the way in which the token is processed. In the previous section it was stated that the token passed between the request-dispatcher and the request-handler cannot be kept indefinitely by any one of the processes. As soon as either process receives the token,

it must process it and pass it on. This behaviour pattern does not occur in the case of the request-dispatcher and a system resource depicted in figure 2.8. The request-dispatcher shares a token with each system resource process, as can be seen in figure 2.6. The processes that share a token are allowed to keep it as long as they needed. This kind of *token-hogging* is possible if the assumption is made that a system resource is only capable of serving a single request at any given point in time for any single process it services. This single-request-per-serviceable-process was a fair assumption in the case of the system resources used for the the request-processor system.

Now, consider the example in Figure 2.8. In this example the request-dispatcher has a request, r_1, in its image-processor request queue. The request-dispatcher can receive two types of tokens from the image-processor. The first type is an *init* token which the image-processor sends when it initializes. This tells the request-dispatcher that the token is *free* and that it can place a request on the token and send it to the image-processor for processing. The second type of token is a *response* token. This token tells the request-dispatcher that the image-processor has completed the processing as required for the request-dispatcher's last request, and that the request can be removed from the request-dispatcher's internal image processor request queue.

For the given example, assume that the request-dispatcher first receives an *init* token from the image-processor. The request-dispatcher inspects whether or not it has a request that needs to be sent to the image-processor for processing. In the example, it needs to send request r_1, but if it turned out that the request-dispatcher had nothing to send, it is allowed to keep the token until it does have a request to send to the image-processor. It now places request r_1 onto the token and sends the token to the image-processor. The image-processor receives the token and starts processing the request. It holds on to the token until it has finished processing the request. It then places the processing result on the token and sends it to the request-dispatcher. On receiving the token, the request-dispatcher sees that the token contains a request response and updates its internal queues.

2.5 Summary

This chapter provided an overview of a request-processor design for a system with a limited set of request processing resources. The first part of the chapter discussed the most important requirements and constraints that influence the design of a request-processor for such a system. The rest of the chapter provided an overview of the multiple processes forming part of a request-processor system that was developed, based on the basic set

user requirements and constraints for an imaging satellite. Each process of the request-processor system was also highlighted. The multiple process design was the cause of some specific design problems that needed to be solved. The most important problem was interprocess communication. The chapter explored two failed interprocess communication designs, developed during the design phase of the request-processor system. The chapter concluded with a description of the interprocess communication solution as used in the final design and implementation of the request-processor system, based on the requirements mentioned earlier in the chapter.

Chapter 3

Modelchecking

3.1 Introduction

Dijkstra once said:

> "Program testing can be used to show the presence of bugs, but never to show their absence" [2].

This statement is especially true for concurrent software. The reason for this is that the non-determinism of concurrent system executions makes it difficult to devise a traditional test suite for sufficient coverage. According to [3], the fundamental problems with concurrent systems are related to both the limited controllability of events in a concurrent system's distributed executions and the limited observation of these events.

It is widely agreed that a system is correct if it meets its design requirements. Alternatively stated, a well -designed system provably meets its design requirements. How does one show this? It is not enough just to show that a designed system meets all its requirements. One also needs to show that a system cannot fail to meet its requirements. Software correctness testing can be compared to the difference between a hypothesis and a mathematical proof. A hypothesis is based on the outcome of a few experiments. A hypothesis is not considered a proof because it is possible that the hypothesis may fail in a future experiment. With this in mind, how can it be proven that a piece of concurrent software is correct? Using standard mathematics is not much of an option in the software domain. A thorough handwritten mathematical proof of even the simplest distributed system can challenge the most hardened of mathematicians. Mechanical proof procedures do also not hold much promise; it was shown that it is fundamentally impossible to construct

a general proof procedure for arbitrary programs. [1]

Fortunately, it is possible to mechanically verify the correctness of distributed systems, using modelchecking tools. This chapter introduces some model-checking concepts and tools. These tools were used to aid in the design of the system described in this thesis. It is worth noting that the aim of this chapter is not to provide an in depth, theoretical discussion, but rather just to provide a short introduction to some aspects of the modelchecking realm. [2]

3.2 Verification models

In order to prove certain properties of a distributed system, a model is needed that describes the following:

- the set of facts that needs to be verified for the system,and
- the relevant aspects of the system that are needed to verify this set of facts.

Models that possess the above mentioned properties are known as verification models. One more noticeable property of verification models is a high level of abstraction. A verification model is used together with a verification tool to test a modelled system for correctness. There are many modelchecking tools available [3]. Choosing which one to use depends mainly on the functionality required and ease of use of the tool. For the purpose of the prototype developed in this thesis, *SPIN* was used to verify the prototype design for correctness. *SPIN* is a verification tool that verifies models that are written in the *PROMELA* modelling language. *SPIN* was chosen for the following reasons:

- The *PROMELA* modelling language used to define system models closely resembles a modern programming language such as C. This makes it easier to understand and use. The fact that *PROMELA* closely resembles a modern programming language also helps when translating a verified system model into working program code.
- *SPIN* has a simple, effective user interface which has functionality that aids in the modelling process.

The sections that follow introduce basic concepts of both *SPIN* and *PROMELA*.

[1] The insolvability of the Halting problem, proven by Alan Turing, is a good example (see [4])
[2] A more in depth, theoretical discussion can be found in [3]
[3] See [5] for a list of available verification tools

3.3 PROMELA

PROMELA is a verification modelling language. It is used to describe models of distributed systems. This description is then used as the input for a verification tool such as *SPIN*. The emphasis of a *PROMELA* model is placed on the synchronization aspects of the distributed system being modelled. *PROMELA* models seldom focus on the computational aspects of a distributed system being modelled. There are good reasons for this:

1. The design and verification of correct coordination structures for distributed systems tend to be more complex than the design of a non-interactive sequential program.

2. The logical verification of the interaction of distributed systems can be done more thoroughly and more reliably than the verification of even the simplest computational procedure. This computational cost to perform the logical verification may be high, but the advantages outweigh the disadvantages.

The first point can be understood if one considers a simple non-interactive sequential computational unit, such as the computation of compound interest. The correctness of such a computational unit can easily be verified by using a well defined traditional test suite. The same cannot be said for any sort of concurrent system. It can be difficult verify the correctness of even a simple two-process concurrent system.

PROMELA takes the above mentioned points into account. The language is deliberately designed to encourage the use abstraction to hide the purely computational aspects of a design, and to focus on the specification of the process interaction at the system level. As a result, *PROMELA* contains many features that are not found in mainstream programming languages. It is worth noting that a verification model differs in at least two aspects from a program written in a mainstream programming language such as C.

- A verification model is a abstract representation of a system. It only contains the aspects of the system that are needed to verify required correctness properties.

- A verification model often contains things that typically are not part of the system implementation code. Specifications of correctness properties for example, are always either implicitly or explicitly part of the verification model.

Main parts and constructs of a *PROMELA* model and the *PROMELA* language will now be introduced and discussed. The discussion will focus on a verification model of the classic mutual exclusion problem. The model can be seen in Listing 3.1.

3.3.1 The mutual exclusion problem as example

The mutual exclusion problem is one of the best-known problems in concurrent system designs. The challenge is to find a way to grant mutual exclusive access to shared resources, to processes in a distributed system. Consider the example where a single file must be accessed asynchronously, by many reader and writer processes. It is impossible to guarantee the integrity of the file without ensuring that each process gets exclusive rights to file.

One of the first attempted solutions of problem was described by Edsger Dijkstra. The first version of the algorithm was devised by the Dutch mathematician T. J. Decer.

The *PROMELA* model in Listing 3.1, outlines a general scheme for the mutual exclusion problem. It is worth noting that all the implementation detail is omitted from the model. Abstracting implementation detail is a trademark of any well formed model.

The model in Listing 3.1, is based on a turn based scheme. The semaphore guarantees that only one user process can enter its critical section at any given point in time. In the model a process can only enter its critical section once. Fairness cannot be guaranteed if a process continually requests access to its critical section.

3.3.2 Basic *PROMELA* object types

One way to model a distributed system and all its coordination problems is by making use of a finite state machine. A finite state is able to describe all the properties of a distributed system. A finite state machine is also relatively simple to design. The biggest disadvantages of finite state machines are that they can be complex to write and understand.

PROMELA makes it possible to define verification models, using three specific types of objects:

- processes,
- state variables, and
- message channels

All of these objects mentioned above, can be translated into finite state machines by using simple translations [4]. It is worth noting that in *PROMELA*, all processes are seen

[4] This translation process is described in [3]

Listing 3.1: Mutual exclusion model

```
1  mtype = {p,v};
2
3  chan sema = [0] of { mtype };
4
5  proctype dijkstra()
6  {
7          do
8                  :: sema!p -> sema?v
9          od
10 }
11
12 proctype user()
13 {
14         sema?p;
15                 /* critical section */
16         sema!v
17                 /* non-critical section */
18 }
19
20 init
21 {
22         atomic {run dijkstra();run user(); run user()}
23 }
```

as global objects while message channels and state variables can either be local or global to a process.

3.3.3 Processes

Processes are used to describe the behaviour of a system in *PROMELA*. A *PROMELA* model is thus not of much use without at least one process definition. In *PROMELA*, are defined in `proctype` declarations. Listing 3.1 contains two `proctype` declarations namely: *dijkstra* (lines 5 - 10) and *user* (lines 12 - 18).

The *PROMELA* grammar rules defines a `proctype` as:

```
proctype: [ active ] PROCTYPE name '(' [ decl_lst ]')'
  [ priority ] [ enabler ] '{' sequence '}'
```

This grammar rule loosely translated means the following (the *active* keyword is omitted from the translation): A `protype` must have a unique name which is optionally

followed by a parameter list. A `proctype` may also contain zero of more data declarations (state variables) and must contain one or more statements. The *user* `proctype` in Listing 3.1 consists of the name *user*, no parameter list and two statements.

The `init` process

In *PROMELA*, a `proctype` definition only defines the behaviour of a process, it does not execute it. Initially, only one process, if it is present, is executed: A process of type `init` which can be declared explicitly in every *PROMELA* specification. The `init` process is comparable to the `main()` function of a standard C program.

The `init` is not meant to be used as a way to model a process of a system. The main purpose of the `init` process is to initialize the modelled system. It is used to initialize global variables, create message channels and create processes. The `init` processes of the mutual exclusion model shown in Listing 3.1 (lines 20 - 23) creates an instance of the `dijkstra` process and three instances of the `user` process.

The creation of the processes is done by making use of the special *PROMELA* unary operator; `run`. Processes that are started in the `init` process execute concurrently. The `init` process in the mutual exclusion model terminates after starting the last `user` processes.

3.3.4 State variables

In *PROMELA*, state variables are represented by variables and data types. Variables are used to store either information about the system as a whole or information that is local to a specific process, depending on where the declaration for the variable is placed. *PROMELA* has five predefined primitive data types. Each of these types has a set range of values they can store. The five data types and their ranges are summarized in Table 3.1. The scope of *PROMELA* variables are global, if declared outside all process declarations, otherwise they are considered local to the process where they are defined. The meaning and use of the basic *PROMELA* data types are similar to those used in mainstream programming languages such as C and Java.

Arrays

In *PROMELA*, variables can be declared as arrays. A typical array declaration has the following form:

Type name	Size(bits)	signed/unsigned	range
bit	1	unsigned	0..1
bool	1	unsigned	0..1
byte	8	unsigned	0..255
short	16	signed	$-2^{15}..2^{15}-1$
int	32	signed	$-2^{31}..2^{31}-1$

Table 3.1: Basic *PROMELA* data types.

```
byte state[N]
```

N can be any positive integer value (technically N has a upper bound of: $2^{31}-1$). The range of valid indexes for array *state* in the example is $0..N-1$. The following example illustrates how to access a *PROMELA* array element:

```
integer c = 2;
b = state[c-1];
```

In the statements above the value of the second element in the array *state* is assigned to the variable b.

3.3.5 Message channels

Previously it was stated that a *PROMELA* model is not of much use without at least one process being modelled. There is also not much use for a modelling language that does not have a way to model the sharing of information amongst processes. Without information sharing the concurrent behaviour of a distributed system cannot be modelled.

One way *PROMELA* models the sharing of data amongst processes (inter process communication) is by using message channels. Line 3 in Listing 3.1 shows a simple message channel declaration. The example shows that message channels declarations consist of four main parts:

1. the `chan` keyword,
2. a name (`sema`),
3. message capacity (0), and
4. message format (`mtype`).

Listing 3.2: Channels with multiple values

```
sema!expr1,expr2,expr3
sema?var1,var2,var3
```

Message channels can be either be declared locally or globally. The message format part of the declaration defines all the different fields of messages that can be passed to the message channel. If messages passed to a message channel have more than one field, for example an integer or boolean value, the declaration will be:

```
chan sema = [0] of { int,bool };
```

The message capacity part of a message channel declaration is an artifact of the purpose for which the *PROMELA* modelling language was designed: to model protocols. The capacity field defines the number of messages that can be stored in a message channel (protocol message queues). If a message channel has a message capacity of 0, it cannot store any messages. In this case it can only pass on messages (synchronous message passing).

Sending and receiving messages

The following statement can be found on line 16 of Listing 3.1

```
sema!v
```

The statement sends the value of `v` to the channel `sema`, that is, it appends the value of `v` to the tail of the channel's message queue.

The statement

```
sema?p
```

receives the value at the head of the message queue of the message channel `sema` and stores it in the variable `p`. All message channels in *PROMELA* can pass messages on a FIFO basis. In *PROMELA* it is not only possible to pass and retrieve single values to and from message channels, it is also possible to pass and receive multiple values to and from message channels. Examples of this can be seen in Listing 3.2.

If the number of values sent to a message channel is more than the number defined by the message channel's format field then the redundant values are lost. If fewer values are sent, the values of the remaining parameters are undefined. These message channel properties are illustrated in Listing 3.3.

PROMELA message channels have two important properties:

Listing 3.3: Passing multiple values

```
chan sema = [0] of { int, bool };
bool b;
int i, j;

sema!i; /*The value of the boolean field in the channel is undefined*/
sema!i,b,j; /*The value of j is not passed to the channel*/
```

1. The send operation is only executable if the message queue of a channel being addressed is not full.

2. The receive operation is only executable if the message queue of a channel being addressed is not empty.

Evaluating i/o operations

PROMELA only allows the evaluation of expressions that do not have side effects. The following is for instance not valid in *PROMELA*:

```
(a > b && qname?msg0)
```

The reason why the expression is invalid is because it cannot be evaluated without side effects or more to the point, because the send and receive operations are not expressions. Send and receive operations are i/o statements. It is however possible to write the following in *PROMELA*:

```
(a > b && qname?[msg0])
```

In this case the i/o operation is not performed. Instead, only the precondition of the i/o operation is evaluated. The contents of the message channel remain undisturbed.

Message channel behaviour

The *message capacity* field of a message channel determines whether a message channel is synchronous or asynchronous. If the message channel has a capacity of 0, it is synchronous. Any *message capacity* value greater than 0 defines an asynchronous channel. Synchronous and asynchronous message channels behave differently. Consider the *PROMELA* code in Listing 3.4.

In the example, the message channel `name` is declared to have a capacity of 1 and is thus an asynchronous message channel. This means that process A can complete its

execution before process B even starts. The model can have the following sequence of events:

1. Process A completes its first send operation, but blocks on the second send because the channel is full.

2. Process B receives the fist message from the channel ($msgtype, 124$) and then terminates.

3. The channel now has space for process A's second message, so process A becomes executable, performs its second send operation and then terminates leaving the second message as a residual in the message channel.

Listing 3.4: Asynchronous message channel

```
mtype = {msgtype};

chan name = [1] of { mtype, byte };
byte name;
proctype A()
{
        name!msgtype(124);
        name!msgtype(121)
}
proctype B()
{
        byte state;
        name?msgtype(state)
}
init
{ atomic { run A(); run B() }
}
```

The mutual exclusion model in Listing 3.1 is a good example of the behaviour of a synchronous message channel. As mentioned earlier, a message channel with a capacity of zero cannot store any messages; it can only pass it on. This means that the `dijkstra` process can only perform the send operation on line 8 when the `user` process is ready to perform a read operation (line 14). This means that the `sema` message channel synchronizes the `dijkstra` and `user` processes. It is worth noting that only two processes can be synchronised in this manner.

3.3.6 Executability of statements

The definition of the *PROMELA* modelling language centres around its semantics of excitability. It enables the modelling of processes synchronizations. Any statement in a *PROMELA* model is either passable (executable) or blocked. A statement is passable if and only if it evaluates to the boolean value *true*, or equivalently to a non-zero integer value. There are exceptions to the rule. For instance, *PROMELA* semantics state that all assignment statements are passable by default. This rule of possibility has the following consequence: if a process in a *PROMELA*, model reaches a point in its code where it has no executable statements left to execute, it blocks. It also enables some convenient shortcuts when defining *PROMELA* models. Instead of writing a busy-wait loop as follows;

```
while (a != b) /*while  is not a keyword in PROMELA*/
skip /* do nothing, while waiting for a==b */
```

one can use a single *PROMELA* statement, which has the same effect.

```
(a != b) /* block until a == b */
```

It was stated that *PROMELA* statements must always be side effect free. The reason for this becomes clear if one considers the passable nature of *PROMELA* statements. A blocking expression statement for example, may have to be evaluated many times before it becomes executable which can have undesirable side effects.

3.3.7 The `timeout` keyword

PROMELA has statements with predefined meanings. One of the more interesting of these statements is the `timeout` statement. The statement allows a process to abort waiting for a condition that can no longer become true. A good example of such a condition is a process waiting on a read operation from an empty message channel. The `timeout` statement can be seen as a predefined condition that only becomes *true* (passable) if no other statement in a distributed system is executable. The `timeout` statement is most commonly used in repetition and selection structures, such as *while* loops and *if* statements.

The `timeout` statement carries no value. It does not specify a timeout interval or how a timeout is implemented. It only specifies the possibility of a timeout. This is a deliberate abstraction to enable the verification of a modelled system.

3.4 SPIN

The word *SPIN* is an acronym for **S**imple **P**romela **IN**terpreter. *SPIN* is a verification system that can verify the correctness of *PROMELA* models. Like any other verification system, *SPIN*'s goal is: Given a set of system requirements, establish what is possible and what is not. It performs a form of logical verification to accomplish this goal. It checks whether a system is logically correct.

3.4.1 Logical correctness

In the realm of logical verification the focus is whether or not a design requirement could *possibly* be violated. Logical verification is not concerned with the likelihood of any kind of design requirement violations. There is good reason for this: dramatic system failures are almost always the result of design shortcomings in a system caused by overlooking a seemingly unlikely sequence of events. Logical correctness is concerned with possibilities and certainty, not with probabilities.

The restriction of logical correctness to the possible, rather than the probable, has two implications:

- It strengthens the proofs of correctness that can be achieved with system verification. A verification result which states that the violation of a given system requirement is impossible, is much stronger than one that states that there is a low probability that a system will violate one of its requirements.
- It makes it possible to efficiently perform the verification of a system.

The proof of logical correctness properties of a distributed system must be independent of any assumption about the relative speeds of execution of the separate processes in the system. Ideally the verification process of a distributed system cannot make any assumptions about process execution speeds of the operating platform. It is thus not surprising that *PROMELA* and *SPIN* makes it impossible to make any such assumptions of a distributed system being modelled and verified.

3.4.2 Automata theory

Before introducing the type of correctness claims that can be tested in *SPIN*, an introduction to basic automata theory is in order. It was stated earlier that any *PROMELA*

model can be translated into a finite state automata. This property of *PROMELA* models enables *SPIN* to efficiently verify correctness properties of the models.

The modelchecking method used by *SPIN* to verify logical correctness properties of modelled systems is based on a variation of the theory of finite state automata. The variation is known as the ω-automata. An ω-automaton can cover the acceptance conditions of finite and infinite executions. This is an important property seeing that process executions of a distributed system can be finite or infinite. According to [3], a finite state machine is a tuple:

$$FSA =< S, s_0, L, T, F > \ where$$

- S is a finite set of states
- s_0 is a distinguished initial state, $s_0 \in S$
- L is a finite set of labels
- T is a set of transitions, $T \subseteq (S \times L \times S)$
- F is a set of final states, $F \subseteq S$

When discussing automata theory, the following definitions are often used:

Runs

A *run* of a finite state automaton $< S, s_0, L, T, F >$ is an ordered, possibly infinite, sequence of transitions

$$RUN = \{(s_0, l_0, s_1), (s_1, l_1, s_2), \ldots\} \text{ such that } \forall i, (i \geq 0) \rightarrow (s_i, l_i, s_{i+1}) \in T$$

Acceptance

An *accepting run* of a finite state automaton $< S, s_0, L, T, F >$, is a finite run, in which the final transition (s_{n-1}, l_{n-1}, s_n) has the property such that $s_n \in F$. The run is considered *accepted* if and only if it terminates in a final state of the automaton.

Omega acceptance

The definitions above show that a finite state automaton is able to model terminating executions. It is however not possible to decide on acceptance or non-acceptance of infinite executions. An infinite run is called an ω-run. One of the many acceptance definitions for

ω-runs is known as Büchi acceptance. According to Büchi acceptance, an *accepting ω-run* of a finite state automaton $< S, s_0, L, T, F >$ is any infinite run σ such that $\exists s_f, s_f \in F \wedge s_f \in \sigma^\omega$. In the definition:

- ω represents a finite run
- σ^ω represents the set of states that appear infinitely often within ω's set of transitions.

The Büchi acceptance definition states that an infinite run of an automaton is accepted if and only if some state in F is visited infinitely often in the run.

Automaton definitions explained

In order to better understand the automaton definitions mentioned, consider the automaton shown in Figure 3.1. Using the definition of a finite state machine:

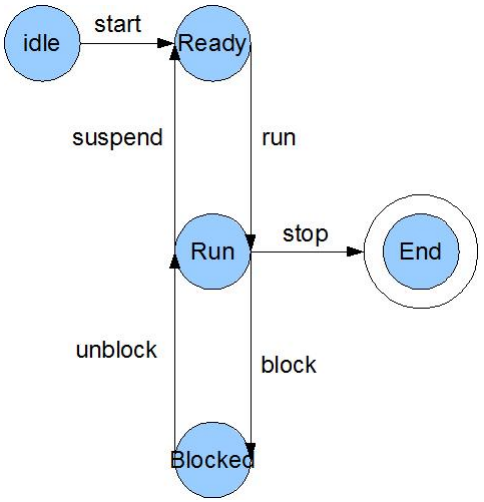


Figure 3.1: A example of a finite state automaton.

- $S = \{Idle, Ready, Run, Blocked, End\}$
- $s_0 = Idle$
- $L = \{start, run, suspend, block, unblock, stop\}$
- $T = \{(Idle, start, Ready), (Ready, run, Run), (Run, suspend, Ready), (Run, block, Blocked), (Blocked, unblock, Run), (Run, stop, End)\}$

- $F = End$

If one interprets the finite state machine in Figure 3.1 as a model for the life of a user process in a time-sharing system, controlled by a scheduler, an example of an accepting run for the finite state machine in figure 3.1 is:

$$(Idle, start, Ready), (Ready, run, Run), (Run, stop, End)$$

Automaton ω-runs can also be visualised. The definition of an ω-run, states that an ω-run consists of a set of transitions that appear infinitely often, and that at least one of these transitions must contain an automaton end state. The only states of the finite state machine in Figure 3.1 that can be visited infinitely often are: Ready,Run,Blocked. None of these states are valid end states of the finite state machine. This means that no ω-run exist for the finite state machine.

3.4.3 Basic types of correctness claims

In distributed system design there are, broadly speaking, two types of correctness properties:

- **Safety properties**: Safety properties state that something bad never happens - that is, that a system never enters an unacceptable state.
- **Liveness properties**: Liveness properties state that something good eventually does happen - that is, that a system eventually enters a desirable state.

A verification system cannot determine what is a *good* or a *bad* property of a system being designed. It can only help a system designer determine what is *possible* and what is not. It is thus up to the system designer to define the set of safety and liveness properties of a system. A verification system such as *SPIN*, can only verify the correctness of these properties. The set of safety and liveness requirements are known as the correctness claims of the system.

All good modelchecking solutions must provide notation to specify the above mentioned correctness claims. To this end, *PROMELA* makes it possible to express the following types of correctness criteria:

- Simple properties like the absence of deadlock in a system.
- Moderate properties like the absence of lifelock in the system.

- Complex properties like inevitably requirements.

The *behaviour* of a validation model is defined by the set of all execution sequences it can perform. Keeping in mind that such a verification model in *PROMELA* is simply an automaton, the behaviour of a model is defined by the set of automaton `runs`. In a *PROMELA* model the states of the underlying automaton is completely defined by the specification of all values for local and global variables, all control flow points of running processes, and the contents of all message channels. A valid execution sequence for a *PROMELA* model $\mathbf{M}$ has the following properties:

- The first state in the sequence is the initial state of $\mathbf{M}$. In this state all variables are initialized to zero, all message channels are empty and the only process active, if it is present, in $\mathbf{M}$ is the `init` processes.
- If $\mathbf{M}$ is placed in the state with ordinal number i, there is at least one executable statement that can bring it to the state with ordinal number $i+1$. This means that if $\mathbf{M}$ is in a state, there is at least one executable statement in $\mathbf{M}$ that will take it to another state.

The *SPIN* verifier uses two types of sequences to validate correctness claims, namely terminating and cyclic sequences. An execution sequence is terminating if no state occurs more than once in the sequence and the model $\mathbf{M}$ contains no executable statements when it is placed in the last state of the sequence. An execution sequence is cyclic if all states except the last one are distinct, and the last state of the sequence is equal to one of the earlier states. This implies that the *SPIN* verifier sees a system as a set of *reachable states*.

PROMELA uses the states of a model and simple propositions to express correctness criteria of a system. A proposition in *PROMELA* is simply a boolean expression. Correctness properties are expressed by explicitly defining in which states of the model certain propositions are required to hold. *PROMELA* provides the following language constructs to define correctness properties:

- basic assertions,
- end-state labels,
- progress-state labels,
- accept-state labels,

Listing 3.5: A simple never claim

```
never { do :: skip :: break od -> P -> !Q }
```

- `Never` claims (Temporal claims), and
- trace assertions

3.4.4 Temporal claims

Temporal claims are very powerful correctness claims. Temporal claims define temporal orderings of properties of states. This makes it possible to define correctness claims like a claim that every state in which the property **P** is true, will be followed by a state in which property **Q** is true. Note that the term *followed by* can be interpreted in two different ways: It might imply that states immediately follow each other, or it might imply that states eventually follow each other. This means that a temporal claim has the ability to loosely model the passing of time.

It was stated earlier, that *PROMELA* defines correctness claims in the form of *impossible* system behaviour. In order to use temporal claims as correctness claims, the correctness cliams must express state orderings that are impossible. It is also worth noting that temporal claims are defined on complete execution sequences (terminating or cyclic). This means that even if a prefix of a sate sequence is not relevant for validating a property, it must still be represented as a trivially true sequence of propositions. If all this is taken into account, the correctness claim, $P \rightarrow Q$ must be expressed as shown in Listing 3.5.

Listing 3.5 expresses that, independent of the initial sequence of events (state sequences), it is impossible for a state in which property **P** is true to be followed by a state in which property **Q** is false. If the claim body terminates, the claim is matched and the property being tested ($P \rightarrow Q$) is thus violated.

PROMELA never claims look similar to the temporal claim example shown in Listing 3.5. There is one difference: *PROMELA* uses progress-state and acceptance-state labels. An acceptance-state label is used to ask the *SPIN* verifier to find all cycles that pass through the labelled acceptance state. The *SPIN* verifier uses the definition of ω-run acceptance to accomplish this task. A progress-state label is used to ask the *SPIN* verifier to verify that every execution (potentially infinite) passes through the labelled acceptance state. A progress-label is thus used to signify that a executing process is making effective

Listing 3.6: A simple *PROMELA* never claim

```
never {
do
          :: skip
          :: P -> break
od;

accept: do
          :: P
od
}
```

progress.

The use of progress-state and acceptance-state labels makes it possible to catch more types of errors than just a complete match of completing behaviour (see Listing 3.5). In the case of *PROMELA*, it makes it possible to express never claims in the form of a special finite state automaton that cycles through an acceptance state, if the claim is violated (an undesirable behaviour is recognized).

The automaton nature of *PROMELA* never claims can be shown by making use of an example. The never claim depicted in Listing 3.6 corresponds to the following temporal claim: **P** can never remain true infinitely long.

From the presence of the *do construct* in the never claim, it can be seen that the never claim translated into a non-terminating automaton. This means that the claim is matched and the claim is violated if and when an acceptance cycle is detected by the *SPIN* verifier. The `skip` statement in the first loop is always executable. It is worth noting that sequences where **P** changes from true to false a few times are permitted.

3.4.5 Verifying a temporal claim

In the previous section the *PROMELA* never claim was introduced. It was stated that such a never claim is a representation of a temporal logic claim. It was not explained how the never claim is used by the *SPIN* verifier to verify (prove) the claim. This will now be done by using an example. Note that the discussion in this section does not go into much detail. It does not, for instance, explain asynchronous and synchronous automata products. The discussion is intended to be overview of how model verification is done. A more detailed explanation of the concepts can be found in [3].

Consider the never claim and *PROMELA* model depicted in Listing 3.7. According to

Listing 3.7: Example *PROMELA* model and never claim

```
#define p (x<4)
int x=4;
proctype A()
{
        do
        ::x%2 -> x = 3*x+1
        od
}


protype B()
{
        do
        ::!(x%2) -> x = x/2
        od
}

init
{
        atomic{run A(); run B()}
}

never{  /*<> []p */
T0_init:
        if
        :: p -> goto accept_S4
        :: true -> goto T0_init
        fi;
accept_S4:
        if
        :: p -> goto accept_S4
        fi;
}
```

the temporal claim (<> []p) in the model, *SPIN* must verify that the following proposition ($x < 4$) is eventually always true. *SPIN* starts by translating each process and the never claim in the model into finite state automata. The automata for `process A`, `process B` and the never claim is depicted in Figure 3.2. The automaton of the `init` process is not shown. It is also worth noting that both `process A` and `process B` have infinite execution cycles. This can also be deduced by studying figure 3.2, seeing that the automaton of each process does not have an end state.

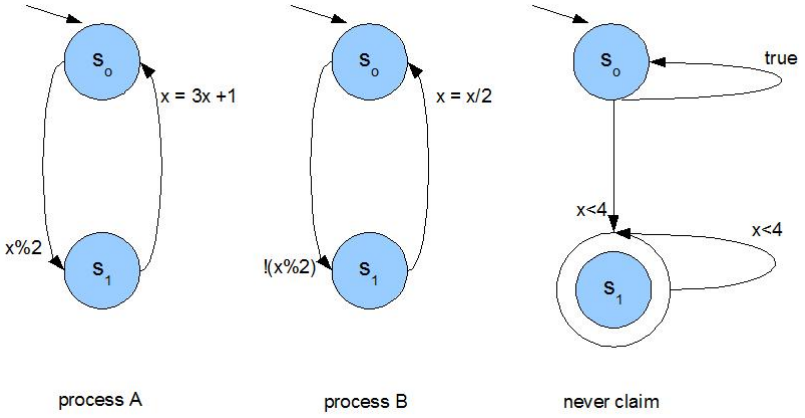


Figure 3.2: FSM's of PROMELA model depicted in Listing 3.7.

SPIN then uses automata theory to compute the asynchronous product of the automata of processes A and B. The asynchronous product of the two automata is shown in Figure 3.3. The states in the figure marked with a double `p,v`, where `p`, references the states of `process A` and `v` references the states of `process B`. *SPIN* uses the asynchronous product of the automata to model the concurrent nature of the system.

In the *PROMELA* model shown in Listing 3.7, the variable x can have three distinct values. The automaton in Figure 3.3 must be expanded to represent this fact. The automaton can also be simplified by noting that there is no feasible way to reach state $(s_1, s1)$ for the initial state, (s_0, s_0), of the asynchronous product. To reach state (s_1, s_1) from state, (s_0, s_0) would require both the condition $(x\%2)$ and its negation to evaluate to *true* without an intervening change in the value of x.

The simplified, expanded version of the automaton depicted in Figure 3.3 is shown in figure 3.4. The states in Figure 3.4 are marked with a triple `p,v,q`, where `p` references the states of `process A`, `v` references the states of `process B` and `q` shows the value of x in each state of the automaton.

SPIN now computes the expanded, synchronous product of the automaton in Figure 3.4 and the automaton of the never claim. This step is performed in order for *SPIN* to perform the verification of the model using automata theory. The resulting automaton is shown in Figure 3.5.

The states in Figure 3.4 are marked with a tuple `p,v,q,r`, where `p` references the states of `process A`, `v` references the states of `process B`, `q` shows the value of x in each state and `r` references the states of the *never claim* automaton. *SPIN* now uses the definition of an ω-run acceptance in order to verify the temporal claim. According to the definition of an ω-run acceptance, it must look for a set of infinitely repeating transitions which contains at least one automaton end-state. In order to do this, *SPIN* must traverse every possible execution sequence of the automaton. In the example model, there are four acceptance states, but none of them form part of a cyclic execution sequence. The temporal claim is thus violated.

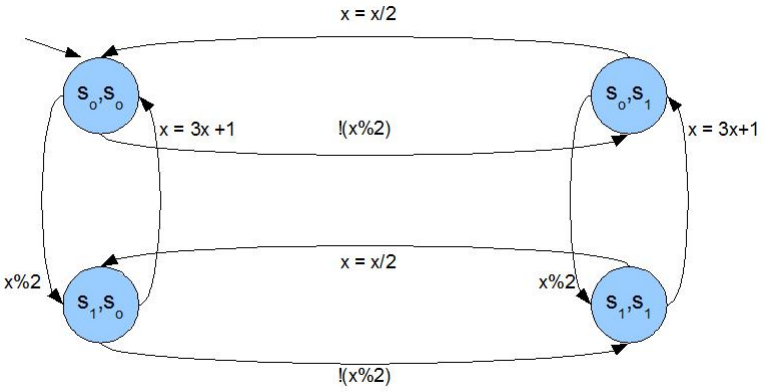


Figure 3.3: Asynchronous product of automata A and B, depicted in Figure 3.2.

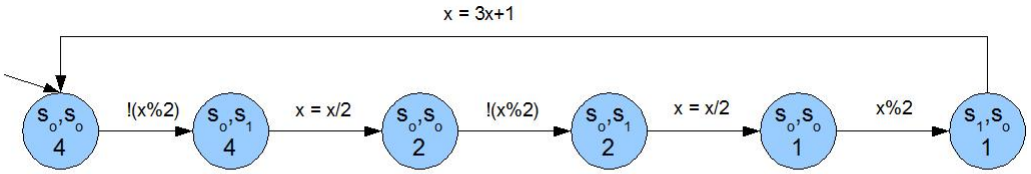


Figure 3.4: Expanded and simplified version of the automaton depicted in Figure 3.3.

3.5 The importance of abstraction

In the verification example discussed in the previous section, it was stated that *SPIN* computes automata products as part of the verification process. To compute these automata products use memory. This statement has two important consequences:

1. The bigger and more complex the model, the more memory is needed to verify the model.

2. The bigger and more complex the model, the more execution sequences need to be traversed, and the more processing time is needed to verify the model.

The bigger and complex the model, the more states are needed to represent the model. The collection of states that represents the model is known as the *state space* of the model. In order to make modelchecking a feasible part of a system design, one needs to keep the *state space* of the model as small as possible. Abstraction plays a large role in achieving this goal. It is, for instance, not a good idea to model a timestamp using a random integer value. A integer can have a range of $-2^{31}..2^{31}-1$. In order to verify a model using such an integer variable, *SPIN* must take into account all the possible interactions between the variable and its different values, and the rest of the modelled system. This causes a *state explosion* and *SPIN* will not be able to verify the model. The importance of the use of abstraction when building models, can thus not be overstated.

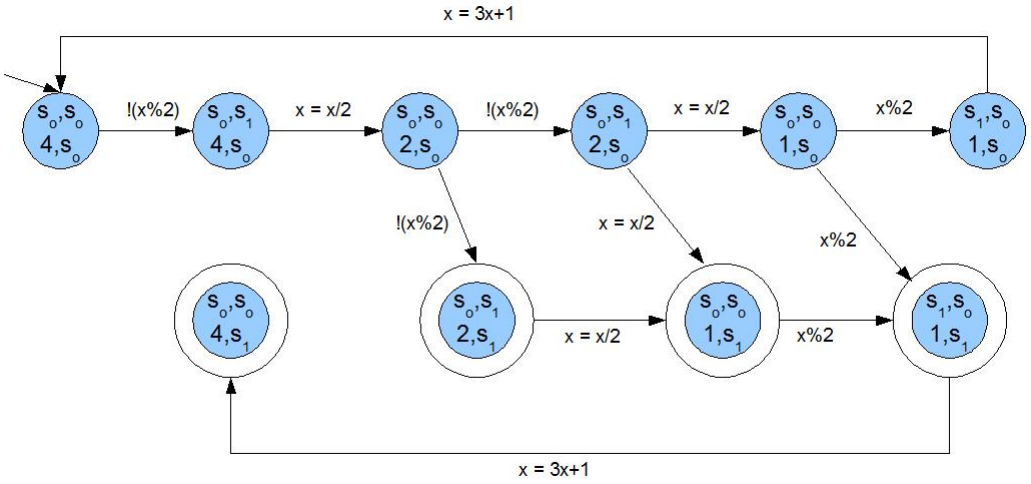


Figure 3.5: The final automaton used by *SPIN*.

3.6 Summary

This chapter introduced two important modelchecking tools. In the introduction of the chapter the importance of modelchecking in the design of distributed and concurrent systems is discussed. The important aspects of the *PROMELA* modelling language and the *SPIN* verification system is then discussed. The chapter ends with an example showing how a concurrent process model is verified by the *SPIN* verifier. The importance of the use of abstraction in system modelling is also discussed.

Chapter 4

Modelling the request-processor system

The design of the request-processor system as described in chapter 2 looks promising, but a promising design is not of much use if it cannot be proven to be logically correct. Modelchecking has a perfect set of tools which makes it possible to test the logical correctness of the multiple process design of the request-processor system. In order to use the modelchecking tools to check the logical correctness of the request-processor system, a *PROMELA* model of the system needs to be built. This chapter describes how the request-processor was modelled.

4.1 State-based process modelling

All the process models of the request-processor system need to be well structured. Well structured process models make it easier to understand how the different processes of the request-processor system interact with each other. This in turn makes it easier to isolate the cause of errors in the request-processor system design.

All the process models of the request-processor system were based on a transition system. A transition system is a form of a finite state automaton. According to [6] a transition system is a quadruple

$$A =< S, T, \alpha, \beta > \text{ where}$$

- S is a finite set of states
- T is a finite set of transitions

Listing 4.1: *PROMELA* model of a client

```
proctype client(chan clientRHOut,out)
{
  do
  ::true->clientRHOut!clientRequest(RAWImageRequest);
  ::true->clientRHOut!clientRequest(processedImageRequest);
  ::true->clientRHOut!clientRequest(scheduleImageRequest);
  od;
}
```

- α and β are two mappings from T to S which takes each transition t in T to the two states $\alpha(t)$ and $\beta(t)$, respectively the source and the target of transition t.

A transition modifies the current state of a transition system. A transition is normally associated with a guard statement which needs to be true in order for the transition to be active. A transition can also have actions associated with it which needs to be preformed before the system changes state. The transition system state property is thus ideal for providing the needed structure to the process models of the request-processor system.

4.2 The client model

The client model represents any kind of user process that needs to send a request to the request-processor system. According to the request-processor system design, a client can issue:

- a *RAW*-image request,
- a process-image request; or
- a schedule request to the request-processor system.

At modelling level we are not interested in the structure of these requests, nor are we interested in how the requests are sent to the request-processor system. We are only interested in the fact that three possible types of requests can be sent by a client, and accepted by the request-processor system. All the non-essential implementation detail such as the request detail is abstracted in the model. The model of the client is shown in Listing 4.1. It is worth noting that the client only sends requests in the model.

In the client model in Listing 4.1, a non-deterministic choice is made to send a *RAW*-image request, a process-image request or a schedule-image request to the request-handler

Listing 4.2: Using an array as a queue

```
#define LEN 5;
int queue[LEN];
int firstMessage;

for (int i =0; i<LEN;i++)
{
  queue[i] = i+1;
}
firstMessage = queue[0];
```

process of the request-processor system. This is a good example of how abstraction can be used when modelling the system. It is not important *how* the client makes the choice of which type of request to send, but rather that the choice is made at some stage during the client's execution cycle. It is worth noting that the client model does not wait for a response from the system. The response from the system does not need to be part of the client model, because it is not the responsibility of the request-handler process to send the response to the client. A system resource (the sender-process) is responsible for sending a response to the client. All that needs to be modelled is that the request-processor system sends the response to the appropriate system resource.

4.3 Modelling message queues

The most important thing that needs to be described in the request-processor system model is how the system utilizes system resources in order to service client requests. This resource utilization is done synchronously. The synchronization is achieved by using interprocess communication based on a token passing scheme and message queues. The interprocess communication scheme used by the request-processor system is described in chapter 2 of this thesis. The message queues must be part of the request-processor system model.

One way to model a queue is to use a one dimensional array and an index counter as shown in Listing 4.2. This kind of queue-modelling is however not ideal when working with verification models. Comparison operations such as $<$ and $==$, which might be used in such queue models, are often the cause of *state space* explosions. State-space explosions can make it impossible to verify a model.

A better approach is to limit the amount of elements in the queue and to explicitly

Listing 4.3: The clientRequestQueue model

```
#define ENQUEUEMES(x)\
d_step{\
  if\
  ::clientRequestsToRH[0]==EMPTY->\
    clientRequestsToRH[0]=x;\
  ::else->\
    if\
    ::clientRequestsToRH[1]==EMPTY->\
      clientRequestsToRH[1]=x;\
    ::else->\
      clientRequestsToRH[2]=x;\
    fi;\
  fi;\
}
```

model every possible insert and remove operation. As an example, consider the model in Listing 4.3. The *PROMELA* macro describes all the insert operations of the client-request queue that was used in the request-processor system model. It is worth noting that the queue size is fixed. The bigger the queue size, the more insert and remove operations need to be modelled.

The final request-processor system required seven message queues to be modelled. Most of these queues were used as message buffers for interprocess communication. The request-handler process for example required three queues:

1. A queue to buffer the client requests that it needs to send to the request-dispatcher process.

2. A queue to buffer the acknowledgements of requests it receives from the request-dispatcher process.

3. A queue to buffer the request-responses that needs to be send back to client processes.

4.4 The request-handler model

The request-handler process must receive client requests and ensure that it gets sent to the request-dispatcher process for processing. It must also receive request-dispatcher

responses and send them on to the sender process if needed. The model of the request-handler process will be explained by making use of its transition system. The transition system can be seen in Figure 4.1. Before describing what happens in each state of the request-handler transition system, it is worth mentioning that the request-handler maintains three internal queues:

1. *clientRequestsToRH*: This *FIFO*-queue contains client requests that the request-handler process receives from clients. The requests in the queue are also the requests that the request-handler process must send to the request-dispatcher process for processing.

2. *clientRequestsAc*: This queue contains the requests that have been acknowledged by the request-dispatcher process. The requests in the queue all have a timestamp. The timestamps are used to remove requests from the queue.

3. *senderRequests*: This *FIFO*-queue contains requests that have been processed by the request-dispatcher and that need to be sent back to the client processes.

Each state of the request-handler transition system which depicted in Figure 4.1, will now be discussed.

State 1

This is the startup state of the request-handler. In this state, the request-handler can only receive a client request or a token from the request-dispatcher. When the request-handler enters the state, it checks to see whether or not it can receive a processing request from a client. It accomplishes this by inspecting the *clientRequestsToRH*-queue. If the queue is full, it cannot receive a client request. If it however finds that the queue is not full, it waits a while for a possible client request. If it receives a request from a client, it inserts the request into the *clientRequestsToRH*-queue.

If the request-handler receives an *init*-token from the request-dispatcher, it checks to see whether or not the *clientRequestsToRH*-queue is empty. If the queue is not empty, the request-handler places the first request in the queue, on the token before sending it back to the request-dispatcher. If the *clientRequestsToRH*-queue is empty, the request-handler sends back an empty token to the request-dispatcher.

If the request-handler receives a *normal* token from the request-dispatcher, it investigates the token to determine whether or not the request-dispatcher has completed processing a request and whether or not the request-dispatcher has acknowledged a request.

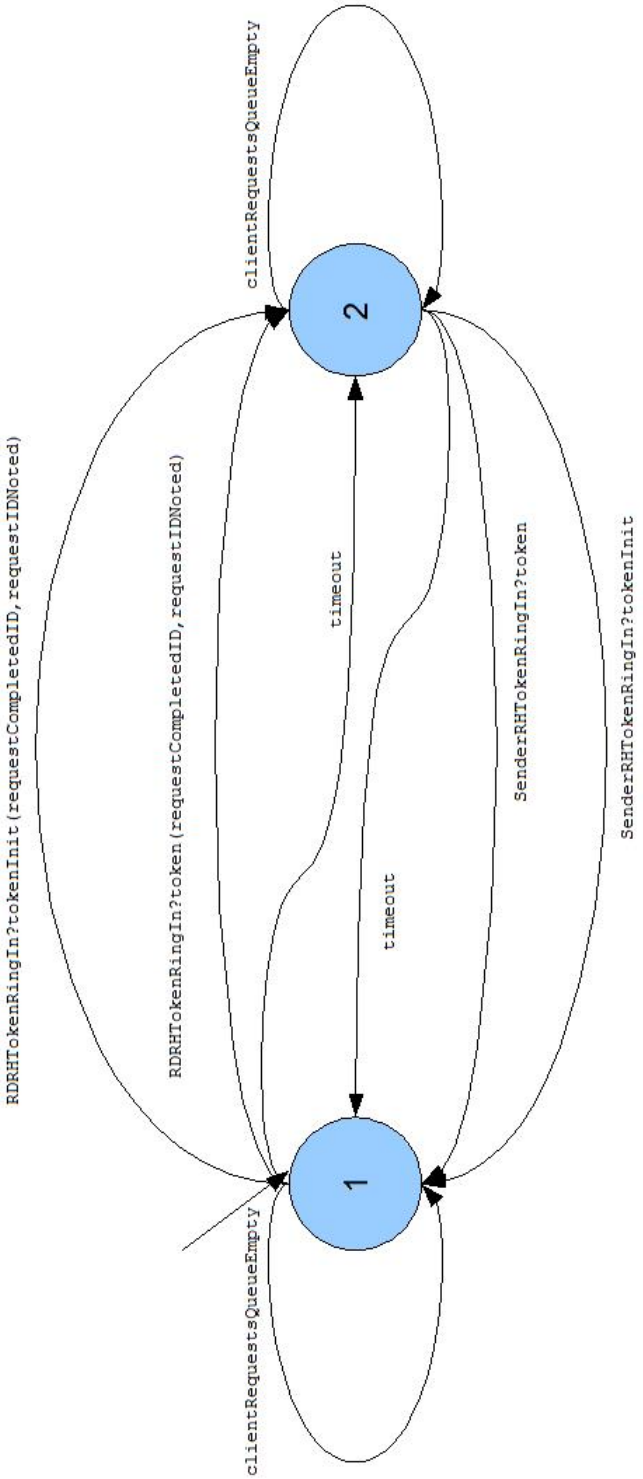


Figure 4.1: Transition system of the request-handler process.

If the request-handler finds that the request-dispatcher has indeed acknowledged a request, it removes the request from the *clientRequestsToRH*-queue and inserts it into the *clientRequestsAc*-queue. If the request-handler finds that the request-dispatcher has completed a request, it removes the request from the *clientRequestsAc*-queue and places it in the *senderRequests*-queue. The completed-request received from the request-dispatcher contains all the processed data and responses that needs to be sent back to the client that made the initial request.

After processing the token it received from the request-dispatcher, the request-handler checks to see whether or not the *clientRequestsToRH*-queue is empty. If the queue is not empty, the request-handler places the first request in the queue on the token before sending it back to the request-dispatcher. If the *clientRequestsToRH*-queue is empty, the request-handler sends back an empty token to the request-dispatcher.

State 2

In this state, the request-handler can only receive a client request or a token from the sender process. When the request-handler enters the state, it checks to see if it can accept a client request. It does this in exactly the same way as in state 1.

If the request-handler receives an *init*-token from the sender process, it knows that the sender process is ready to receive and process send-requests. If the request-handler finds that the *senderRequests*-queue is not empty, it places the first request in the queue onto the token and sends it back to the sender-process. If the request-handler however finds that the *senderRequests*-queue is empty, it keeps the token. The request-handler will keep the token during its execution lifetime until it has a request that it needs to send to the sender process.

If the request-handler receives a *normal* token from the sender process, it removes the first request from the *senderRequests*-queue. The request-handler assumes that the last request it sent to the sender has been processed, because the sender process will only send back the token when it has completed processing of a request, and because the request-handler can only send one request to the sender processes at any given time. After removing the request from the *senderRequests*-queue, it checks to see if it needs to send another request to the sender process by checking whether or not the *senderRequests*-queue is empty. If the queue is not empty, it places the first request in the queue onto the token and sends it back to the sender process. If the queue is empty, the request-handler keeps the token until it is needed.

4.5 The request-dispatcher model

The request-dispatcher is the most important process of the request-processor system. It must process client requests using the needed system resources. In order to achieve this, it must me able to communicate with the request-handler process, the memory-controller process, the scheduler-dispatcher process and the image-processor process. This in turn causes the request-handler to maintain four request queues:

1. *internalRDResponseQueue*: This queue is a *FIFO*-queue. The queue contains completed requests and their responses which need to be sent back to the request-handler process.

2. *internalRDscheduleRequestQueue*: This queue is a *FIFO*-queue which contains requests that the request-dispatcher needs to send to the scheduler-dispatcher process.

3. *internalRDRAWImageRequestQueue*: This queue is a *FIFO*-queue which contains requests that the request-dispatcher needs to send to the memory-controller process.

4. *internalRDIPRequestQueue*: This queue is a *FIFO*-queue which contains requests that the request-dispatcher needs to send to the image-processor process.

The model of the request-dispatcher will now be explained by using its transition system which is depicted in Figure 4.2.

State 1

State 1 is the startup state of the request-dispatcher. In this state, the request-dispatcher sends an *init*-token to the request-handler. The request-dispatcher is a process that provides a service to the request-handler. In the request-processor system design, all service-providing processes are responsible for letting the rest of the system know they are ready to receive requests for processing.

State 2

In this state, the request-dispatcher can only receive a token from the request-handler. When the request-dispatcher receives a token in this state, it checks to see what type of request (schedule, raw-image, image-processing or *NONE*) was received.

If a raw-image request was received, the request-dispatcher verifies whether or not it is able to accept the request. It does this by checking whether or not there is any

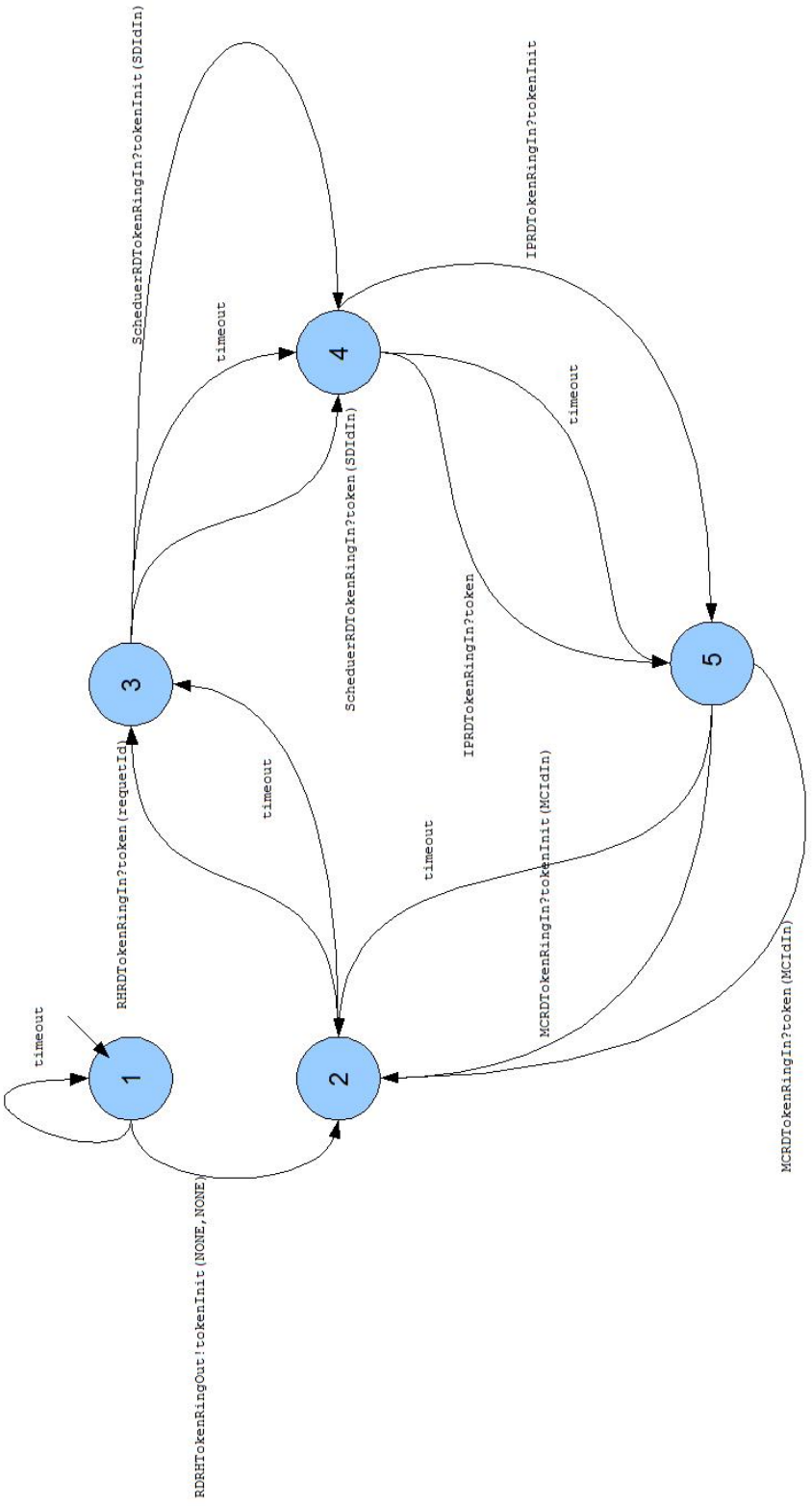


Figure 4.2: Transition system of the request-dispatcher process.

raw-image request already in the system. If there is, the request is rejected and will not be acknowledged. If the request-dispatcher finds that it is able to accept the request, it inserts the request into the *internalRDRAWImageRequestQueue*-queue. It also notes that the request was accepted.

The request-dispatcher now investigates the *internalRDResponseQueue*-queue. If there is a request in the queue, it gets noted as a response request and is removed from the queue. The noted accept request and the response request are then placed onto the token and is sent back to the request-handler. It is possible that no request was accepted and that there is no response request to send back to the request-handler. In this case the space allocated on the token, for the requests are marked as *empty* before the token is sent back to the request-handler.

The processing sequence for schedule and image-processing requests are almost exactly the same as that of a raw-image request. The only difference being the resource queue which gets updated. In the case of receiving a raw-image request, the request-dispatcher updates the *internalRDRAWImageRequestQueue*-queue. In the case receiving a schedule or image-processing request, the request-dispatcher updates either the *internalRDscheduleRequestQueue*-queue or the *internalRDIPRequestQueue*-queue.

State 3

In this state, the request-dispatcher can only receive a token for the scheduler-dispatcher process. This token can either be a *normal* or an *init* token.

When the request-dispatcher receives an *init* token from the scheduler-dispatcher process, it checks whether or not the *internalRDscheduleRequestQueue*-queue is empty. If the queue is not empty, the request-dispatcher places the first request in the queue onto the token and sends it back to the scheduler-dispatcher process. The request-dispatcher, however, keeps the token if it finds that the queue is empty.

After receiving a *normal* token from the scheduler-dispatcher process, the request-dispatcher removes the first request form the *internalRDscheduleRequestQueue*-queue and inserts the request into the *internalRDResponseQueue*-queue. The request-dispatcher assumes that scheduler-dispatcher process has processed the first request in the *internalRDscheduleRequestQueue*-queue, because according to the request-processor system design, the scheduler-dispatcher process will only send back a *normal* token, if it is done processing a request. This assumption is always made when the request-dispatcher communicates with a system resource process.

If the request-dispatcher enters this state and finds that it is in possession of the token that is used for communication between the scheduler-dispatcher and itself, it checks to see whether the *internalRDscheduleRequestQueue*-queue is empty. If the queue is not empty, the request-dispatcher places the first request in the queue on the token and sends it to the scheduler-dispatcher process.

State 4

In this state, the request-dispatcher can only receive a token for the image-processor process. As was the case with the scheduler-dispatcher token, the token can either be a *normal* or an *init* token.

If the request-dispatcher receives an *init* token from the image-processor process, it checks the *internalRDIPRequestQueue*-queue to see if it needs to send a request to the image-processor process. If the queue is empty, the request-dispatcher keeps the token; else it places the first request in the queue onto the token and sends it to the image-processor process.

When the request-dispatcher receives a *normal* token from the image-processor process, it removes the first request from the *internalRDIPRequestQueue*-queue and then inserts the request into the *internalRDResponseQueue*-queue.

If the request-dispatcher enters this state, and it finds that it is in possession of the needed communication token, it checks to see whether the *internalRDIPRequestQueue*-queue is empty. If the queue is not empty, the request-dispatcher places the first request in the queue, on the token and sends it to the image-processor process.

State 5

The request-dispatcher can only respond to the memory-controller process in this state. The request-dispatcher can once again, only receive two types of tokens: an*init* token or a normal token.

If the request-dispatcher receives an *init* token form the memory-controller process, it checks the *internalRDRAWImageRequestQueue*-queue to see if it needs to send a request to the memory-controller process. If the queue is empty, the request-dispatcher keeps the token; else it places the first request in the *internalRDRAWImageRequestQueue*-queue on the token and sends it to the memory-controller process.

If the request-dispatcher receives a *normal* token from the memory-controller process, it once again assumes that the last request that was send to the memory-controller process

has been successfully processed. The request is thus removed from the *internalRDRAW-ImageRequestQueue*-queue.

A response from the memory-controller process can tell the request-dispatcher whether the raw-image it requested was found or not. The request-dispatcher responds differently to each kind of response. If the memory-controller indicates that it has found a requested raw-image, the request-dispatcher examines the response. The original raw-image request could have either been made in order to satisfy a client raw-image request or a client image-processing request. In the case of a raw-image request, the request-dispatcher inserts the memory-controller process response into the *internalRDResponseQueue*-queue. In the case of the client image-processing request, the request-dispatcher uses the response it received from the memory-controller process to construct an image-processor request which it then inserts into the *internalRDIPRequestQueue*-queue.

If the response from the memory-controller process indicates that it did not find the requested raw-image, the request-dispatcher also examines the memory-controller process response to see if the original client request indicates if an image acquirement request must be added to the satellite imaging schedule, if the raw-image request cannot be found. If this is the case, the request-dispatcher constructs a schedule request and inserts it into the *internalRDscheduleRequestQueue*-queue; else the memory-controller response gets added to the *internalRDResponseQueue*-queue.

After processing the memory-controller token; the request-dispatcher checks to see whether or not it needs to send a request to the memory-controller process. If it finds that the *internalRDRAWImageRequestQueue*-queue is not empty, it places the first request onto the token and sends it to the memory-controller process, else it keeps the token until it is needed.

If the request-dispatcher enters this state and it finds that it is in possession of the communication token, it checks to see whether the *internalRDRAWImageRequestQueue*-queue is empty. If the queue is not empty, the request-dispatcher places the first request in the queue onto the token and sends it to the memory-controller process.

4.6 Modelling the system resources

In order to successfully process a client request, the request-dispatcher needs access to certain system resources. In the request-processor system design, these resources are controlled by separate processes. The processes are:

Listing 4.4: The scheduler-dispatcher process model

```
proctype schedulerDispatcher(chan TokenRingOut, TokenRingIn)
{
  int idIn;
  int lp;
  lp=1;

  do
  ::lp==1->TokenRingOut!tokenInit(NONE); lp=2;
  ::TokenRingIn?token(idIn)->TokenRingOut!token(idIn);
  od;
}
```

- **Memory-controller**: Processes any requests that relates to mass memory access.
- **Scheduler-dispatcher**: Processes any requests that relates to the imaging-schedule of the satellite.
- **Image-processor**: Processes any requests that relates to the image-processing subsystem.

The request-handler also uses a system resource to send back processed requests to clients. The system resource is the satellite's communication payload. The control software of this resource is represented in the request-processor system by the sender process.

Modelling the system resources is simple: a system resource can receive process and send back responses for requests. The basic model for the scheduler-dispatcher process is listed in Listing 4.4. All the other system resource process models are similar to that of the scheduler-dispatcher process. The transition system of a system resource process can be seen in Figure 4.3. The transition system depicted in Figure 4.3 will now be explained.

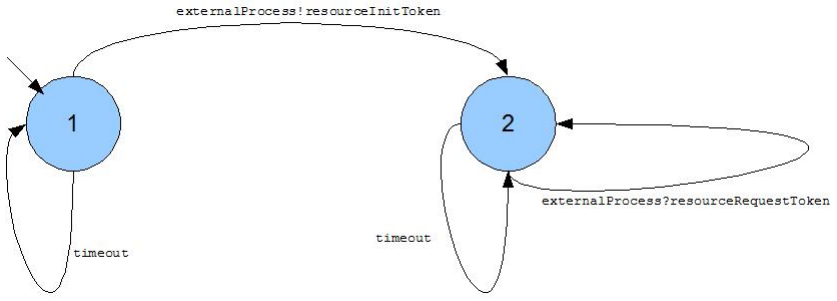


Figure 4.3: Transition system of a system resource.

State 1

This is the startup state of the system resource process. In this state the process sends an *init*-token to the request-dispatcher to let the request-dispatcher know that the system resource process is ready to process requests.

State 2

In this state the system resource process can only receive one request at a time from the request-dispatcher. After receiving the request, the system resource processes the request before sending back a response to the request-dispatcher.

4.7 Extending the models

The final section of this chapter describes how to extend the processes of the request-processor system. In order to ensure processing *fairness*, all the processes that form part of the request-processor system were designed as transition systems. In each state of the request-dispatcher, for instance, it can only process requests from one specific process at a time.

When extending the processes, this transition system design must always be upheld to ensure processing fairness amongst the processes of the request-processor system. Suppose for instance that the request-dispatcher needs to be extended to enable communication with a *configure* process. The *configure* process can for argument sake send periodic configuration updates to the request-dispatcher process. The request-dispatcher process must process and acknowledge the configuration updates.

To understand how the request-dispatcher processes design needs to be modified, consider its current transition system design. The transition system of the request-dispatcher consists of five different states. The states are shown in Listing 4.5. Listing 4.5 can be summarized as follow:

- **State1**: Send *init*-token to needed listening processes.
- **State2**: Accept, process and respond to requests originating from the request-handler process.
- **State3**: Accept, process and respond to requests originating from the scheduler-dispatcher process.

Listing 4.5: Simplified *PROMELA* model of the request-dispatcher transition system

```
/*A high level model of the request dispatcher,
depicting its state-based nature*/
proctype requestDispascher()
do
  if
  /*This process is a service provider,
    it must send the initial token in state 1*/
  ::lp==1->/*send init token request handler*/
      RDRHTokenRingOut!tokenInit(NONE,NONE);
      lp=2;
  ::lp==2->
      /*In this state only service and accept requests
      from the request handler*/
      lp=3;
  ::lp==3->
      /*In this state only service and accept requests
      from the scheduler dispatcher*/
      lp=4;
  ::lp==4->
      /*In this state only service and accept requests
      from the image processor*/
      lp=5;
  ::lp==5->
      /*In this state only service and accept requests
      from the memory controller*/
      lp=2;
  fi;
od;
```

- **State4**: Accept, process and respond to requests originating from the image-processor process.
- **State5**: Accept, process and respond to requests originating from the memory-controller process.

From a design point of view, the *configure* process is one more process that requires request processing from the request-dispatcher process. This means that the request-dispatcher transition system must be extended by adding a state. In the newly added state, the request-dispatcher can only accept, process and respond to requests originating from the *configure* process. The updated transition system of the request-dispatcher processes can be seen in Listing 4.6. Note the newly added state (lines 26 - 30).

Listing 4.6: Updated *PROMELA* model of the request-dispatcher transition system

```
1  /*A high level model of the request dispatcher,
2  depicting its state-based nature*/
3  proctype requestDispascher()
4  do
5     if
6     /*This process is a service provider,
7       it must send the initial token in state 1*/
8     ::lp==1->/*send init token request handler*/
9         RDRHTokenRingOut!tokenInit(NONE,NONE);
10        lp=2;
11    ::lp==2->
12        /*In this state only service and accept requests
13        from the request handler*/
14        lp=3;
15    ::lp==3->
16        /*In this state only service and accept requests
17        from the scheduler dispatcher*/
18        lp=4;
19    ::lp==4->
20        /*In this state only service and accept requests
21        from the image processor*/
22        lp=5;
23    ::lp==5->
24        /*In this state only service and accept requests
25        from the memory controller*/
26        lp=6;
27    ::lp==6->
28        /*In this state only service and accept requests
29        from the configure process*/
30        lp=2;
31    fi;
32 od;
```

Listing 4.5 can be summarized as follows

- **State1**: Send *init*-token to needed listening processes.
- **State2**: Accept, process and respond to requests originating from the request-handler process.
- **State3**: Accept, process and respond to requests originating from the scheduler-dispatcher process.
- **State4**: Accept, process and respond to requests originating from the image-processor process.
- **State5**: Accept, process and respond to requests originating from the memory-controller process.
- **State6**: Accept, process and respond to requests originating from the *configure* process.

It is worth noting that the order of the states of the request-dispatcher process is not important. The only important thing is that there exists separate states in which the request-dispatcher communicates with all the different processes in the request-processor system.

4.8 Summary

This chapter described how the request-processor system design, described in chapter 2, was modelled in *PROMELA*. The chapter started by introducing the concept of a transition system. Next, the transition system based models of the request-processor processes were discussed. The chapter concluded by explaining how the transition system based design of request-processor processes, makes them easily extendible.

Chapter 5

Verifying the system design

In order for the model of the request-processor system developed in chapter 4 of this thesis to be of any use, it must be verified that the model satisfies certain logical correctness properties. In order to verify the request-processor system some of the basic system requirements needed to be translated into LTL-formulas. These formulas were then used together with the *SPIN* verifier to verify the logical correctness of the requests-processor model. All the correctness properties and their results are discussed in this chapter.

5.1 Additions made to the system model

It was mentioned earlier in this thesis that *PROMULA* and *SPIN* uses propositions to verify correctness properties of a system. In order to specify the propositions needed to verify the request-processor model, global boolean variables needed to be added. The needed additions are shown in Listing 5.1.

The boolean variables, *clientRequestQueued* and *clientRequestDequeued*, were used to keep track of the client requests entering and exiting the system. The system design dictates that a client request can only leave the request-processor system if it has been processed or rejected. In the system design, a client request is considered to be processed if a response for the request is delivered to the sender process of the request-processor system.

The following boolean variables were used to keep track of specific types of requests in the system:

- *scheduleImageRequestMade*, *scheduleImageRequestDone*: These boolean variables kept track of schedule requests that entered the system.

Listing 5.1: Boolean variables used for verification

```
bool clientRequestQueued=false;
bool clientRequestDequeued=false;

bool scheduleImageRequestMade=false;
bool scheduleImageRequestDone=false;
bool RAWImageRequestMade=false;
bool RAWImageRequestDone=false;
bool processedImageRequestMade=false;
bool processedImageRequestDone=false;
bool senderRequestMade=false;
bool senderRequestDone=false;
```

- *RAWImageRequestMade*, *RAWImageRequestDone*: These boolean variables kept track of raw-image requests that entered the system.
- *processedImageRequestMade*, *processedImageRequestDone*: These boolean variables kept track of image-processing requests that entered the system.
- *senderRequestMade*, *senderRequestDone*: These boolean variables kept track of requests that the request-handler sent to the sender process.

5.2 Correctness properties

The additions made to the model made it possible to formalize some basic system properties in the form of *LTL*-formulas. The *LTL*-formulas were used by *SPIN* to verify the system. The following system properties were verified:

Safety properties

S1: The system is deadlock free. This critical system property is automatically verified by *SPIN*. It was thus not needed to specify a *LTL*-formula to test whether or not the system is deadlock free.

Liveness properties

L1: Every request made by a client process and which is accepted by the request-processor system, eventually gets serviced.

$$[](clientRequestMade_i \Rightarrow <> cleintRequestServiced_i)$$

L2: Every raw-image request is eventually processed.

$$[](RAWImageRequestMade_i \Rightarrow <> RAWImageRequestDone_i)$$

L3: Every schedule request is eventually processed.

$$[](scheduleRequestMade_i \Rightarrow <> scheduleRequestDone_i)$$

L4: Every process-image request is eventually processed.

$$[](processImageRequestMade_i \Rightarrow <> processImageRequestDone_i)$$

L5: Every sender request that the system generates, is eventually processed by the sender process.

$$[](senderRequestMade_i \Rightarrow <> senderRequestDone_i)$$

5.3 Additional correctnesses properties

The presence of the verification boolean variables in the model makes it possible to check interesting correctness properties of the designed system. It is for instance possible to check that an image-processing request follows the correct processing path trough the system, namely $requestHandler \rightarrow requestDispatcher \rightarrow memoryController \rightarrow requestDispatcher \rightarrow imageProcessor \rightarrow requestDispatcher \rightarrow requestHandler$.

These kind of properties were verified for simpler versions of the request-processor system model. Due to memory restrictions these kind of properties could not be verified for the complete request-processor system model and was thus excluded from the final set of correctness properties that were verified.

5.4 How to verify correctness properties using *SPIN*

How does one verify correctness properties using the *SPIN* verifier? This section answers this question by making use of an example and a small, useful program called *XSPIN*. *XSPIN* is a GUI (graphical user interface) for the *SPIN* verifier.

Suppose the following correctness property needs to be verified:

Every raw-image request is eventually processed.

$$[](RAWImageRequestMade_i \Rightarrow <> RAWImageRequestDone_i)$$

The verification of the above mentioned correctness property is a three step process. Each step will now be discussed in turn.

5.4.1 Specify the LTL formula

In order to verify the property, the LTL formula for the property must be formatted so that it can be used by the *SPIN* verifier. The needed formatting is done using the LTL manager depicted in Figure 5.1.

The main window of the LTL manager consists of the following sections:

- **Formula**: The formula section enables the user to enter an LTL formatted correctness property.
- **Notes**: The notes section summarizes the LTL formula that was entered in the formula section. In the case of figure 5.1, it states that the formula entered, states that if the expression p is true in at least one state, then sometime thereafter, the expression q, must also become true at least once. The information in the Notes section is not always very accurate. For the property being tested in Figure 5.1 for example, the fact that the property *must always* hold, was omitted.
- **Symbol definitions**: In this section, the boolean expressions that are associated with the symbols in the LTL formula are defined. In the case of Figure 5.1, the associations are:
  1. *Symbol p*: (RAWImageRequestMade == true)
  2. *Symbol q*: (RAWImageRequestDone == true)

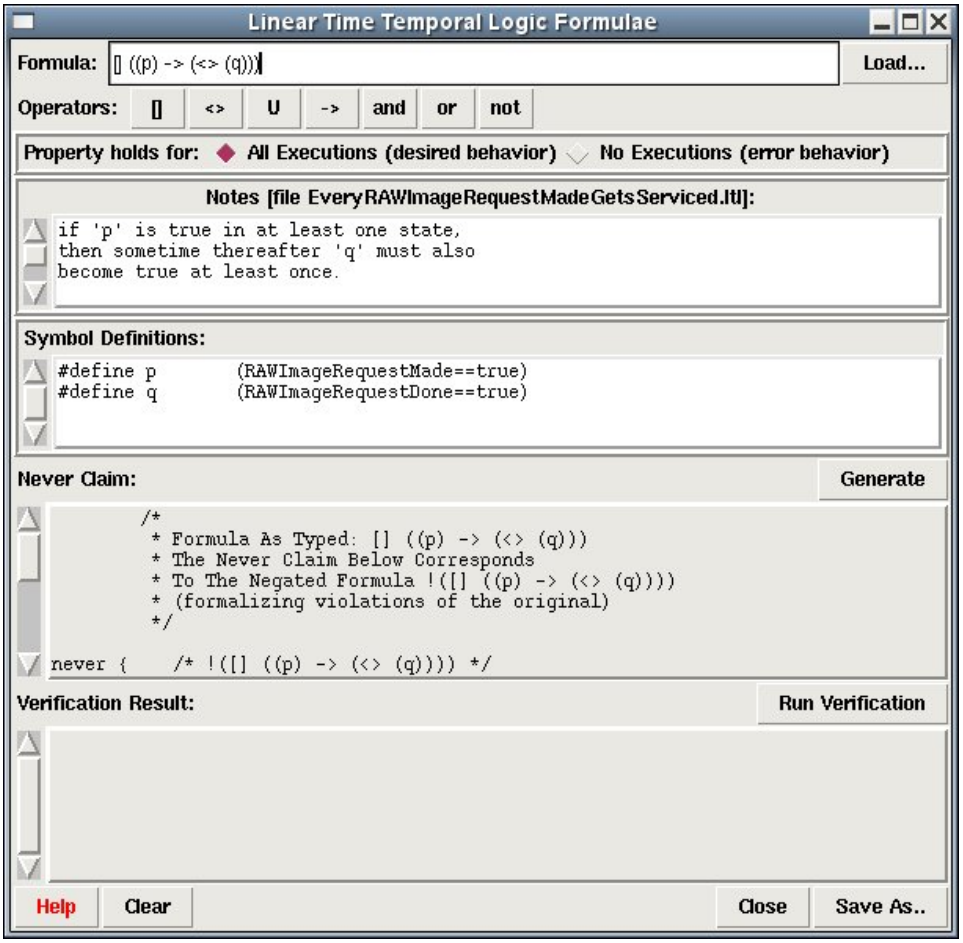


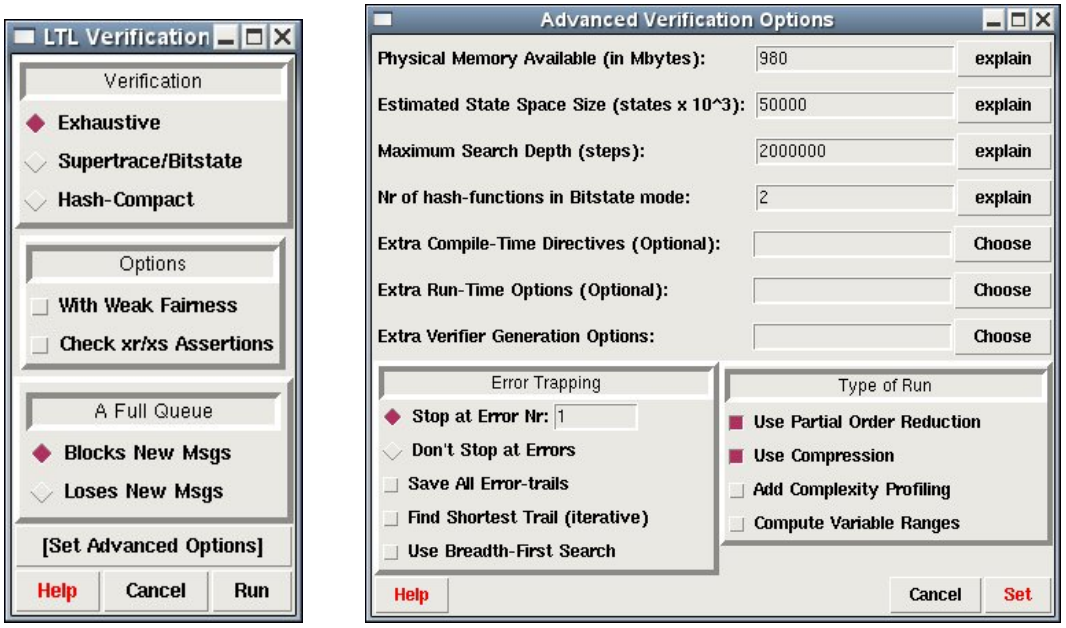
Figure 5.1: The LTL manager used to specify temporal claims.

- **Never claim**: It was stated earlier in this thesis that the *SPIN* verifier uses *never claims* to verify correctness properties. In order to verify the correctness property represented in an LTL formula, the LTL formula needs to be translated into a *never claim*. The *SPIN* verifier is able to automatically do the needed LTL → never claim translation. The result of the translation is displayed in the never claim section of the LTL manager window.

- **Verification result**: This section displays all types of results for a given verification run. The most important of these results is whether or not the correctness property being verified for a given model is valid or not.

5.4.2 Set the verification options

The *SPIN* verifier offers the user a variety of options when it comes to the verification of correctness properties. The options are shown in Figure 5.2. The most important options used for the request-processor verification runs will now be highlighted. A more in-depth discussion of these options can be found in [3].



(a) Verification options (b) Advanced verification options

Figure 5.2: The verification options.

- **Exhaustive verification run**: The *SPIN* verifier is able to perform three types of verification runs. The most important of these type of runs is the exhaustive verification run. An exhaustive verification run is truly exhaustive in that it tests all possible event sequences in all possible orders within the state space. An exhaustive verification run ensures an 100% coverage of the state-space, compared to the $0\% - 99\%$ coverage of a hash-compact or supertrace verification run. The exhaustive state method, however, has its disadvantages:

  1. it uses a large amount of RAM; and
  2. has an order of magnitude longer execution time when compared to that of hash-compact and supertrace verification runs.

The disadvantages of the exhaustive method is a small price to pay for a 100% state space coverage when verifying complex asynchronous system models.

- **Physical memory available**: It was mentioned previously that verification is a memory intensive operation. The *physical memory available* verification option indicates how much physical system memory may be used by the *SPIN* verifier to perform a verification run. This option is part of a advanced set of options which *SPIN* uses to optimize the verification runs for memory use and performance. When the *SPIN* verifier reaches the specified physical memory limit, verification is stopped to avoid trashing. In the case of the verification runs for the request-processor system, the limit was set at 980MB.

- **Estimated State Space Size**: This option is used to calculate the size of the hash-table used by the *SPIN* verifier during verification runs. Setting the value of the option too high may cause an out-of-memory error with zero states being reached. This means that the verification process could not be started. Setting the value of the option too low can cause inefficiencies due to hash collisions. The correct value of this option can be attained by experimental means. The adequate value for this option, where verification of the request-processor was concerned, was 50000×10^3.

- **Maximum search depth**: The value of this option determines the size of the depth-first search stack that is used by the *SPIN* verifier during the verification runs. The stack uses memory, so a larger value for this option increases the memory requirements for verifying a correctness property of the model. If the *SPIN* verifier reaches the maximum search depth during a verification run, it may not find a sequence of states that violate the correctness claim, because an insufficient search depth in effect reduces the coverage of the state space. The value that was used for this option during the request-processor verification runs was 2000000.

- **Partial order reduction and compression**: These options can be used by the *SPIN* verifier to optimize memory usage and performance for the verification runs. Both options were enabled during the request-processor verification runs.

5.4.3 Perform the verification run and interoperate the results

The result of a verification run is shown in Figure 5.3. Figure 5.3 depicts the result of the verification for the following correctness property:

Every raw-image request is eventually processed.

$$[](RAWImageRequestMade_i \Rightarrow <> RAWImageRequestDone_i)$$

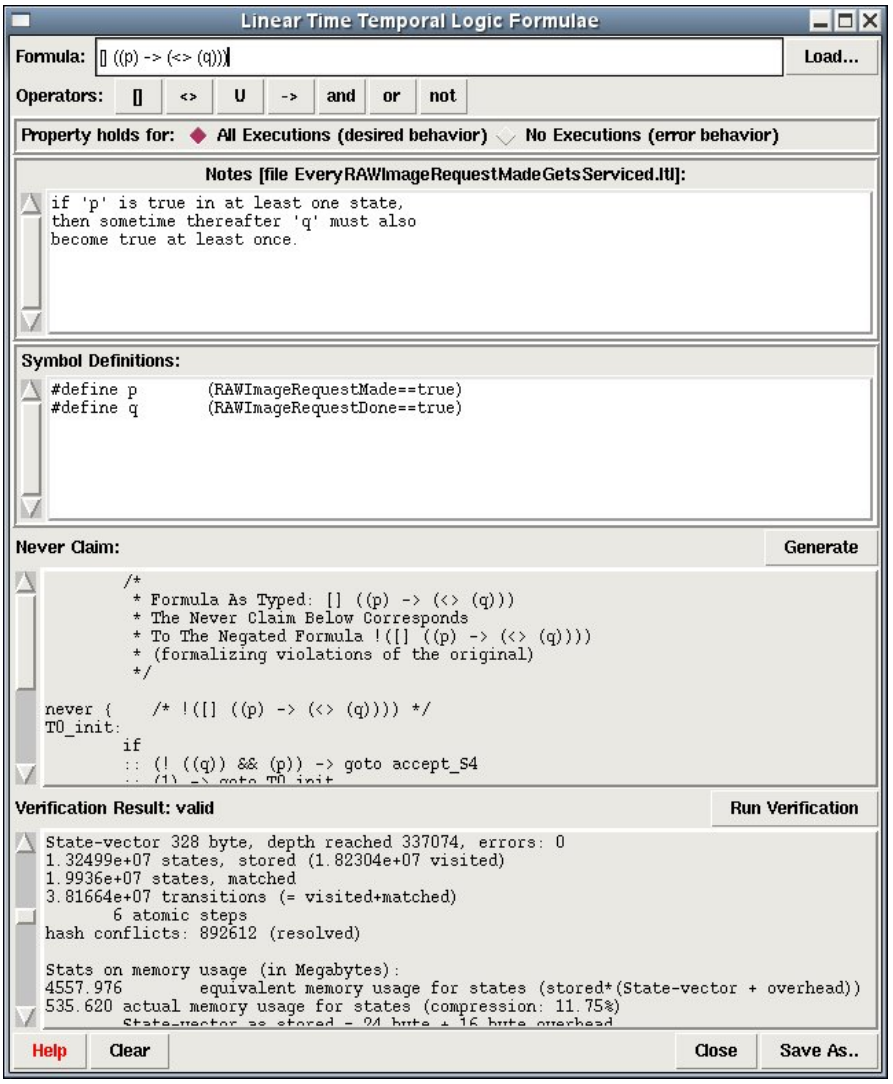


Figure 5.3: Result for the RAWImageRequest verification run

The most important part of Figure 5.3 is the **verification result**. In this case the result indicates that the correctness property is **valid**. This means that the designed model does indeed satisfy to the requirement that every raw-image request that enters the request-processor system must eventually be processed. The verification output window contains a couple of interesting statistics of the completed verification run:

- **The size of a single state**: For the raw-image request verification run, the memory used to store a single state was 328 bytes.
- **The longest execution path**: For the raw-image request verification run, the longest execution path (search-depth) was 3337074.
- **Whether or not the correctness property was satisfied**: For the raw-image request verification run, the number of errors was 0. This not only indicates that the correctness property is satisfied by the model, but it also indicates that the modelled system is free of deadlock.
- **The size of the state space**: For the raw-image request verification run, the state space consisted of $1.32499e + 07$ states.
- **Total memory used for the verification run**: The raw-image request verification run required 875.360MB of physical memory (not shown in figure).

5.5 Verification results

All correctness properties listed earlier in this chapter were successfully verified for the request-processor system model. Table 5.1 summarizes the results of some of the verification runs that were performed for the model:

Property	**Verification result**	**Effective memory used**	**Actual memory used**	**Approximate time taken**
L2	Valid/Pass	4557.976MB	875.360MB	4 minutes
L3	Valid/Pass	4310.490MB	848.020MB	4 minutes
L4	Valid/Pass	4713.583MB	892.564MB	5 minutes

Table 5.1: Verification results and statistics.

The values in the *effective memory used* column in Table 5.1 are obtained by multiplying the number of stored states with the size of each state descriptor, and adding the overhead of the lookup table used for the verification run. The value can also be seen as a rough estimation of the amount of memory that would have been used, if the compression option was not enabled during the verification run. The verification runs not listed, (**L1** and **L5**) in table 5.1, produced similar results. The successful verification

of the request-processor system proves that the system does what it was designed to; to process and respond client-requests.

Chapter 6

From model to implementation code

A *PROMELA* model is not of much practical use on its own. It can be used to prove that a system design satisfies its requirements, but if the model of the design cannot be translated into a modern programming language such as C, the proven design becomes just another theoretical exercise. Fortunately, it is possible to derive a software implementation from a transition system based *PROMELA* model. The derivation from *PROMELA* model to implementation code is achieved by making use of simple mapping rules. The mapping rules ensure that all the benefits of model verification remain intact. This chapter highlights the mapping rules for the C programming language. The C programming language was used to implement a software prototype of the request-processor system.

6.1 *PROMELA* to C

Realizing the value of *modelchecking* in software system design, one would think that there exists a program that is able to translate a verified *PROMELA* model into a mainstream programming language such as C. The assumption would unfortunately be incorrect. However, a *PROMELA* model closely resembles a modern programming language. It has well-known programming structures such as *if*-statements, *repeat*-statements and variables. This makes it possible to translate a *PROMELA* model into any modern programming language using the appropriate translation mappings. The following section looks at mappings that can be used to translate a *PROMELA* model into a C-based software system.

Translating the `proctype` declaration

The *PROMELA* `proctype` declaration was introduced in chapter 3. It was stated that a `proctype` declaration defines a single system process and its behaviour. In C a process can be a standalone program or a program thread. The choice of whether to use a standalone process or a program thread as the mapping for a `proctype` declaration depends on the system being developed. Elements such as performance and user requirements will normally indicate which one of the two mappings to use. In the case of the request-processor system that was designed as part of this thesis, one of the design decisions was to not use program threads. The `proctype`$-$*declaration* $\Leftrightarrow$ *standaloneprogram* mapping was thus used to implement the request-processor model processes.

Decision structures

PROMELA decision structures such as the *if-fi*-structure can easily be translated into C. One can use the following mapping:

if-fi $\Leftrightarrow$ *if*-statement

An example of an *if-fi* $\Leftrightarrow$ *if*-statement mapping can be seen in listings 6.1 and 6.2.

Listing 6.1: *PROMELA if-fi*

```
if
:: a == b->
   a = 1;
:: a < b->
   b = 9;
:: else->
   b = a-9;
fi
```

Listing 6.2: C code *if-fi*-mapping

```
if(a == b){
    a = 1;
}else if{
    b = 9;
}else{
    b = a-9;
}
```

Repeating structures

One of the most common repeating structures in *PROMELA* is the *do-od*-structure. The *do-od*-structure can be translated into C by using the following translation mapping:

do-od $\Leftrightarrow$ *while*-loop

Listing 6.5: *PROMELA*. non-determinism

```
proctype client(chan clientRHOut,out)
{
  do
  ::true->clientRHOut!clientRequest(RAWImageRequest);
  ::true->clientRHOut!clientRequest(processedImageRequest);
  od;
}
```

The *while*-loop mapping has a very specific form. The statements within a *PROMELA* *do-od*-structure execute infinitely. This means that the C mapping must also be an infinite loop.

An example of an *do-od* ⇔ *while*-loop mapping can be seen in listings 6.3 and 6.4.

Listing 6.3: *PROMELA do-od*

```
do
:: a == b->
   a = 1;
:: a < b->
   b = 9;
od
```

Listing 6.4: C code *do-od*-mapping

```
while(1){ //infinite loop
    if(a == b){
        a = 1;
    }else if(a < b){
        b = 9;
    }
}
```

Non-deterministic behaviour

In *PROMELA* it is possible to specify a model such as the one listed in Listing 6.5

In the model, multiple guard conditions simultaneously evaluate to *true*. If *SPIN* is used to verify the model listed in Listing 6.5, it will non-deterministically choose which statement to execute. The fact that *PROMELA* allows multiple guard conditions to evaluate to *true*, makes it easy to model the complete behaviour of the system. One may ask: *How can this non-deterministic behaviour be translated into a modern programming language such as C?*

Non-determinism is used to model the complete asynchronous execution behaviour of a system. It is used by the *SPIN* verifier to verify correctness properties of the system. Implementation code cannot specify the execution behaviour of a system. It can only specify a sequence of statements to execute. When keeping this in mind, a possible translation for multiple *PROMELA* guard conditions that simultaneously evaluate to *true*,

is multiple *if*-statements which works in conjunction with some sort of random boolean condition. The translation is illustrated in listings 6.6 and 6.7.

Listing 6.6: *PROMELA* non-determinism

```
a = 5; b = 3;
do
:: a == 5->
   a = 1;
:: b == 3->
   b = 9;
od
```

Listing 6.7: C code non-determinism

```
a = 5; b = 3;
while(1){ //infinite loop
    j=rand(1,2); //choose randomly 1 or 2
    if(a == 5) && (j=1){
        a = 1;
    }else if(b == b)&& (j=2){
        b = 9;
    }
}
```

Message channels

Interprocess communication is an important aspect of a multi-process system. It was stated earlier that *PROMELA* can use message channels to model interprocess communication. Modern programming languages such as C provide various interprocess communication schemes. The availability of the schemes depends on the programming language and operating system. The three most common schemes in C, on a *UNIX*-based system are:

- TCP/UDP-sockets
- Shared memory
- Named pipes

As the name implies, the shared memory scheme uses a shared block of memory between processes to enable the processes to communicate with each other. Named pipes are also shared memory-based. Shared memory-based interprocess communication schemes work well in systems where the processes are able to share a common block of memory. These type of schemes do however not scale well and have inherent concurrency implementation problems. For this reason, this type of shared memory-based schemes are not a preferred mapping for *PROMELA* message channels.

TCP/UDP-sockets is one of the mappings that can be used for *PROMELA* channels. Sockets are highly configurable. It is for instance possible to configure a socket to be

blocking or *non-blocking*. It is thus easy to mimic the exact behaviour of *PROMELA* message channels. Sockets can also be used for interprocess communication between processes of a distributed system - named pipes and shared memory cannot. TCP sockets were used to map all the *PROMELA* message channels used in the request-processor model, to a C-based equivalent.

Translating the `init`-process

According to the definition of the *PROMELA* `init`-process, it is responsible for initializing a *PROMELA*-modelled system. The `init`-process does not play any other role in the modelled system. The fact that the `init`-process only plays an initialisation role in a system model means that it can easily be translated into a software implementation of the modelled system. One appropriate translation is to use *batch*-files to initialize all the processes of a designed system.

The *batch*-file approach was used in the software implementation of the designed request-processor system. The `init`-process functionality is simulated using a single *batch*-file which starts all the processes of the request-processor system.

6.2 Transition system

It was stated in chapter 4 of this thesis that the processes of the request-processor system were modelled based on a transition system. The transition system process design gives the processes a logical structure and makes it easier to trace and fix errors. In order to preserve these properties in the final software prototype of the system a C-based transition system is needed. Listing 6.8 shows the transition system that was used in the software implementation of the request-processor processes.

The C-based transition system depicted in Listing 6.8 consists of an infinite *while*-loop and seven function calls.

performStateBasedPreprocessing (line 3)

The purpose of the `performStateBasedPreprocessing` function is to perform any pre-processing that is required by a process before it listens for peer token requests. The request-dispatcher process, for example, must in state 1 first send an *init*-token to the request-handler process before moving to state 2.

Listing 6.8: The C-style transition system

```
1  while(1)
2  {
3          performStateBasedPreprocessing();
4          setupSelectParameters(locationPointer,&fd,&readfds);
5          retval= Select(fd+1,&readfds,NULL,NULL,&tv);
6
7          if(retval != 0)
8          {
9                  handleIncommingMessages(readfds);
10                 performStateBasedPostprocessing();
11         }else if(retval==0)// A timeout occured
12         {
13                 performStateBasedTimeOutProcessing();
14         }
15         //Set the timeout of the select
16         setTime(&tv,selectTimeOutSec,selectTimeOutuSec);
17
18 }
```

setupSelectParameters (line 4)

The `setupSelectParameters` function modifies the parameters needed by the *POSIX* `select`-system call. The modifications are made based on the current state. A process knows which peer token it must accept and respond to, based the current state. The request-dispatcher process can for instance accept and respond to tokens originating from the request-handler, memory-controller and scheduler-dispatcher or image-processor process. In state 2, the request-dispatcher can only accept and respond to tokens that originate from the request-handler process.

Select (line 5)

The `Select` function, wraps the *POSIX* `select`-system call, which in turn monitors a *read* file-descriptor. The file-descriptor is linked to the socket connection which the process needs to monitor in the current state. The function call will return when there is something to be read from the connection (a peer process sends a token to the process) or when a predefined timeout period has elapsed.

handleIncommingMessages (line 9)

The `handleIncommingMessages` function gets called if the `Select` function did not return because of a timeout. The `handleIncommingMessages` function processes the tokens, the process receives from its peers.

performStateBasedPostprocessing (line 10)

The purpose of the `performStateBasedPostprocessing` function is to perform any post-processing that is required by a process before it switches to the next state in its transition sequence. The `performStateBasedPostprocessing` function was not actively used in the software implementation of the request-processor processes seeing that the processes required almost no state-based post-processing. The function is only present to complete the C-based transition system.

performStateBasedTimeOutProcessing (line 13)

The `performStateBasedTimeOutProcessing` function defines what happens when the `Select` function retuned due the elapse of a predefined timeout period. The function might perform some processing, but it primarily calculates the next sate of the process.

setTime (line 16)

The only purpose of the `setTime` function is to reset the timeout value used by the `Select` function-call.

6.3 The translated request-processor system

The C implementation of the request-processor *PROMELA* model consists of an initialization *batch*-file and seven console applications. Each console application represents a different process in the request-processor system. The console applications are:

1. the *client* application,
2. the *request-handler* application,
3. the *sender* application,
4. the *request-dispatcher* application,

5. the *memory-controller* application,

6. the *scheduler-dispatcher* application; and

7. the *image-processor* application,

The different applications communicate with each other by using TCP/IP-connections. The *batch*-file starts all the applications and provides every application with the appropriate IP-addresses and port numbers.

On its own the implemented request-processor prototype system is not much to look at. All that a client notices is that the request-processor system accepts and responds to its requests. This in itself is not a problem, seeing that this is exactly what the request-processor system was designed to do, but the difficult part of the system design was the resource allocation and synchronization in the system. A Java GUI was thus also developed. The Java GUI connects to the request-handler and request-dispatcher applications and periodically collects the queue data of the system. The collected queue data is displayed in the form of graphs. The Java GUI is thus able to show how the request-processor system is processing requests. A screen shot of the Java GUI is shown in Figure 6.1.

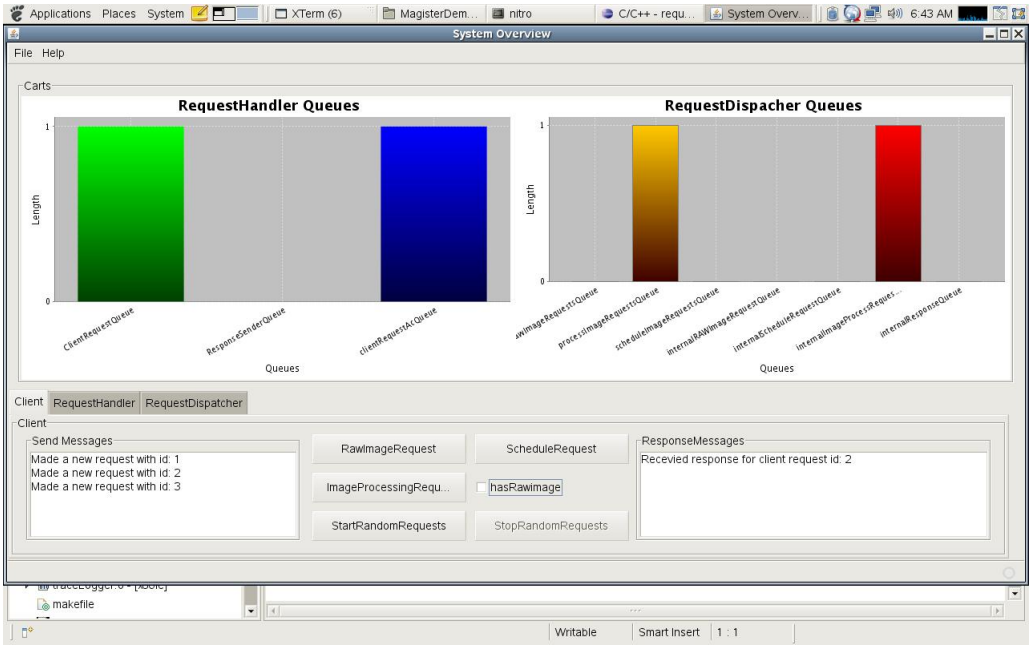


Figure 6.1: A screen shot of the JAVA GUI.

6.4 Notes on testing

Deriving implementation code from a verified *PROMELA* model is not a automated process. This means that the translation from model to implementation code must be meticulously checked to ensure that the logical correctness properties of the *PROMELA* model is still present in the implementation code.

In the case of the C-based implementation of the request-processor, the transition-system design of the processes made it easy to ensure an exact *PROMELA*-model $\Leftrightarrow$ C-code translation.

A *PROMELA* model is a high-level representation of a system design. The model always contains some level of abstraction. The request-processor model is no different. It for instance abstracts the message-queues that are maintained by all the different processes in the system. The actual C-based implementation of these queues needed to be tested. The *PROMELA* abstraction of the queues do not define how the queues must be implemented, but it does define the behaviour of the queues. This makes it possible to define a traditional software test-set for the queue-implementations.

The C-based queue implementation of the request-processor system was tested to verify that it worked as expected. The queue implementation was mainly subjected to some functional tests (black-box) testing. Table 6.1 highlights some of the functional tests that were performed on the queue implementation. The size of the queue was set to 10 elements.

The request-processor system was also subjected to some reliability tests. In the tests, some of the processes of the system were killed, while the system was busy processing requests. The system had to be able to recover when the killed processes were restarted. The reliability tests conducted were successful.

The final C-based implementation of the request-processor system contained a lot of other implementation detail that were abstracted in the *PROMELA* model of the system. All the implementation detail of the system was tested using functional tests similar to those of the queue implementation. All the tests that were performed, was successful. Some demo runs of the final, C-based implementation prototype, showed that the system performed as expected.

Test	Description	Result
Insert test	The purpose of this test was to see if elements could be inserted into the queue.	Elements were successfully inserted into the queue.
Capacity test	The purpose of the test was to see whether or not the queue can store exactly 10 elements	The result of the test confirmed that the queue can indeed store exactly 10 elements.
Overflow test	This test was designed to test whether or not the queue rejects additional element insert operations, when it is already filled to capacity. When trying to insert eleven elements without removing any, the queue implementation must reject the eleventh element.	The result was positive. The queue implementation accepted ten elements and rejected the eleventh element.
Remove test	The purpose of the test was to test whether or not the queue implementation can remove elements from the queue, while still maintaining the queue's integrity.	The test was successful. The queue integrity was maintained throughout successive insert and removal operations.
State report test	The queue implementation is meant to keep track of the amounts of elements in the queue. This test consists of a series of insert and remove operations. After each operation the queue implementation must report the correct amount of elements in the queue.	The test was successful. The queue implementation consistently reported the correct amount of elements in the queue.

Table 6.1: Some functional tests performed on the queue implementation.

Chapter 7

Conclusion, recommendations and summary

7.1 Conclusion

The purpose of the work conducted for this thesis, to design and implement a reliable and responsive request-processing software system, which can be used as the basis for a request processing framework for systems with limited request processing resources, was met.

1. It has satisfied the three main requirements outlined for the system, namely:
   - it is able to process and respond to image processing requests;
   - it is responsive; and
   - it is as reliable as possible.
2. However, during the development cycle of the request-processor it became clear that it can not reliably prove the correctness of the system. The reasons are the following:
   - The request-processor had to control access to limited system resources.
   - Design considerations dictated that the request-processor must consist of multiple processes.
3. Furthermore, the required multiple process design poses the problem of interprocess communication and concurrent process execution. During the design phase of this system, it became clear that the only reliable way to implement the request-processor was to first build a system-model. The model could then be verified for correctness by using *modelchecking* tools such as the *SPIN* verification system. The

verified request-processor model was finally translated into a C-based prototype. The prototype was tested and proven to work satisfactorily.

7.2 Recommendation

1. The final implementation of the request-processor system does not include the ageing of requests in the system. A scenario might occur where the system is paused for a prolonged period of time. The request-processor keeps track of the requests in the system by making use of queues. When the system resumes, the request-processor will continue to process the requests in the queues even though some of the requests may not be valid any more. Thus, non-ageing requests have an undesirable impact on the performance of the request-processor system, and therefore it is recommended that ageing requests should be incorporated into future versions of the system. The proper way to add ageing requests to this system should include adding the ageing requests to the system model. This will require optimization the existing request-processor model, seeing that the model already used 1 GB of memory when its correctness properties were verified.

2. The final implementation of the request-processor system contains almost no request-processing code. It is recommended that the implementation detail of the resource-processes of the system be refined. In the C-based implementation of the system, the processing-time of the system-resource processes, was simulated by using configurable delay-timers.

7.3 Summary and contribution

This thesis presents a reliable and responsive request-processor prototype which can be used as the basis for a request processing framework for systems with limited request processing resources. The developed system was checked by formal means, to prove the correctness and reliability, as far as possible.

It is foreseen that with the necessary adaptation for a specific implementation, the presented solution might be conveniently utilised as a component in a variety of practical and critical applications, such as remote standalone data acquisition, satellite applications and automatically piloted vehicles.

List of References

[1] Cooke, A.: *Rural E-mail System for the Sumbandila Satellite.* Master's thesis, Stellenbosch University, 2007.

[2] Dijkstra, E.: Notes on structured programming. 1969.

[3] Holzmann, G.J.: *The Spin model checker Primer and Reference Manual.* Addison-Wesley, 2004.

[4] Sipser, M.: *Introduction to the theory of computation.* PWS Publishing Company, 1997.

[5] YAHODA verification tools database: `http://anna.fi.muni.cz/yahoda`.

[6] Swart, R.: *A language to support verification of embedded software.* Master's thesis, Stellenbosch University, 2003.