university of groningen

faculty of mathematics and natural sciences

Mutual Exclusion Using Weak Semaphores

Master's Thesis Computer Science

January 2011

Student: Mark IJbema

Primary supervisor: Wim H. Hesselink

Secondary supervisor: Marco Aiello

Abstract

In this thesis we describe two algorithms which implement mutual exclusion without individual starvation, one by Udding[8] and one by Morris[7]. We prove, using the theorem prover PVS, that they both implement mutual exclusion, and are free from both deadlock and individual starvation. Though the algorithms were provided with a proof, they were not mechanically verified as of yet.

We conclude by looking at a conjecture by Dijkstra, which was disproved by the existence of the algorithms by Udding and Morris. We weaken it in such a way that it is true, and prove it informally.

Contents

Chapter 1

Introduction

In this thesis we define and introduce the problem of mutual exclusion, and discuss the implementations and proofs of two algorithms which implement mutual exclusion without individual starvation, as proposed by Udding[8] and Morris[7].

To this end we first discuss the problem of mutual exclusion in chapter 2. Then we present semaphores as a possible solution to this problem in chapter 3. We continue by discussing the different types of semaphores. In the last part of chapter 3 we also present and interpret a conjecture about the strength of different types of semaphores, as posed by E.W. Dijkstra.

In chapter 4 we pose the problem of solving the mutual exclusion problem using weak semaphores. From this problem we derive the algorithms as presented by Udding and Morris. We present the exact implementations for these algorithms in chapter 5.

Because we proved the correctness of the algorithms in PVS, a theorem prover, we first discuss how one goes about proving statements about programs in PVS, in chapter 6. We show how to model a program, formulate invariants, and prove invariants.

Chapters 7, 8, 9, and 10 contain a human-readable version of the proofs that both algorithms confirm to the safety properties: Mutual Exclusion (chapter 7), Freedom from Deadlock (chapter 8), and Freedom from Individual Starvation (chapters 9 and 10). We have proved these properties in PVS, but in these chapters we refrain from going into detail about the implementation of this proof.

In chapter 11 we present a method to compare both algorithms to other algorithms solving mutual exclusion. The goal is to be able to compare both algorithms to a wait-free algorithm as proposed by Wim H. Hesselink and Alex A. Aravind in [4].

In chapter 12 we formulate and informally prove a weakened version of the conjecture by E.W. Dijkstra.

Lastly we present suggestions for future work in chapter 13 and formulate our conclusions in chapter 14.

1.1 Appendices

For the interested reader we offer a few appendices. In appendices A and B we provide a list of all invariants used in the PVS-proof. For each invariant we provide both a reference to the page where it is introduced, and the name of the invariant in the PVS-proof.

In appendix C we offer justification for the choice of type of semaphore used in our model for the algorithms.

Chapter 2

Mutual Exclusion

If we have a program with multiple processes we desire a mechanism to ensure that certain parts of the program cannot be executed by multiple processes at the same time[1]. Let us look at this simple fragment for instance:

$x := x - 1;$

One might expect that, since this is only a single line of code, only one process at the time can execute it. However, in machine instructions this simple line of code translates to several lines of code, and might look something like this:

$tmp :=$ **read** $x;$
$tmp := tmp - 1;$
write tmp **to** $x;$

Here the tmp variable is local, that is, each thread has one, whereas the x variable is shared. When we run this program with multiple processes, the scheduler might decide to switch the active process at any moment, so there is no way to predict in which order the program will be executed if multiple processes are involved. We can therefore think of the following example of an execution, where this program will not give the result we would expect it to.

Suppose we have two processes A and B which execute this code fragment concurrently, both with their own tmp variable ($A.tmp$ and $B.tmp$), and one shared variable x (initially 7). In the following table we show what line(s) are executed for each process, and to the right the state after execution of those lines is shown:

A executes	B executes	$A.tmp$	$B.tmp$	x
		?	?	7
	$tmp :=$ **read** $x;$	?	7	7
$tmp :=$ **read** $x;$		7	7	7
	$tmp := tmp - 1;$	7	6	7
	write tmp **to** $x;$	7	6	6
$tmp := tmp - 1;$		6	6	6
write tmp **to** $x;$		6	6	6

Now two processes tried to decrement x, but it is only decremented once. We frequently encounter situations like this, where we have a certain section of the program which must be executed atomically by one process, before another

process is allowed to execute the same fragment. Such a section is called a *critical section*.

It is also important to look at what we call "atomic". This means a piece of code is executed by a process without interruptions from other processes. However, we do not really care about processes which only change non-related variables; therefore we also call an execution atomic when the execution can be reordered in such a way that the outcome is the same, and the lines to be executed atomically are not interspersed with other lines of other processes.

2.1 Solutions to the Mutual Exclusion Problem

Several mechanisms have been invented to ensure that no two processes can execute the critical section at the same time. Broadly speaking these mechanisms can be divided into two categories: with waiting and without waiting. Recently, there has been more interest in solutions without waiting.

A program which uses waiting has some sort of mechanism which can cause processes to 'go to sleep' to be 'awoken' later by the scheduler. A program without waiting has no such mechanism, and at each point in time, each process is able to perform a step.

One solution without waiting is presented by Wim H. Hesselink and Alex A. Aravind in [4]. A referee commented that their solution looked a lot like two solutions with waiting, one by Udding[8] and the other by Morris[7].

In this thesis we concentrate on the latter two algorithms, by proving them rigourously using interactive theorem proving. After that we return to the algorithm by Hesselink and Aravind, and propose a method to compare the algorithms by Udding and Morris to it.

Chapter 3

Semaphores

To implement mutual exclusion with waiting, one of the most obvious mechanisms to use is the semaphore.

A semaphore allows processes to continue or halts them, depending on whether the semaphore is open. A semaphore consists of two operations, the $\mathcal{P}$ and $\mathcal{V}$ operation,[1] and a variable, say s. The $\mathcal{P}$ operation is placed before the critical section. It only allows a process to continue if the semaphore is open, and then it immediately closes the semaphore. The $\mathcal{V}$ operation is placed after the critical section, and opens the semaphore, thus allowing processes to pass the semaphore. The section between the $\mathcal{P}$ and $\mathcal{V}$ operations is called the guarded section. To ensure the critical section is executed by at most one process at a time the critical section is typically placed inside a guarded section.

To ensure that the piece of code from the previous chapter is executed atomically, we could enhance it with semaphore operations:

$\mathcal{P}(s)$;
$x := x - 1$;
$\mathcal{V}(s)$;

A semaphore is modelled by a natural number s initialized to 1, and by the following definitions of the atomic $\mathcal{P}$ and $\mathcal{V}$ operations:

procedure $\mathcal{P}(s)$
 $\langle$**await** $s > 0$;
 $s := s - 1;\rangle$

procedure $\mathcal{V}(s)$
 $\langle s := s + 1;\rangle$

Note that this implementation is in fact more general, since s can take on the value of any natural number; one could use this mechanism to allow at most two processes to execute a section, for instance. However, in this document we restrict ourselves to semaphores which only take on the values 1 and 0, the so-called binary semaphores.

[1] The $\mathcal{V}$ stands voor the Dutch word "verhogen" (increase), and the $\mathcal{P}$ stands for the made-up Dutch word "prolaag", which is a contraction of "probeer" and "verlaag" (respectively try and decrease).

3.1 Safety Properties

When we want to extend the usability of semaphores, we want to ensure that a semaphore satisfies several properties implying that the program 'works as one would expect'. There are three properties which are commonly regarded as the important safety properties for binary semaphores, when used in the context of mutual exclusion.

3.1.1 Mutual Exclusion

The first property is mutual exclusion: only one process can be in the guarded section at any one time.

3.1.2 Freedom from Deadlock

The second property is that we do not want the program to come to a halt before it is finished. This means we do not want to have a situation where no process is able to perform an action. This property is called freedom from deadlock.

3.1.3 Freedom from Starvation

The third desirable property is having some form of progress for each process; we do not want it to be possible for some process to wait indefinitely, while other processes keep executing the guarded section. This is called freedom from starvation.

We do have some freedom in expressing the exact terms of this property. We want some progress, so the weakest way to demand this is that there is no process which waits forever at some point. In practice the stronger demand of bounded overtaking is desirable: each process is overtaken at most a certain number of times, for instance, at most twice by any other process. This gives a better worst-case performance, and is often easier to prove.

3.2 Semaphores with a Progress Property

Morris and Udding use different descriptions for the semaphores they used, even though they have similar goals. Morris' description is:

> "If any process is waiting to complete a $\mathcal{P}(s)$ operation, and another process performs a $\mathcal{V}(s)$, the semaphore is not incremented, and one of the waiting processes completes its $\mathcal{P}(s)$"

The advantage of this semaphore is that it conforms to the invariants from section 3.3.1. The only difference to a semaphore without a progress property is when there are processes waiting, and a $\mathcal{V}$ operation is executed. In this case the semaphore stays 0, but some process is directly 'pushed' into the guarded section. The disadvantage is that the property is very strong. Not only is the process which may pass explicitly chosen, it also has to make a step immediately, meaning the semaphore influences the scheduler as well.

Udding uses an apparently weaker property, being:

> "For weak semaphores, one assumption is made, however, viz. a process that executes a $\mathcal{V}$-operation on a semaphore will not be the one to perform the next $\mathcal{P}$-operation on that semaphore, if a process has been blocked at that semaphore. One of the waiting processes is allowed to pass the semaphore."

This is only marginally weaker, because the only difference is when the elected process makes the step into the guarded zone. In Udding's version this might be postponed until the elected process is scheduled again, instead of immediately. On first reading Udding's definition seems much weaker, because of the (redundant) first sentence. An interesting question is whether the algorithms would still be correct without the second sentence, that is, only demanding that the last process to execute the $\mathcal{V}$ operation cannot be the first to execute the $\mathcal{P}$ operation, if already processes were waiting. We look further into this question in appendix C.

3.3 Formal Definitions

We now present the formal definitions used for semaphores in this document. We start by showing the invariants we have for semaphores, and then continue by introducing several types of semaphores.

3.3.1 Invariants for Semaphores

We can describe the semaphores we are going to use more formally by defining several invariants which must hold for a binary semaphore:

(SEM0) $s \geq 0$

(SEM1) $s \leq 1$

Additionally, since we only use semaphores in the context of mutual exclusion, we also want the following two invariants to hold:

(SEM2) there is a process in guarded section $\Rightarrow s = 0$

(SEM3) $s = 0 \Rightarrow$ there is a process in guarded section

If we assume the first invariant to be trivial (since we take s to be a natural number[2]) we can summarize the invariants as follows:

(SEM) $\#(\text{processes in guarded section}) + s = 1$

3.3.1.1 Split Binary Semaphores

For the algorithms discussed in this document we also need a so-called split binary semaphore[5]. This is a construct consisting of two (or more) actual semaphores. Furthermore, the two semaphores together behave as a semaphore. This means that if $s_1 + s_2$ is a split binary semaphore we get almost the same invariant:

(SSEM) $\#(\text{processes in guarded sections}) + s_1 + s_2 = 1$

In the code, each $\mathcal{P}$ on s_1 or s_2 is followed by a $\mathcal{V}$ on s_1 or s_2. Note that a $\mathcal{P}$ on s_1 does not necessarily require a $\mathcal{V}$ on s_1 as next $\mathcal{V}$ operation. This means that both $\mathcal{P}(s_1) \ldots \mathcal{V}(s_1)$ and $\mathcal{P}(s_1) \ldots \mathcal{V}(s_2)$ are valid blocks. This allows us to have multiple guarded sections, and to guide which section will be executed next.

3.3.2 Types of Semaphores

We can now introduce the four types of semaphores we use in this document: the weak semaphore, which doesn't have any progress property, and three types of semaphores which do have a progress property.

[2] We use natural numbers in the same way PVS does: non-negative integers.

We denote atomicity by the usage of line numbers. One line is one atomic action. Furthermore we introduce the variable 'self' which refers to the process identifier of the executing process, and remark that -1 is not a valid process identifier.

3.3.2.1 Weak Semaphore

The weak semaphore does not offer any guarantees with respect to individual starvation. One important implication is that this semaphore does not make any difference between processes attempting a $\mathcal{P}$ operation for the first time, and processes which are already blocked on a $\mathcal{P}$ operation.

A semaphore is weak if and only if it is logically equivalent to the following one:

Algorithm 3.3.1: Weak Semaphore

```
procedure P(s)
 10     if s > 0
          then s := s − 1; goto 11;
          else goto 10;
 11     ...

procedure V(s)
 20     s := s + 1;
```

This semaphore conforms to the aforementioned invariants.

3.3.2.2 Polite Semaphore

The difference between a weak semaphore and a semaphore which has some progress property becomes apparent when processes are blocked on a $\mathcal{P}$ operation. So we consider the situation that process p executes a $\mathcal{V}$ operation on semaphore s, while other processes are blocked on the $\mathcal{P}$ operation of semaphore s. We propose a new type of semaphore, which forbids process p to be the first to execute a $\mathcal{P}$ operation on semaphore s. We call this type of semaphore the polite semaphore. Note the importance of a second state for processes which have already attempted the $\mathcal{P}$ operation but have failed.

The specification of this semaphore might look like this:

Algorithm 3.3.2: Polite Semaphore

```
procedure P(s)
 10     if s > 0 ∧ self ≠ forbidden
          then
            s := s − 1; forbidden := −1; goto 12;
          else goto 11;
 11     await s > 0 ∧ self ≠ forbidden;
        s := s − 1; forbidden := −1;
 12     ...

procedure V(s)
 20     if proc_at(11) ≠ ∅
            then forbidden := self;
        s := s + 1;
```

Note that for this semaphore the invariants from section 3.3.1 hold true. Furthermore we need the following invariant for freedom from deadlock:

$$forbidden \neq -1 \Longrightarrow \exists p \in proc\_at(11) : p \neq forbidden \tag{3.1}$$

It is easy to see that this invariant is initially true (since then there is no process at line 11), and is at no point invalidated.

3.3.2.3 Buffered Semaphore

The buffered semaphore offers a stronger form of progress: when a process executes a $\mathcal{V}$ operation, and there were waiting processes, it is guaranteed that one of these waiting processes is the first to enter the guarded section.[3]

This guarantee is obtained in the following way: a process executing $\mathcal{P}(s)$ is allowed to continue (and decrement s) if $s > 0$, but if $s = 0$ the process is added to a set of waiting processes: the buffer. The process is then only allowed to continue when it is not in the set of waiting processes anymore.

Since $s = 0$, there is still a process in the guarded section. If the set of waiting processes is not empty when this process leaves the guarded section, one of the processes is released by removing it from the set from waiting processes. If the set of waiting processes is empty, s is incremented.

We hereby present a model:

Algorithm 3.3.3: BUFFERED SEMAPHORE

```
procedure P(s)
 21     if s > 0
          then
            s := s − 1;
            goto 23;
          else
            s_buffer += self;
            goto 22;
 22     await self ∉ s_buffer; goto 23;
 23     ...

procedure V(s)
 40     if s_buffer = ∅
          then s := s + 1;
          else extract some q from s_buffer;
```

An important difference between this semaphore and the previous ones is that s can remain 0 when a process leaves the guarded section. Therefore it is important that we consider processes on line 22 which are not in s_{buffer} to be in the guarded section. In that way the invariants remain valid.

3.3.2.4 Strong Semaphore

A strong semaphore is defined by the fact that it does not allow for individual starvation. This means that once a process blocks on a $\mathcal{P}$ operation, it can be overtaken at most a finite number of times, before it is allowed to enter the guarded section. Note that this is not the case for the previous two semaphores, when there are more than two processes involved.

[3] This is not the case for the polite semaphore. This semaphore can also allow a later arriving process to pass.

Because this demand is rather imprecise it allows for diverse implementations. One strong version is when we take the buffered semaphore, and replace the set by a queue. In that way we enforce a strict order, and in fact guarantee that no other processes will overtake a process once it is blocked.

3.4 Dijkstra's Conjecture

In practice all types of semaphores conform to the first two safety properties, mutual exclusion and freedom from deadlock. This is not true for freedom from starvation. Therefore it is a good idea to look at the amount of safety the different kind of semaphores offer in this aspect. First we consider the strong semaphore.

The problem with strong semaphores is that they are not offered by most hardware and therefore require a non-trivial software effort to implement. Most operating systems do however offer a weak semaphore, which offers no guarantees in the sense of freedom from individual starvation. Note that this means that individual starvation can already happen with only two processes.

According to Dijkstra[3] it is not possible to solve the mutual exclusion problem for an undefined number of processes using only a fixed number of buffered semaphores:

> "A starvation-free solution [to the mutual exclusion problem] using a fixed number of weak semaphores is not known for an unbounded number of processes, and can probably be shown not to exist."

Morris claims that this is possible[7], and presents a solution. However, according to Annot et al.[2], this is not a refutal of Dijkstra's Conjecture. They claim the weak semaphores Dijkstra refers to are the ones mentioned above, and not the buffered semaphores. This, however, is clearly untrue, as Dijkstra uses the following definition for weak semaphores:

> "In the weak implementation it is left absolutely undefined which of the blocked processes is allowed to proceed when two or more processes are blocked by $\mathcal{P}(m)$ when $\mathcal{V}(m)$ is executed."

In this definition he implies one of the blocked processes is allowed to proceed, which means it is a buffered semaphore. However, the other conjecture, that it is not possible to solve thu mutual exclusion problem with weak semaphores is interesting as well. Therefore we reformulate the weakened version of the conjecture of Dijkstra:

Conjecture 3.1. *It is not possible to solve the mutual exclusion problem using only a fixed number of weak semaphores.*

Chapter 4

Mutual Exclusion using Weak Semaphores

In this chapter we show how the two algorithms implement mutual exclusion using semaphores. We first give a bird's-eye view version of both algorithms, and then work our way down from there, to a concrete model of each algorithm.

4.1 The Mutual Exclusion Problem

The first thing we need to do is present the mutual exclusion problem. To this end we look at a setting in which all processes execute some non-critical section, and then the critical section, ad infinitum. This is shown in the following algorithm:

Algorithm 4.1.1: MUTUAL EXCLUSION PROBLEM

```
for each p : process
    while true
      NCS;
      entry;
      CS;
      exit;
```

The mutual exclusion problem can be stated as follows: to define *entry* and *exit* in such a way that mutual exclusion is guaranteed. In this chapter we also want that freedom from deadlock and freedom from starvation are guaranteed as well. We require the number of processes to be finite, though the number needs not to be known. In the rest of this chapter we do not look at the *entry* and *exit* sections separately, but instead discuss only the full algorithm, encompassing *entry*, CS and *exit*.

4.2 General Structure of the Algorithm

We want to solve the mutual exclusion problem using only weak and buffered semaphores. Note that we want to prevent individual starvation. To this end both algorithms are set up as a kind of airlock (like ones commonly found when entering buildings in very hot or very cold climates). First all processes enter the airlock one by one, until at a certain moment there are no more processes wanting to enter. Then the entrance is closed, and the processes are released

through the exit one by one. When everyone has left the airlock, the entrance is once again opened.

When we explain the algorithm like this it is easy to see that this implements mutual exclusion in the same way that a strong semaphore would do. Because all processes leave the exit one by one, it is safe to place the critical section here, and have guaranteed mutual exclusion. Furthermore freedom from individual starvation is also ensured. A process which has gone through the whole algorithm can never catch up with a process which is still in the middle of it, because the entrance stays closed until all processes have left through the exit. We define ne to be the number of processes that want to enter the airlock, and nm to be the number of processes in the airlock. This translates into the following piece of code:

Algorithm 4.2.1: General structure

```
ne := ne + 1;                          } REGISTRATION

P(enter);                              }
nm := nm + 1;                          }
ne := ne − 1;                          } ENTER
if ne = 0                              }
  then V(mutex);                       }
  else V(enter);                       }

P(mutex);                              }
nm := nm − 1;                          }
Critical Section;                      } MUTEX
if nm = 0                              }
  then V(enter);                       }
  else V(mutex);                       }
```

Note how directly the story maps to the code. Initially the front door ($enter$) is open. Then we let processes enter one by one, and if there are more processes waiting ($ne > 0$), we re-open the front door. When nobody is waiting, we close the front door, and open the exit ($mutex$). We then keep on re-opening the exit until everyone has left the airlock ($nm = 0$), at which point we reopen the front door.

4.3 Multiple Models

Up until this point of abstraction, both algorithms are actually the same. However, one more thing is needed. We need to make sure that no two processes concurrently assign to ne. Actually, if we would use an atomic counter, which could also do the decrement-and-test in the block ENTER atomically, we would already be done.

However, since we want to have an algorithm using only semaphores as concurrency technique, we need to ensure that both the increment and the decrement-and-test of ne are protected by the same semaphore, say X, as in algorithm 4.3.1 (for reference, we label the $\mathcal{P}$ operations on X with an index, but they are the same operations).

To see how we need to place the semaphore operations on X, we first consider that we need to ensure freedom from starvation. To this end we look at where starvation can occur. This can appear anywhere where a process can be passed, therefore it can occur anywhere outside the guarded sections. This means that it can only occur on a $\mathcal{P}$ operation at the start of a block.

Algorithm 4.3.1: PLACE OF $\mathcal{P}$ (X)

$\mathcal{P}_1(X); ne := ne+1; \mathcal{V}_1(X);$ } REGISTRATION

...
$\mathcal{P}_2(X);$
...
$ne := ne - 1;$
if $ne = 0$
  then ... $\mathcal{V}_2(X);$
  else ... $\mathcal{V}_2(X);$
...
} ENTER

It is rather easy to see that starvation cannot occur at the start of ENTER, nor at the start of MUTEX. That is, if a process stands at the start of the block ENTER, ENTER wil open until all processes have passed ENTER. This is because as soon as a process has entered the airlock, the entrance of the airlock will stay open until it has passed. The reasoning for the block MUTEX is similar, since ENTER only is opened after all processes between ENTER and MUTEX have passed MUTEX.

Therefore the tricky part is located in the $\mathcal{P}_1(X)$ operation at the start of REGISTRATION.

To ensure that no starvation can occur at this point, we want to make sure that at the moment the $\mathcal{P}_2(X)$ operation is executed for the last time (by the process which sets ne to zero, and subsequently opens MUTEX), no processes are blocked at the $\mathcal{P}_1(X)$ operation. This is ensured by the following two conditions.

Condition 4.1. *Whenever a $\mathcal{V}_2(X)$ operation is executed, no processes are waiting at the $P_2(X)$ operation.*

Condition 4.2. *X is a buffered semaphore.*

Now we have two cases for the last execution of the $\mathcal{P}_2(X)$ operation:

1. The process executing $\mathcal{P}_2(X)$ was blocked on $\mathcal{P}_2(X)$, and subsequently released. But since it cannot have been released by the $\mathcal{V}_2(X)$ (because of condition 4.1), it has been released by the $\mathcal{V}_1(X)$, which means that ne is at least 2. Therefore, this is not the last process.

2. The process executing $\mathcal{P}_2(X)$ last can immediately continue, therefore, no processes are blocked on $\mathcal{P}(X)$, because of condition 4.2.

This shows that no starvation can occur at the $\mathcal{P}_1(X)$ operation. The only problem remaining is how to ensure that condition 4.1 holds. We discuss two solutions to this problem: the algorithms as presented by Morris and Udding. We discuss them separately.

4.4 Morris' Algoritm

Morris chooses to introduce a new semaphore for X, named b. This is placed around both the increment and the decrement-and-check. Because both the $\mathcal{P}_2$ and $\mathcal{V}_2$ operation of the semaphore are placed within the guarded region of *enter*, no process can wait for $\mathcal{P}(b)$ in the block ENTER when $\mathcal{V}(b)$ within ENTER is executed. This is exactly condition 4.1.

Algorithm 4.4.1: MORRIS' ALGORITHM

REGISTRATION:
$\mathcal{P}(b); ne := ne + 1; \mathcal{V}(b);$

ENTER:
$\mathcal{P}(enter);$
$nm := nm + 1;$
$\mathcal{P}(b); ne := ne - 1;$
if $ne = 0$
  then $\mathcal{V}(b); \mathcal{V}(mutex);$
  else $\mathcal{V}(b); \mathcal{V}(enter);$

MUTEX:
$\mathcal{P}(mutex);$
$nm := nm - 1;$
Critical Section;
if $nm = 0$
  then $\mathcal{V}(enter);$
  else $\mathcal{V}(mutex);$

4.5 Udding's Algorithm

Udding solves the problem by using the semaphore *enter* for X. This means that both the increment and the decrement-and-check of ne are guarded by the same semaphore. To ensure that condition 4.1 is also fulfilled, an extra semaphore, *queue*, is placed around the entire ENTER block. This has as a direct consequence that at $\mathcal{V}(enter)$ at the end of the block ENTER, no other process can be blocked on the $\mathcal{P}(enter)$ at the start of the ENTER block (because of mutual exclusion, within the guarded region of *queue*).

Algorithm 4.5.1: UDDING'S ALGORITHM

REGISTRATION:
$\mathcal{P}(enter); ne := ne+1; \mathcal{V}(enter);$

ENTER:
$\mathcal{P}(queue);$
$\mathcal{P}(enter);$
$nm := nm + 1;$
$ne := ne - 1;$
if $ne = 0$
  then $\mathcal{V}(mutex);$
  else $\mathcal{V}(enter);$
$\mathcal{V}(queue);$

MUTEX:
$\mathcal{P}(mutex);$
$nm := nm - 1;$
Critical Section;
if $nm = 0$
  then $\mathcal{V}(enter);$
  else $\mathcal{V}(mutex);$

4.6 Switching between Open and Closed

As seen, Morris' algorithm and Udding's algorithm are very similar. This is also apparent in the proof, which for both algorithms more or less follows along the

same lines. The choice of implementation to solve individual starvation at the start of REGISTRATION does lead to large differences in the proof of freedom from individual starvation. Where this is straightforward for the algorithm of Udding, we need to use ghost variables to prove this for the algorithm of Morris. This follows from how is seen which block is open. To determine this, we introduce two predicates: pred-enter and pred-mutex. These can be seen as predictors for which block is about to open: if pred-enter is true, ENTER will be the next block to open, or it is already open. If pred-mutex is true, MUTEX will be the next to open, or a process is already in block MUTEX. Because the predicates become true at a certain moment, we need to make this description somewhat more precise:

(4.1) pred-enter $\Rightarrow mutex = 0$;

(4.2) pred-mutex $\Rightarrow enter = 0$;

Note that, since $enter + mutex$ is a split binary semaphore, we also know that pred-enter $\Leftrightarrow \neg$pred-mutex.

Because the two algorithms do differ, they have (subtly) different predicates. The predicates for the algorithm by Udding follow in a straightforward manner from the algorithm:

(pred-enter) $ne > 0 \vee nm = 0$

(pred-mutex) $ne = 0 \wedge nm > 0$

One would expect these to be more or less the same for the algorithm of Morris, however, due to placement of the $\mathcal{V}$ operations we run into a little trouble here. If $ne = 0$ in ENTER the process will enter the first branch of the **if** -statement and execute $\mathcal{V}(b)$. After this, another process might again increase ne, therefore ensuring '$ne > 0$', however the first process will still be executing $\mathcal{V}(mutex)$, thereby invalidating our assertion that pred-enter $\Rightarrow mutex = 0$. The problem is clear: we are interested in the value of ne at the moment the **if** was executed.

To solve this problem, we introduce the ghost variable ne', which is exactly that. We call this variable a ghost variable since it does not really exist in the algorithm itself, and is only needed for the proof. Therefore, when we write out an execution, we can consider assignments to ghost variables to be executed atomically, that is, no other process can intervene. We can even consider one normal statement together with a list of assignments to ghost variables atomic.

We now propose the following code:

Algorithm 4.6.1: Morris' Algorithm (2)

```
P(b); ne := ne + 1; V(b);                          } REGISTRATION

P(enter);                                          }
nm := nm + 1;                                      }
P(b); ne := ne − 1; ne' = ne;                      } ENTER
if ne = 0                                          }
  then V(b); V(mutex);                             }
  else V(b); V(enter);                             }

P(mutex);                                          }
nm := nm − 1;                                      }
Critical Section;                                  } MUTEX
if nm = 0                                          }
  then V(enter);                                   }
  else V(mutex);                                   }
```

with the following predicates:

(pred-enter) $ne' > 0 \vee nm = 0$

(pred-mutex) $ne' = 0 \wedge nm > 0$

However, this introduces a new problem. Now pred-mutex becomes temporarily true if nm is increased from zero to one while ne' is equal to zero, even though ne itself might be very large. Now pred-mutex will be true until ne' is set to ne, at which point pred-enter will be true again. Though not technically wrong, this erroneous switching of predicate makes the proof harder, and therefore we introduce nm' which will only be increased at the moment ne' is set:

Algorithm 4.6.2: MORRIS' ALGORITHM (3)

```
P(b); ne := ne + 1; V(b);                                  } REGISTRATION

P(enter);                                                  }
nm := nm + 1;                                              }
P(b); ne := ne − 1; ne' = ne; nm' = nm;                    } ENTER
if ne = 0                                                  }
  then V(b); V(mutex);                                     }
  else V(b); V(enter);                                     }

P(mutex);                                                  }
nm := nm − 1; nm' = nm;                                    }
Critical Section;                                          } MUTEX
if nm = 0                                                  }
  then V(enter);                                           }
  else V(mutex);                                           }
```

After which we can fully express pred-enter and pred-mutex in ghost variables:

(pred-enter) $ne' > 0 \vee nm' = 0$

(pred-mutex) $ne' = 0 \wedge nm' > 0$

Chapter 5

Implementation of the algorithms

To be able to reason formally about the algorithms we need to make a formal model. The most important difference between this implementation, and the previous presentations is that we make the atomicity explicit, and that we use the more complex semaphore. Each linenumber is executed atomically. To show that assignments are not atomic they are split into two separate statements.

To keep the length of the listings manageable, we have not written out the definition of $\mathcal{V}$ for the buffered semaphore. We use the following definition (as described in section 3.3.2.3) of the $\mathcal{V}(s)$ operation:

Algorithm 5.0.3: BUFFERED SEMAPHORE

procedure $\mathcal{V}(s)$
40 **if** $blocked_s = \emptyset$
then $s := s + 1$;
else extract some q **from** $blocked_s$;

Since the $\mathcal{P}$ operation of the buffered semaphore has two different states, spread over two different lines, we indicate those two states splitting the $\mathcal{P}$ of a strong semaphore into two operations: $\mathcal{P}$ and $\mathcal{P}_2$. For each algorithm we indicate which semaphores are weak, and which is the buffered semaphore.

5.1 Udding's Algorithm

In Udding's Algorithm *enter* is the buffered semaphore, and *queue* and *mutex* are weak semaphores.

The initialization is as follows: semaphores *enter* and *queue* are initialized to 1, all processes are at line 10, so $ne = nm = 0$ and $blocked_{enter} = \emptyset$.

The algorithm as presented in Algorithm 5.1.1 follows the control structure as presented by Udding in his paper. This means that the conditions for the **if** statement is the inverse of the conditions in the algorithm by Morris; and the branches are of course reversed.

To be able to reason formally about the guarded regions, we formally define those by defining the set of processes which are in the respective guarded regions. In these definitions we introduce the function *pc* which returns the program counter for a given process.

Algorithm 5.1.1: UDDING'S ALGORITHM

```
10  P(enter) : if enter > 0                                   }
      then enter := enter − 1; goto 12;                       }
      else blocked_enter += self;                             }
11  P_2(enter) : await self ∉ blocked_enter;                  } REGISTRATION
12  tmp := ne;                                                }
13  ne := tmp + 1;                                            }
14  V(enter);                                                 }

20  P(queue);                                                 }
21  P(enter) : if enter > 0                                   }
      then enter := enter − 1; goto 23;                       }
      else blocked_enter += self;                             }
22  P_2(enter) : await self ∉ blocked_enter;                  }
23  tmp := nm;                                                }
24  nm := tmp + 1;                                            } ENTER
25  tmp := ne;                                                }
26  ne := tmp − 1;                                            }
27  if ne > 0                                                 }
28    then V(enter);                                          }
29    else V(mutex);                                          }
30  V(queue);                                                 }

40  P(mutex);                                                 }
41  tmp := nm;                                                }
42  nm := tmp − 1;                                            }
43  Critical Section;                                         } MUTEX
    if nm > 0                                                 }
44    then V(mutex);                                          }
45    else V(enter);                                          }
```

$$proc_{enter} = \left\{ p \;\middle|\; \begin{array}{l} (12 \leq pc(p) \leq 14) \vee (23 \leq pc(p) \leq 29) \\ \vee((pc(p) = 11 \vee pc(p) = 22) \wedge p \notin blocked_{enter}) \end{array} \right\}$$

$$proc_{queue} = \{p \mid 22 \leq pc(p) \leq 30\}$$

$$proc_{mutex} = \{p \mid 41 \leq pc(p) \leq 45\}$$

Note that we choose the definition of $proc_{enter}$ in such a way that a process on line 11 or 22 is included as soon as it is released. We also introduce the notation $\#S$ for the number of elements of the set S. With this we present two sets, for which $\#between_{ne} = ne$ and $\#between_{nm} = nm$:

$$between_{ne} = \{p \mid 14 \leq pc(p) \leq 26\}$$

$$between_{nm} = \{p \mid 25 \leq pc(p) \leq 42\}$$

5.2 Morris' Algorithm

Up until now we presented a somewhat different version of the algorithm by Morris from the version presented by Morris in his paper. The reason for this is that we wanted to show the commonalities between Morris' algorithm and Udding's algorithm. From here on we however conform to the version as presented by Morris himself. This means there are a few differences:

- Semaphore *enter* is called a
- Semaphore *mutex* is called m
- variable ne is called na, and ne' is called na'

In Morris' algorithm both a and m are weak semaphores. Semaphore b is a buffered semaphore and use the above-described $\mathcal{V}$. The $\mathcal{P}$ operations are written out.

The initialization is as follows: semaphores b and a are initialized to 1, all processes are at line 10, so $na = na' = nm = nm' = 0$ and $blocked_b = \emptyset$.

There is one more important thing to note about the program. For proof-technical reasons the line numbers are not totally sequential, and line 31 comes before 29 in the listing (but when executing the line numbers are always increasing, since those two lines are from different branches from the **if** -statement). We present the implementation in Algorithm 5.2.1.

This also means the aforementioned predicates are changed, they become:

(pred-enter) $\quad na' > 0 \vee nm' = 0$

(pred-mutex) $\quad na' = 0 \wedge nm' > 0$

Algorithm 5.2.1: Morris' Algorithm

```
10   P(b) : if b > 0                                   }
       then b := b − 1; goto 12;                       }
       else blocked_b += self;                         }
11   P_2(b) : await self ∉ blocked_b;                  } REGISTRATION
12   tmp := na;                                        }
13   na := tmp + 1;                                    }
14   V(b);                                             }

20   P(a);                                             }
21   tmp := nm;                                        }
22   nm := tmp + 1;                                    }
23   P(b) : if b > 0                                   }
       then b := b − 1; goto 25;                       }
       else blocked_b += self;                         }
24   P_2(b) : await self ∉ blocked_b;                  }
25   tmp := na;                                        }
26   na := tmp − 1; na' := tmp − 1; nm' := nm;         } ENTER
27   if na = 0                                         }
       then                                            }
28     V(b);                                           }
31     V(m);                                           }
       else                                            }
29     V(b);                                           }
30     V(a);                                           }

40   P(m);                                             }
41   tmp := nm;                                        }
42   nm := tmp − 1; nm' := tmp − 1;                    }
43   Critical Section;                                 } MUTEX
     if nm = 0                                         }
44     then V(a);                                      }
45     else V(m);                                      }
```

Furthermore we introduce the same sets, as for the algorithm by Udding, to be able to reason about the guarded regions, and about na and nm:

$$proc_b = \left\{ p \;\middle|\; \begin{array}{l} (12 \leq pc(p) \leq 14) \vee (24 \leq pc(p) \leq 29) \\ \vee((pc(p) = 11 \vee pc(p) = 24) \wedge p \notin blocked_b) \end{array} \right\}$$

$$proc_a = \{p \mid 21 \leq pc(p) \leq 31\}$$

$$proc_m = \{p \mid 41 \leq pc(p) \leq 45\}$$

$$between_{ne} = \{p \mid 14 \leq pc(p) \leq 26\}$$

$$between_{nm} = \{p \mid 23 \leq pc(p) \leq 42\}$$

Chapter 6

Concurrent Programs in PVS

Before we present the proof of our algorithm, we first introduce our method of proving statements about concurrent algorithms using the theorem prover PVS. We start gently, by modelling a non-concurrent program in PVS, and formulating one invariant. We then show the extra complications concurrency introduces, and model a simple concurrent program. Finally, we show how one goes about proving invariants and other statements about concurrent programs.

6.1 Model of a Non-Concurrent Program

We start by modelling the following (never-ending) program (initially with i set to zero):

Algorithm 6.1.1: COUNTER

10 $i := i + 1$;
11 **goto** 10;

To be able to prove statements about this program we need to represent it in a mathematical model. To this end we first introduce the concept of a state. The state is a record consisting of all the variables the program uses, including internal ones, like the program counter. In this case the state is of the following type:

```
state : TYPE = [#
                 pc: nat,
                 i : int
                #]
```

We also define the initial state, which we call init:

```
init : state : (# pc := 10, i := 0 #)
```

Next we introduce the step function. This is a boolean function which gets two states x and y as input, and returns *true* if there is a valid step from state x to state y. Since one large step function would get unwieldy for large programs, we create a step function for each line, and we take their disjunction as our step function. For our program, the step function becomes:

```
step10(x,y) : bool = x`pc = 10  AND y = x WITH [`pc = 11, `i := x`i + 1]

step11(x,y) : bool = x`pc = 11  AND y = x WITH [`pc = 10]

step(x,y) : bool = step10(x,y) OR step11(x,y)
```

We have now formally modelled the program. We are now able to define invariants. Invariants are predicates on the state. They should be *true* for the initial state, and when they are *true* in state x and we take a valid step to state y, they should be *true* in state y as well. Note that this implies that invariants are *true* for all reachable states. In this case we could for instance formulate the following invariant:

```
i_geq_zero(x) : bool = x`i >= 0
```

For this invariant we then need to prove the following two theorems:

```
i_gr_zero_init : THEOREM
  i_gr_zero(init)

i_gr_zero_kept_valid : THEOREM
  i_gr_zero(x) AND step(x,y) IMPLIES i_gr_zero(y)
```

6.2 Model of a Concurrent Program

When we introduce concurrency we have to alter the model in various ways. First of all, we cannot assume that an increment is atomic, so our program now becomes:

Algorithm 6.2.1: COUNTER

10 $tmp := i$;
11 $i := tmp + 1$;
12 **goto** 10;

In this program, *tmp* is a process-private variable. Another variable which is now per process is the program counter. For each variable which we have per process instead of program-wide, we need an array of variables.

However, PVS does not support arrays. We therefore emulate an array using functions of a specific type. For instance, we denote an array of three integers as follows:

```
arr : [below[3] -> int]
```

Using this technique, we can now define our state type:

```
process : TYPE = below[Nproc]
state   : TYPE = [#
                   %global variables
                   i : int,

                   %private variables
                   pc : [process -> nat],
                   tmp: [process -> int]
                  #]
```

For readability purposes we first introduced the type `process`, which is a finite subset of the natural numbers (because we choose `NProc` to be an arbitrary, but finite, positive integer). We then use this type to index our arrays of private variables.

Since we represented our arrays by functions, we also need to initialize them as functions. PVS provides us with a syntax for lambda functions, which we can use for this purpose:

```
init : state = (#
                 i  := 0,
                 pc := (LAMBDA (p) : 10),
                 tmp:= (LAMBDA (p) : 0)
                #)
```

We can now define the step function for our concurrent algorithm. Strictly speaking the form of our previous function would suffice. This function would however be horribly complex, since the allowed changes to process-private variables would become very hard to express.

Instead, we add an additional parameter for the process executing the step. This means that the step functions return whether process p can make a valid step, which takes the program from state x to state y. For our example program, we now get the following step functions:

```
step10(p,x,y) : bool =
   pc(p) = 10 AND y = x WITH [‘pc(p) = 11, ‘tmp(p) = i]

step11(p,x,y) : bool =
   pc(p) = 11 AND y = x WITH [‘pc(p) = 12, ‘i = tmp(p) + 1]

step12(p,x,y) : bool =
   pc(p) = 12 AND y = x WITH [‘pc(p) = 10]

step(p,x,y) : bool =
   step10(p,x,y) AND step11(p,x,y) AND step12(p,x,y)
```

Next we can again formulate invariants. But since we now have multiple processes, we can use process parameters in the invariant. However, since this is such a simple example, we cannot actually show any useful invariants. Instead we show the form of some declarations (in which `p` and `q` are processes, and `x` is the state):

```
inv(x) : bool = ...
inv(p,x) : bool = ...
inv(p,q,x) : bool = ...
```

6.3 Proving an Invariant

The process of proving an invariant is quite standard. First one proves the invariant for most steps. Because an invariant is normally kept trivially *true* by most steps, it is enough to use a brute force command like `grind` here.[1] To determine for which steps the invariant can be proved using this method, the easiest way is to prove it for all steps:

```
inv_list : THEOREM
  inv(x) AND step(p,x,y) IMPLIES inv(y)
```

The next step is to execute the proof, and from the unproven branches we can easily deduce which steps were not proved using this method. We can then exclude those steps in the theorem, and by using the `listproof` command[2] we ensure that they are excluded from the proof:

```
inv_list : THEOREM
  inv(x) AND step(p,x,y) IMPLIES inv(y)
```

[1] If one otherwise has to prove a lot of steps separately, one might go for a more sophisticated proof here. However, this is almost never needed.

[2] This is a command which we defined ourselves, and which is discussed in section 6.6.

The only disadvantage of this method is that we need to wait a relatively long time to determine the non-trivial steps, since grind takes a long time, especially with branches which it cannot prove.

Next we create a theorem for each step that we could not prove trivially:[3]

```
inv_A : THEOREM
  inv(x) AND addit_invA(x) AND addit_invB(x) AND stepA(p,x,y)
  IMPLIES inv(y)
```

This is the interesting part of the proof. The human prover needs to choose what additional invariants are needed to prove the invariant for this step. Additionally, he needs to think about the structure of the proof, which is typically non-trivial.

When we have finished these interesting steps, we need to return again to some uninteresting manual labour: we need to formulate the theorem which combines the previous ones into one, thereby proving the invariant for all steps:

```
inv_kept_valid : THEOREM
  inv(x) AND addit_invA(x) AND addit_invB(x) AND addit_invC(x)
  AND ... AND stepA(p,x,y) IMPLIES inv(y)
```

The formulation of this theorem only requires scraping together the additional invariants needed by the various steps. The proof of the theorem is trivial as well: it consists of the invocation of the previous theorems, and a few trivial proof steps like `assert`, `flatten`, and `split`.

After we have proved all invariants, we need to formulate an `iqall`, which is basically a list of all invariants:

```
iqall(x):bool = inv(x) AND inv2(x) AND ...
```

We then need to prove that `iqall` is initially *true* (usually done using `grind`):

```
iqall_init : THEOREM
  iqall(init)
```

Here `init` is the initial state. Next comes a very extensive and time-consuming, but utterly trivial proof: the proof that `iqall` is invariant. This is proved by involving all `kept_valid` theorems, and then initializing **`FORALL`**s from the premise as necessary:[4]

```
iqall_kept_valid : THEOREM
  iqall(x) and step(p,x,y) IMPLIES iqall(y)
```

When we have proved these last two theorems, we have proved that all invariants included in `iqall` are indeed invariant.

6.4 Using Invariants on the End-State

When proving an invariant stays *true*, using another invariant, we normally use the other invariant `otherinv` like this:

```
inv_X : THEOREM
  inv(x) AND other_inv(x) AND stepX(p,x,y) IMPLIES inv(y)
```

However, one might encounter situations where assuming `oth_inv(y)` would be more helpful than assuming `oth_inv(x)`. Of course, assuming the latter will always suffice, since we can simply repeat its own invariance proof for stepX. However, using `oth_inv(y)` might make the proof more elegant.

[3] We can also prove invariance for multiple steps in one theorem, if we know from the context we can prove them in the same way.

[4] This simple explanation does not feature such **`FORALL`**s.

6.4.1 Disadvantages

When proving by hand such a practice would result in a spaghetti proof, much like using backwards gotos leads to spaghetti code.

This means it would be possible to use circlular reasoning. Naturally, this is not possible when proving with a theorem prover, since it would just reject the proof. Much in the same way, a program with backwards gotos could end up in an endless loop, and we would notice this when running the program.

Since we want to prevent such errors, and keep our proofs readable for human beings, we propose some guidelines to ensure some sanity in our proofs.

6.4.2 Guidelines

Instead of using just one monolithic `iqall`, we split this into `iqall1`, `iqall2`, ... Then for `iqall2` we use the following `kept_valid`:

```
iqall2_kept_valid : THEOREM
  iqall2(x) AND iqall1(x) AND iqall1(y) IMPLIES iqall2(y)
```

That way we can only use the invaliant `oth_inv` in the y state when proving invariant `inv`, if `inv` is in a higher `iqall`. In this way we keep our proofs sane, in much the same way as forbidding backwards gotos keeps programs sane.

As a last step we of course still need to define `iqall`:

```
iqall(x):bool = iqall1(x) AND iqall2(x) AND ...
```

The proof of `iqall_kept_valid` follows trivially from invoking `iqall1_kept_valid`, `iqall2_kept_valid`, etc. in order.

6.5 Proving Other Statements

Some statements are harder to prove using invariants. Instead we formulate them as seperate theorems, which assume only `iqall`. We then prove statements from this assumption using a normal proof structure of lemmas and theorems. Examples are the proof of Freedom from Deadlock and Freedom from Starvation, which we present later.

6.6 Interesting Commands we Devised

To make our proofs easier we developed several time-saving proof commands. We discuss them here, and explain what they do, but not show their definition. If one is interested, please refer to the `pvs-strategies` file.

6.6.1 expand-all-steps

This command expands the definition of `step`, and then the definitions of the individual steps, and, if applicable, the associated nextvalues[5]. This allows you to either unpack the whole step-function, or to unpack a single step using only one command.

[5] Instead of defining the whole step at once, we define it by taking the conjunction of the condition on the program counter and the assignment of the nextfunction to the end-state.

6.6.2 expand-only-steps

This commands expands `step` and the individual steps, but not the next-functions. This is useful when a lot of steps are not possible, and we want to do an assume, before expanding the remaining next-functions (very big formulas slow down PVS significantly).

6.6.3 splitgrind and splitgrind+

Splitgrind takes a sequent-number, splits that sequent and then grinds in each branch. Splitgrind+ takes no parameters, but repeatedly splits, then repeatedly flattens and asserts, and finally grinds.

6.6.4 assume

Assume is a replacement for case. Instead of having to prove the assumption, we grind it. This is really only a convenient notation, but it saves us from proving a lot of simple side-proofs.

Chapter 7

Mechanical Proof of Mutual Exclusion

To prove mutual exclusion we first need to prove some invariants regarding the semaphores. We first prove those, and then show how we prove mutual exclusion using those invariants. We describe the proofs of both algorithms simultaneously, as they are very similar.[1]

7.1 Methodology

To prove statements about the algorithm we use invariants. An invariant is a statement which is always true during every execution of the program. We prove that this is the case by proving that all invariants are initially true, and proving that if the conjunction of all invariants is true, and we take any valid step, then the conjunction of all invariants is again true. From this we can then prove various statements.

7.2 Generic Proof

To prove mutual exclusion we first need to prove that the semaphores behave as we would expect. In this case we have binary semaphores and split binary semaphores. For the semaphores we need to prove invariant SEM, and for split binary semaphores we need to prove SSEM.

It is easy to see that those invariants indeed are true. They are initially true, since initially all processes are outside the guarded regions, and the semaphore is 1 (or in case of a split semaphore: one of the semaphores is 1). Furthermore, all steps which change the value of the semaphore also change the number of processes in the guarded section, and keep the invariant valid.

The next step is formulating mutual exclusion:

(MUTEX) $\quad p$ in guarded section $\wedge\, q$ in guarded section $\implies p = q$

It's easy to see that mutual exclusion follows directly from the semaphore invariant, for all guarded sections, since the semaphores are natural numbers. Lastly, mutual exclusion in the critical section follows since the critical section is embedded in the guarded section.

[1] This chapter and the following chapters only give an overview for the proof. They provide the essence of the proof, but don't go into all the details needed for an automated proof. For the full proof, read the PVS source with the appendices A and B and these chapters to guide you through the PVS proof.

In the following two sections we show the invariants used for the two algoritms.

7.3 Udding's Algorithm

Based on the general description for the proof we can now formulate the specific invariants needed for the algoritm by Udding.

(U:S1) $$queue + \#proc_{queue} = 1$$

(U:S2) $$enter + mutex + \#proc_{enter} + \#proc_{mutex} = 1$$

If all the semaphores were regular semaphores this would be directly provable, but since *enter* is not, and due to the definition of $proc_{enter}$ we need an extra invariant to prove that invariant U:S2 stays true when $\mathcal{V}(enter)$ is performed:

(U:B1) $$p \in blocked_{enter} \Longrightarrow pc(p) = 11 \vee pc(p) = 22$$

From U:S1 and U:S2 we can directly derive the needed mutual exclusion invariants:

(U:M1) $$p \in proc_{queue} \wedge q \in proc_{queue} \Longrightarrow p = q$$

(U:M2) $$\begin{array}{c}(p \in proc_{enter} \vee p \in proc_{mutex}) \\ \wedge (q \in proc_{enter} \vee q \in proc_{mutex}) \\ \Longrightarrow p = q\end{array}$$

7.4 Morris' Algorithm

In the same way we derive the following invariants for the algorithm by Morris

(M:S1) $$b + \#proc_b = 1$$

(M:S2) $$a + m + \#proc_a + \#proc_m = 1$$

Which in their turn need the following additional invariant:

(M:B1) $$p \in blocked_b \Longrightarrow pc(p) = 11 \vee pc(p) = 24$$

Then we can again directly prove the mutual exclusion invariants:

(M:M1) $$p \in proc_b \wedge q \in proc_b \Longrightarrow p = q$$

(M:M2) $$\begin{array}{c}(p \in proc_a \vee p \in proc_m) \\ \wedge (q \in proc_a \vee q \in proc_m) \\ \Longrightarrow p = q\end{array}$$

Chapter 8

Mechanical Proof of Freedom from Deadlock

In this chapter we give the proof of Freedom from Deadlock for both algorithms. First we give a general overview of the proof, and then we go into specifics for both algorithms separately.

8.1 General Idea of the Proof

First note that deadlock can only occur if all processes are blocked on a $\mathcal{P}$ operation. Then, we can refine this case to the different possible configurations of how the processes are spread over the several $\mathcal{P}$ operations. Next we can quickly deduce from invariants that deadlock is not possible. However, the details for Morris' algorithm and Udding's algorithm are quite different and we therefore deal with them separately.

Since we use a lot of invariants in this chapter, and the following, we have provided you with appendices A and B which list all invariants used in the PVS proofs. We do not introduce all of them in the text, since some are very trivial. We also provide a little pointer in the margin to where the invariant was defined, so you can easily look up the place where the invariant was first introduced.

8.2 Udding's Algorithm

We start by proving that if there is deadlock, then all processes need to be blocked on a $\mathcal{P}$ operation:

Theorem 8.1. *If there is deadlock, all processes are on line 11, 20, 22, or 40.*

Proof. This is of course true, since otherwise one of the processes could make a step, and then there would be no deadlock. We prove this by assuming that each process has a program counter which is set to a valid line of the program, and we then prove that the program counter must be equal to one of the lines where a process is blocked on or about to execute a $\mathcal{P}$ operation (lines 11, 20, 22, and 40). This is trivial, because if there would be a process on another line, it would be able to make a step. □

Next we prove that it is not possible in case of deadlock that there are both processes blocked on *mutex* and on *enter*. Either all processes are blocked on a $\mathcal{P}$ operation on *mutex* or *queue*, or all processes are blocked on a $\mathcal{P}$ operation on *enter* or *queue*:

Theorem 8.2. *If there is deadlock, and all processes are on line 11, 20, 22, or 40, then either all processes are on line 20 or 40, or all processes are on line 11, 20, or 22.*

U:S2 p.36

Proof. This is easily seen to be true, when we look at invariant U:S2: if all processes are blocked on a $\mathcal{P}$ operation, then both $\#proc_{enter}$ and $\#proc_{mutex}$ are zero, and therefore either *enter* or *mutex* is 1.[1] However, we do need to prove that if all processes on line 11 and 22 are blocked (if they all are in $blocked_{enter}$) that then *enter* must be 0, and therefore *mutex* is 1. To this end we use the following invariant:

(U:B2) $$pc(p) = 11 \vee pc(p) = 22 \Longrightarrow enter = 0$$

The invariant holds because a process only goes to line 11 or 22 if *enter* is zero. Furthermore the $\mathcal{V}(enter)$ never releases a process from one of those lines as long as a process is waiting, and does not increase *enter* until no process is on either of those two lines, and therefore does not invalidate the invariant either. □

We now look at the two possible cases of distribution of processes over the $\mathcal{P}$ statements, and prove that in neither case we can get deadlock.

8.2.1 No Deadlock on Enter

The first case is that all processes are blocked on either line 11, 20 or 22. We first prove that if there are processes blocked on line 20, then there must necessarily be a process blocked on line 22, or there is no deadlock:

Theorem 8.3. *If there is deadlock, and there is a process on line 20, there must be a process on line 22.*

Proof. This is true because a process can only be blocked on line 20 if there are processes in the guarded section of *queue*, and a process in the guarded section of *queue* is either on line 22, or is able to make a step (in which case there is no deadlock). □

From this we can directly deduce the following corollary:

Corollary 8.4. *If all processes are on line 20, there is no deadlock.*

Lastly we prove that if we have deadlock and all processes are on line 11, 20 and 22, then necessarily all processes must be on line 20 (that is: deadlock on *enter* is not possible):

Theorem 8.5. *If there is deadlock, and all processes are on line 11, 20 and 22, then all processes are on line 20.*

Proof. We first note that *nm* must be zero, since otherwise processes must have increased *nm* but not yet decreased it, and therefore be between lines 25 and 42. However, all processes are on lines 11, 20 and 22, so this cannot be. To formally express this meaning of *nm* we introduce the following invariant:

(U:NM1) $$nm = \#between_{nm}$$

Next we use the following invariant to show that then *mutex* must also be zero:

(U:NM2) $$nm = 0 \Longrightarrow mutex = 0$$

Next we can assume that there is a process on line 11 or 22 (since otherwise we would already be done). This means that *enter* is zero, again using U:B2. Since we also now that $\#proc_{mutex}$ is zero (because all the processes are on line 11, 20 or 22), and *mutex* is zero, this must mean that $\#proc_{enter}$ is 1 (using U:S2), and therefore one of the processes on line 11 or 22 must be in $proc_{enter}$ and therefore not in $blocked_{enter}$ and thus able to continue. Therefore there is no deadlock in this case. □

U:B2 p.38

U:S2 p.36

Since we now have proved that there can be no deadlock in this case, we cannot have deadlock on semaphores *enter* and *queue*.

8.2.2 No Deadlock on Mutex

We now look at the case that all processes are blocked on either line 20 or line 40. We now first prove that in this case all processes must be blocked on line 40:

Theorem 8.6. *If there is deadlock, and all processes are on line 20 or 40, all processes are on line 40.*

Proof. This follows directly from theorem 8.3: if a process is blocked on line 20, then necessarily a process is blocked on line 22, which leads to a contradiction, since alle processes were blocked on either line 20 or line 40. □

Lastly, we need to prove that if all processes are blocked on line 40 we cannot have deadlock:

Theorem 8.7. *If all processes are on line 40, we cannot have deadlock*

Proof. To this end we need to prove that $mutex = 1$, which is equal to proving that $\#proc_{enter} = 0$, $\#proc_{mutex} = 0$ and $enter = 0$ (using U:S2). The first two are trivial, but for the last we need the following invariant:

U:S2 p.36

(U:O1) $$ne = 0 \wedge nm > 0 \implies enter = 0$$

To prove that the premise of this invariant is true we use U:NM1, and a similar invariant for *ne*:

U:NM1 p.38

(U:NE2) $$ne = \#between_{ne}$$

Since we know that $between_{ne}$ is empty, and $between_{nm}$ cannot be empty, it directly follows that *enter* must be zero. □

This means that we cannot get deadlock in the second case either, and therefore the algoritm is free from deadlock:

Theorem 8.8. *The algoritm is free from deadlock.*

Proof. Using Theorem 8.1 we know that we can only get deadlock if all processes are waiting for a $\mathcal{P}$ operation. Using theorem 8.2 we know that then either all processes are blocked on *enter* and *queue* or on *mutex* and *queue*. Using corollary 8.4 and theorem 8.5 we see that there cannot be deadlock in the first case. Using theorem 8.6 we know that in the second case all processes must be blocked on enter, and from theorem 8.7 we now we cannot have deadlock in that case. □

[1] And if either *enter* or *mutex* is 1, there cannot be processes blocked on both of them.

8.3 Morris' Algorithm

We present the proof of the algorithm by Morris. Since this is very similar to the proof of the algorithm by Udding, we go into less detail for the proofs of the theorems:

Theorem 8.9. *If there is deadlock, all processes are on line 11, 20, 24, or 40.*

Proof. This is true, since processes on all other lines can always make a step. □

Now that we narrowed it down to a few lines we are able to exclude two of them:

Theorem 8.10. *If there is a process on line 11 or 24, there cannot be deadlock.*

Proof. This follows from the fact that if there is a process on line 11 or 24, we know that $b = 0$, meaning there is a process in the guarded section (either on line 11, 24, or another line). This follows directly from invariant M:S1 and invariant M:B2:

M:S1 p.36

$$\text{(M:B2)} \qquad pc(p) = 11 \vee pc(p) = 24 \Longrightarrow b = 0$$

This invariant follows trivially for almost all steps, and can easily be shown to be kept true for the $\mathcal{V}$ operations as well, using invariant M:M1. □

M:M1 p.36

We now only need to consider the case that all processes are on line 20 and 40:

Theorem 8.11. *If there is deadlock, and all processes are on line 20 or 40, then all processes are on line 40, or all processes are on line 20.*

Proof. This follows directly from the fact that $a + m$ is split binary semaphore: since there are no processes in the guarded section, either a or m is 1, thus allowing processes to pass. □

Next we consider both cases: all processes are on line 20, or all processes are on line 40.

Theorem 8.12. *If all processes are at line 20, there cannot be deadlock.*

Proof. From invariant M:NM2 we know that nm is the number of processes which are between lines 23 and 42. This means that in this case $nm = 0$. Then we can use the following invariant to deduce that m must be 0:

M:NM2 p.71

$$\text{(M:O2)} \qquad nm = 0 \Longrightarrow m = 0$$

If m is zero, this must mean there is a process in a guarded section or $a = 1$. Since there cannot be a process in the guarded section, because all processes are at line 20, we know that $a = 1$, and therefore a process can make a step. □

We now have two cases left, which we treat separately:

Theorem 8.13. *If all processes are at line 40, there cannot be deadlock.*

Proof. This is proved in much the same way as the previous proof. First we prove the invariant M:O1:

$$\text{(M:O1)} \qquad na = 0 \wedge nm > 0) \Longrightarrow a = 0$$

To prove that the condition of this invariant is true we use invariant M:NM2 to prove that $nm > 0$ and invariant M:NA2 to prove that $na = 0$. □

M:NM2 p.71
M:NA2 p.70

We have proved every single case, and combine this into one theorem:

Theorem 8.14. *The algorithm is free from deadlock.*

Proof. We know from theorem 8.9 that, if there is deadlock, all processes are on lines 11, 20, 24 and 40. However, there cannot be processes on lines 11 or 24, as follows directly from theorem 8.10. This means that all processes are on line 20 or 40. Using theorem 8.11 we know this means there are two possible cases. The first case is that all processes are on line 20, in which case we cannot have deadlock according to theorem 8.12. The second case is that all processes are on line 40, which is a deadlock free situation as well, according to theorem 8.13. Therefore the algorithm by Morris is deadlock free. □

8.4 Formalization of the Proof

To be able to formalize these proofs we define the new proposition $hasNext(x)$, which states that from state x the algorithm is able to make a valid step. The proposition "the algorithm is deadlock free" now becomes:

$$\text{iqall}(x) \implies hasNext(x)$$

Here iqall is the conjunction of all invariants. Notice that this means that each reachable state has a valid next step, since for each reachable state the invariants are all true.

There is however one more important remark. We want a process *in* the algorithm to make a valid step. This means we need to exclude step 10 (otherwise, every algorithm would be free from deadlock, as long as one process stays on line 10). Of course, that also means that we have to consider the case that all processes are on line 10. In that case $hasNext(x)$ will necessarily be false. Therefore the proposition we prove becomes:

$$\text{iqall}(x) \implies hasNext(x) \vee \forall p : p \text{ at } 10$$

This of course means that in each theorem proved above, there is a clause about processes on line 10. We have chosen not to mention this explicitly in the text, since it and only complicates reading and makes no difference to the structure of the proof.

Chapter 9

Mechanical Proof of Freedom from Individual Starvation

To prove Freedom from Individual Starvation, we introduce a variant function. Using this function we prove that no individual process can be left behind infinitely long.

9.1 Participation

To prove Freedom from Individual Starvation we first need to define when we want to guarantee Freedom from Individual Starvation. Since we only concern ourselves with the presented algorithms, we only care about freedom from individual starvation in the algorithm, which comprises the *entry* and *exit* section. Therefore we can ignore the posibility of starvation in the critical section, by assuming it to be one atomical step. Lastly, since we do not care about the non-critical section, we only guarantee freedom from individual starvation when a process is participating. A process is participating when it is in either *entry*, CS, or *exit*. For the presented algorithms this means that a process is participating if it is on a valid line, unequal to 10.

9.2 Proof Using a Variant Function

The proof of freedom from individual starvation using a variant function works as follows. We define a function from a state and a process to a natural number. Then we prove that whenever the process participates in the algorithm any step, *made by any process*, decreases the value of the function. This means that zero is reached in a finite number of steps, and because the variant function is larger or equal to zero (since it is a natural number), there can only be a finite number of steps made during the participation of the process. Since we know that the algorithm is free from deadlock, the process must necessarily go through the algorithm in finite time. Since we prove this for an arbitrary process, individual starvation is not possible.

9.3 Proof Obligation

Now that we have shown how a proof using a variant function works we can make precise what we are going to prove. We only prove that the variant function decreases. We do not need to show that it is larger than zero, because we enforce

this by giving it the type constraint that it has to have natural number as value. This leaves us with only the following proof obligation:[1]

$$participating(q, x) \wedge iqall(x) \wedge step(p, x, y) \Longrightarrow \mathrm{VF}(q, x) > \mathrm{VF}(q, y)$$

We have formalized the concept of q participating with the function *participating*, from process and state to boolean. The *step* function indicates that process p makes a step from state x to state y.

9.4 Choosing the Variant Function

It now comes down to a good choice of variant function. Perhaps the most obvious candidate for the variant function is the exact (worst case) number of steps which can be taken until the process has left the algorithm. Because this can lead to complex case distinctions, we settle for a variant function that has a somewhat larger value, but is easier to formulate.

We define a variant function for a process q which is participating. To construct the variant function we consider the number of steps an arbitrary process p can take, until q reaches the end of the algorithm.

Consider the situation that q enters the algorithm at the moment pred-mutex is just made true. Now all processes can finish the algorithm before q gets to pass even block ENTER. After that, all the processes can join q in its iteration of the algorithm, overtake q, finish the algorithm again, and get to the first $\mathcal{P}(enter)$ before q leaves the algorithm.

An upper limit is therefore the number of steps to the first $\mathcal{P}(enter)$, plus the number of steps to make a full iteration of the algorithm. We choose this to be our variant function. The tricky part is now to define where in the iteration a process is.

9.5 Phases of the Algorithm

To exactly define the variant function we need a way to define where each process is. Since every process can make a full iteration, and then continue to $\mathcal{P}(enter)$, we know that for most of the lines of the algoritm a process can be in two places on this path. To understand this concept better we look at the different phases. We reason from the aforementioned situation, and later we look at the differences between this and the other cases.

In the aforementioned situation, process q enters the algorithm at the moment that ENTER is already closed, and MUTEX is open. We call this phase 1. Phase 1 ends (more or less) at the moment ENTER gets opened again. At this point phase 2 starts. All participating processes (q included) now move towards the start of block MUTEX. When (or at least around the time) all processes have arrived here, phase 3 starts.

We consider the some other situations. First we look at the situation where q has to wait less. This means that the other processes are not in the block MUTEX at the moment q enters. Therefore all processes (possibly zero) are in block ENTER or block REGISTRATION, and we are therefore now in phase 2. Note that it is not possible to enter in phase 3 (since then process q would need to be at the start of block MUTEX).

Lastly we look at the worst case. This is a little bit worse than the situation we first considered. Instead of arriving just after pred-mutex has been made

[1] In the proof of de decrease of the variant function we do not actually prove the inequality $\mathrm{VF}(q, x) > \mathrm{VF}(q, y)$. Instead we formulate a stronger equality ($\mathrm{VF}(q, x) = \mathrm{VF}(q, y) + a$, with a either 1 or 2) which we prove, and then deduce the inequality from that. This is because PVS cannot reason as well with inequalities as it can with equalities.

true, we consider the situation where pred-mutex is just about to be made true. This is the case when q enters just after some process p has executed the second $\mathcal{P}(X)$ (which is $\mathcal{P}(enter)$ in the case of Udding, and $P(b)$ in the case of Morris), but has not yet made pred-mutex true. To be able to make pred-mutex true, this also means that there is no other process in between q and p, otherwise we would be in phase 2. We call this new situation of entering phase 0, and from here on we treat it the same as phase 1.

We can now precisely define each phase by whether pred-enter or pred-mutex is true, and the place of q in the algoritm:

Phase	Predicate	Place of q in Udding	Place of q in Morris
Phase 0	pred-enter[2]	$pc = 11$	$pc = 11$
Phase 1	pred-mutex	$pc = 11$	$11 \leq pc \leq 20$
Phase 2	pred-enter	$11 \leq pc \leq 40$	$11 \leq pc \leq 40$
Phase 3	pred-mutex	$40 \leq pc$	$40 \leq pc$
	pred-enter	$40 < pc$	$40 < pc$

9.6 The Variant Function

To create the actual variant function we now need to define the value of the variant function for each possible phase. To this end we propose the following variant function for the algorithm by Udding:

$$d(pc,q,x) = \begin{cases} \left.\begin{matrix} phase0(q,x) \\ phase1(q,x) \end{matrix}\right\} & : d_{to\_11}(pc) + d_{full\_code} \\ phase2(q,x) & : d_{to\_40}(pc) + d_{40\_to\_11} \\ phase3(q,x) & : d_{to\_11}(pc) \end{cases}$$

$$\mathrm{VF}(q,x) = \sum_{pc} \#proc_{at}(pc) * d(pc,q,x)$$

The specific distance functions are exactly what one would expect: $d_{to\_11}(pc)$ is the (maximum) number of steps from pc to line 11, $d_{to\_40}(pc)$ is the (maximum) number of steps from pc to line 40, d_{full_code} is the number of steps it takes to make a full iteration through the code, and end up on the same line again, and lastly, $d_{40\_to\_11}$ is the number of steps needed to go from line 40 to line 11.

For the algorithm by Morris we use the same VF, but another distance function d:

$$d(pc,q,x) = \begin{cases} \left.\begin{matrix} phase0(q,x) \\ phase1(q,x) \end{matrix}\right\} & : d_{to\_20}(pc) + d_{full\_code} \\ phase2(q,x) & : d_{to\_40}(pc) + d_{40\_to\_20} \\ phase3(q,x) & : d_{to\_20}(pc) \end{cases}$$

Again, the specific distance functions are what one would expect: $d_{to\_20}(pc)$ is the number of steps from pc to line 20, $d_{to\_40}(pc)$ the number of steps to line 40, d_{full_code} is the number of steps for a full iteration, and lastly, $d_{40\_to\_20}$ is the number of steps from line 40 to line 20.

9.7 Proof

To prove that individual starvation is not possible we need to prove two things. First we need to prove that the variant function is larger than or equal to zero.

[2] In addition to pred-enter we also need the condition specified in the preceding text: there is a process on line 22 and not in $blocked_{enter}$ or between lines 23 and 26. Furthermore we require $ne = 1$.

The variant function is a sum of non-negative terms. Each term is positive because it is a function with only non-negative values, multiplied by the number of elements in a set, which is also necessarily non-negative.

Second, we need to prove that the variant function is strictly decreasing. Since this proof is very extensive, we prove this in a separate chapter.

Chapter 10

The Variant Function is Decreasing

The proof that the variant function is decreasing is rather straightforward for most steps. If the phase does not change, we only need to prove the distance function has a higher value for the line the process comes from, than for the one where it ends up. The harder part is proving that the variant function also decreases when a phase change is made. This means we need to prove that the phase can only change to a higher phase.

10.1 Structure of the Proof

The proof can be divided in two parts: the easy part and the hard part.

10.1.1 The Easy Part

Even though the easy part is not complex, it does take a fair number of steps. We start off by proving that the distance function does not change for most of the steps. Next we show that for each step the variant function does indeed decrease for these steps. For most steps this is straightforward, since the distance functions have been chosen in such a way that they are decreasing for each step except for one. This means we do need to show that the increasing step is not possible for the phase in which this distance function is used.

10.1.2 The Hard Part

The hard part is proving that the variant function decreases for the two steps that can change the state. We do this by showing that invalid phase changes are impossible, and by showing that the distance function does not increase by permitted phase changes. Since the phase-changing steps are decreasing in every distance function, we have no additional complications here.

10.2 Function d is Invariant Over Most Steps

To prove that d is invariant over most steps we need to prove that the different phases are invariant over those steps. But first we state what "most steps" constitutes. For Udding these are all steps, except for step 26 and 42. For

Morris those are all steps, except for step 26, and for step 42 when not in phase3.[1]

Since most steps obviously keep most phases invariant, they are also easy to prove with a theorem prover. Therefore, the only interesting proofs are the ones of the steps for which invariance is less obvious. To be able to discuss their proofs, we first make our proof obligations more formal. We want to prove:

$$stepX(p, x, y) \Longrightarrow phaseY(q, x) = phaseY(q, y)$$

Here p is the process which takes the step, thereby taking the state of the program from x to y. As before, q is the process for which the variant function is calculated, and phaseY is any phase, and stepX is any step. To prove this equality we divide it into two implications, which are easier to prove with a theorem prover:

$$\begin{aligned} stepX(p, x, y) \wedge phaseY(q, x) &\Longrightarrow phaseY(q, y) \\ stepX(p, x, y) \wedge phaseY(q, y) &\Longrightarrow phaseY(q, x) \end{aligned}$$

It would seem that the second implication is not needed, but it makes later proofs easier.

To prove that d is invariant over most steps we need to prove that all the phases are invariant. However, the proofs are not very interesting, and are in a way very typical interactive theorem proving: instead of thinking up beforehand how one goes about proving it, one proves each non-trivial step on an ad-hoc basis by looking at what is required to prove. Though the reasoning is not trivial, it is not really interesting either. We show the proof for the algorithm by Udding as example. The proof for the algorithm by Morris is done in a similar fashion, but of course the reasoning and used invariants are different.

Note that we do not need to prove invariance of phase3, because we can formulate phase3 as not phase0, not phase1, and not phase2. In the formalized proof we also have defined phase3 this way. This means that if phase0, phase1, and phase2 are invariant, so is phase3.

The most interesting thing to remark about the proof is that this is the place where pred-enter and pred-mutex come into play. Therefore we first prove some invariants regarding these predicates.

10.2.1 Invariants Using pred-enter and pred-mutex

Based on the implications we demanded to be true for the predicates in chapter 4, we can now derive several invariants:

(U:P1) $\quad \text{pred-enter} \Longrightarrow mutex = 0$

(U:P2) $\quad \text{pred-enter} \Longrightarrow (\forall p : p \in proc_{mutex} \Rightarrow pc(p) \geq 43)$

(U:P3) $\quad \text{pred-mutex} \Longrightarrow enter = 0$

(U:P4) $\quad \text{pred-mutex} \Longrightarrow (\forall p : p \in proc_{enter} \Rightarrow pc(p) = 27 \vee pc(p) = 29)$

(U:TMP6) $\quad p \in proc_{mutex} \Longrightarrow ne = 0$

Since we have constructed pred-enter and pred-mutex is such a way that invariants U:P1 and U:P3 are true, we do not show additional proof here. The invariants U:P2 and U:P4 are their logical consequences. They respectively state (more or less) that when pred-enter is true, block MUTEX is empty, and that

[1] We made this choice for the algorithm by Morris to make the hard part of the proof smaller. However, the effort required was much larger than the effort saved. Therefore we excepted the entire step 42 in Udding.

when pred-mutex is true, block ENTER is empty. However, we have to account for the fact that pred-enter is made true in one of the last lines of block MUTEX, and pred-mutex is made true in one of the last lines of block ENTER. Therefore the range of lines which are guaranteed to be free of processes is a bit smaller than the entire block. The proof is relatively simple, as invariant U:P2 follows almost directly from invariant U:P1, and invariant U:P4 from U:P3. Finally, invariant U:TMP6 is needed to prove invariant U:P1, and vice versa.[2]

10.2.2 Phase0 is Invariant Over Most Steps

To be able to read the proof more easily, we first provide the full definition of phase0:

$$\text{pred-enter} \wedge pc(q) = 11 \wedge ne = 1 \wedge$$
$$(\exists p : (pc(p) = 22 \wedge p \notin blocked_{enter}) \vee (23 \leq pc(p) \leq 26))$$

Because of mutual exclusion on the split semaphore $enter + mutex$, and the demand that there is a process in the first part of the guarded section of *enter* in ENTER, most of the steps are not possible, and therefore invariance is easy to prove using invariant U:M2. The steps which we have proved using another method are steps 11, 21, 14, 28 and 45.

U:M2 p.36

Step 11

For step 11 we use invariant U:M2 as well, to show that that step cannot be made, since the process making that step must still be in $blocked_{enter}$. Next we have to show that if phase0 is true in y it is as well in state x. But this is trivially true: this step changes no variables, the process executing cannot be q (since q is on line 11 in y), nor can this step change the place of a process which is not on line 11 in state x.

U:M2 p.36

Step 21

If a process made step 21, and phase0 is true in x, that would mean that ne is at least 2, which is a contradiction.

If phase0 is true in y we know that *enter* is zero in state x, from invariant U:B2. We therefore know that the process wil start to block on the semaphore *enter*. Now we show that both the process making the step and the process which is in the start of ENTER by the condition of phase0 are in the set $between_{ne}$ which means that ne is at least two (which we show using invariant U:NE2), and therefore phase0 cannot be true in x, nor can it in y, because the value of ne is not changed by step 21.

U:B2 p.38

U:NE2 p.39

Step 14

For step 14 we use the same reasoning to show that ne must be larger than two if phase0 is true, and therefore phase0 cannot be true in either x or y.

Step 28

The reason step 28 is interesting is that the $\mathcal{V}$ operation could release the process which is about to close block ENTER. Because we cannot use our $ne \geq 2$ argument here, we opt for using our first method using mutual exclusion, but this time using U:M1, because both this process as well as the process which

U:M1 p.36

[2] This seems to be circular reasoning. However, we only require that all invariants in the y state (iqall(y)) follow from all invariants in the x state (iqall(x)).

might be about to close block ENTER are within the guarded region of *queue*, which is clearly impossible.

Step 45

Lastly, step 45 keeps phase0 true in y if it was true in x. The other way around is harder to show (since it could have just released the process which is about to close block ENTER). Therefore we opt for showing this side of the proof by showing that ne must be zero, and therefore, there cannot be a process blocked on line 22. We do this by using invariant U:TMP6. After that we again use invariant U:NE2 to show that if $ne = 0$ there cannot be a process on line 22.

U:TMP6 p.48

U:NE2 p.39

10.2.3 Phase1 is Invariant Over Most Steps

As for phase0, most steps are again quite trivial to prove, even without using any invariant. We only discuss the steps which we proved separately: 11, 13, and 24. For your convenience, we provide the full definition of phase1:

$$\text{pred-mutex} \wedge pc(q) = 11$$

Step 11

Step 11 is not hard to prove. The only complication is that we require that process q stays on line 11, so we need to use U:P4, which shows that this is actually an impossible step to take.

U:P4 p.48

Step 13

We first show that if phase1 is true is state x, we cannot make step 13, since there are no processes on line 13, by invariant U:P4. For the other side, it is easy to show that if q is on line 11 in state y it was also there in state x. We can also easily prove that if $nm > 0$ in state y, then it also was in state x. The interesting part is ne: if ne is zero in state y we have a contradiction, since it was just increased, and it should always be larger or equal to zero. To show this contradiction we use invariants U:TMP5 and U:NE3.

U:P4 p.48

U:TMP5 p.67

U:NE3 p.66

Step 24

We can simply show that phase1 cannot be true before or after this step, since ne is larger than zero, which we can show by using invariant U:NE2.

U:NE2 p.39

10.2.4 Phase2 is Invariant Over Most Steps

For phase2, again most steps are trivial, and again no additional invariants are needed to prove those steps. The steps which we proved separately are steps 13, 24, and 40. For your convenience, we provide the full definition of phase2:

$$\text{pred-enter} \wedge 11 \leq pc(q) \leq 40$$

Step 13

The only hard part of this step is to show that if $ne > 0$ is true in state y, pred-enter is true in state x. We do this by using that otherwise pred-mutex must be true in state x. However, we know from U:P4 that the process making the step cannot be on line 13 in that case, which is a contradiction with the execution of step 13.

U:P4 p.48

Step 24

To prove that phase2 is kept true if step 24 is executed, we need to show that pred-enter stays true, or was true if it is now true in y. Since pred-enter is a disjunction we have two cases. The only non-trivial case is when $nm = 0$ in x or in y.

If $nm = 0$ in x, we prove ne to be larger than zero using U:NE2. If $nm = 0$ in y, we show that nm had to be smaller than 0 in state x by using invariant U:TMP5. This contradicts invariant U:NE3, which means that nm cannot be zero in y.

U:NE2 p.39
U:TMP5 p.67
U:NE3 p.66

Step 40

It follows directly from invariant U:P1 that this step is not possible when phase2 is true. Because this step cannot make phase2 true either, it is not possible either that phase2 is true in state y but not in state x.

U:P1 p.48

10.3 VF is Decreasing for Most Steps

Now that we have shown that the value of d is invariant, we can show that VF decreases for every step. In the formalized proof this requires many theorems, but the proof is actually very easy. For each step from line a to b we show that $d(b) < d(a)$. Then we proceed to show that the number of processes on line a has decreased by 1, and the number of processes on line b has increased by one. Therefore the total value of the VF must have decreased.

There are two important exceptions, and these are the steps which can increase the value of d in certain phases. For these cases we prove that the fact that the step is taken, must mean that they are in a phase in which d is decreasing for this step. This proof follows immediately, because said step is always a $\mathcal{P}$ step. Depending on the value of pred-enter and pred-mutex, which follows directly from the phase, this step is possible or not.

10.4 VF is Decreasing for Phase-Changing Steps

We still need to show that the variant function is also decreasing for steps 26 and 42. Instead of looking at the whole step, we divide the proof up in subcases for the different phases which can be true in state x.

When proving that one of the potentially phase-changing steps decreases the VF in a certain phase, we encounter two types of proof. First we have the situation where this combination of step and phase is invalid, which means we need to prove the situation to be invalid.

Second, we have the situation where we can take such a step. In that case we also need the phase after the step to prove that the variant function is decreasing. If that does not trivially follow from the phase before the step, the proof for this situation is again divided into proofs for each possible end-phase.

As soon as we know the phase before and after the step we apply the same proof as we did for the "easy" steps. The only difference is that we do not know that d stays the same, so instead of assuming it we have to "unpack" the whole definition of d and show that it is indeed the same.[3]

Because these proofs are all quite similar we only give two examples, one of each type of proof to give a taste for the types of proof. Again, we choose our examples from the proof of the algorithm by Udding.

[3] One would expect we would need to show that it is smaller or equal, but we can indeed show that it is the same. (Of course, the function does increase for some lines, but there are no processes on those lines, which is implied by the step just made.)

10.4.1 Step 26, phase0

If we are in phase0 and we make step 26 we are in phase1. We can easily see this, because q is (and stays) on line 11. Furthermore we knew that ne was 1, and is now decreased to 0 (we need U:TMP5 to show that $ne = tmp$). Next we also know nm to be larger than zero because of the process making the step, so we can show $nm > 0$ using U:NM1. When the phase changes from phase0 to phase 1, the distance function does not change, which means we can show the variant function decreases.

U:TMP5 p.67

U:NM1 p.38

10.4.2 Step 26, phase1

Since phase1 requires $ne = 0$ and the execution of step 26 implies $ne > 0$, as shown by by U:NE2, this step is not possible.

U:NE2 p.39

10.5 Combining the Proofs

We then combine these proofs, we show that regardless of which phase is initially true, the variant function is decreasing. We then combine those results to conclude that for steps 26 and 42 the variant function is decreasing. Since we already knew this for the other steps, we can then conclude that the variant function is indeed decreasing.

Chapter 11

Executions

To compare the algorithms of Udding and Morris to the algorithm of Hesselink and Aravind we take a look at the executions. If the sets of possible executions of the algorithm by Hesselink and Aravind is the same as the set of possible executions of the algorithms by Udding and Morris, we could deem the algorithm similar. If the sets of executions are indeed the same, it might be interesting to see whether it is possible to introduce ghost variables, and represent one algorithm in ghost variables inside the other algorithm. However, this is outside the scope of this document.

Looking for more similarity is in vain in our opinion, since both algorithms use different synchronization primitives, and thus cannot be called equal in any meaningful way.

11.1 Executions

A valid execution of an algorithm is a list of steps which conforms to that algorithm. That is, if one were to execute the algorithm one step at the time, and list the steps, one might end up with said execution. This means that given a certain input, a (non-randomized) single-threaded program has only one possible execution. A multi-threaded program however, can have many possible executions.

Since the algorithm by Udding and the algorithm by Morris are quite different from the algorithm by Hesselink and Aravind, looking at the exact steps will not give any useful insights. Instead we look at the executions at a more abstract level, by considering only the first step of the algorithm and the step which executes the Critical Section.[1]

Let us call the entry step S (for Start) and the Critical Section step C, with added subscript to indicate which process executes the step.

Though we do not formally prove it, it should not be too hard to see from the proofs of freedom from starvation, that the sets of possible executions of the algorithms by Udding and Morris are indeed the same. Therefore we discuss their executions without specifying the algorithm.

We first look at some sample executions. A simple execution is the one where a process goes through the whole algorithm (*entry*, CS, *exit*), without another process arriving. Then a second process does the same. The execution looks like $S_1C_1S_2C_2$. Of course this is not a very interesting case. It starts to

[1] Normally, looking at the exit step is interesting as well. However, for the algoritms of Morris and Udding, as well as for the algorithm by Hesselink and Aravind, this does not seem to be of relevance. However, might one be interested, it should not be to hard to add it.

get interesting when we get multiple processes, say at least three. For instance, a possible legal execution might be:

$$S_1 S_3 C_1 S_2 C_3 C_2$$

However, it is hard to see that this is indeed a legal execution, whereas the following is not:

$$S_1 S_3 C_3 S_2 C_1 C_2$$

To make it more readable we introduce two more steps: the two block-commands. The step which opens block MUTEX, $^{\mathbf{M}}$, and the step which opens ENTER, $^{\mathbf{E}}$. Now the first execution could read like:[2]

$$^{\mathbf{E}}S_1{}^{\mathbf{M}}S_3 C_1 S_2{}^{\mathbf{EM}}C_3 C_2{}^{\mathbf{E}}$$

11.2 Valid Executions

We now want to define what executions are valid. To do this we look at the way the algorithms work. A process can always execute the first step, however, it can only execute the Critical Section when the block MUTEX is open. Moreover a process firsts needs to execute a start step before executing the critical section step. Lastly, since we have freedom from individual starvation, we know that there is an upper time limit between the moment the S_i is executed and the C_i is executed.

We now use our knowledge of the way the blocks are opened to define what is the maximum number of block switches that can occur between a process starting to enter and entering the CS. If $^{\mathbf{E}}$ is the last block-command executed, ENTER is open. If a process enters, it will go through the whole of block enter[3] and complete its critical section after the first $^{\mathbf{M}}$ and before the next $^{\mathbf{E}}$.

If a process enters after an $^{\mathbf{M}}$, it is blocked until the first $^{\mathbf{E}}$ command, then it can pass through block ENTER, and after the next $^{\mathbf{M}}$ but before the next $^{\mathbf{E}}$ it will pass through the critical section. That means that a C_i has to be executed in the next (but never current) $^{\mathbf{M}}{}^{\mathbf{E}}$ block encountered after executing the corresponding S_i. When we combine this with the condition that a process may only enter again after it has executed its critical section, we can now formally define a grammar for valid executions.

11.3 Grammar

To be able to define a grammar we first need to introduce some functions, as normal BNF notation will not do for an infinite alfabet, with the amount of possible permutations we have. This means we first have to define some notation. We discuss those functions as if they nondeterministically give one result. If one wants to read the grammar as a list of production rules, one needs to interpret these functions as giving all possible outcomes in a set, instead of only one random result.

---

[2] Of course, for some executions there can exist multiple corresponding executions with $^{\mathbf{M}}$ and $^{\mathbf{E}}$.

[3] Actually, as we know, there is one exception to this, the same way phase0 was an exception. We circumvent this by saying the $^{\mathbf{M}}$ is placed at the moment it is clear that MUTEX will open (that is, at the $\mathcal{P}_2(X)$ just before the decrement of ne)

11.3.1 Meta Functions

First we introduce some meta functions, which allow us to express more complex grammars than standard BNF notation can.

11.3.1.1 Permute

The permute function gets as input a set S, and then returns a list made of the elements of set S. Every element of set S has to be used exactly once in the list. This means the Permute function returns a random permutation of the elements in set S.

11.3.1.2 Zip

Sometimes a random permutation is not good enough, because we want to ensure that certain elements are in a certain order. To this end we put the elements which should be in a certain order in a list, and then bundle all those lists in a set. We then give this set of lists to the function Zip: this function makes a random interleaving of the lists in the set. This keeps the elements of the sublists in order, but necessarily together.

11.3.1.3 Split

Split gets a set as input, and returns a tuple of two non-overlapping subsets, which when joined are the original set again. More formally, if $(A, B) = \text{Split}(C)$, then:

1. $A \cup B = C$
2. $A \cap B = \emptyset$

11.3.1.4 Rnd

The function Rnd returns a random, possibly, empty subset of the natural numbers. The only condition is that the intersection of this set with the set which is given as parameter is the empty set: $S \cap \text{Rnd}(S) = \emptyset$.

11.3.2 Terminals and Functions to Create Them

In our grammar the terminals are the start steps S_i, and the Critical Section steps C_i. Since there can be infinitely many processes, we have an infinite alphabet. To be able to cope with this we introduce some functions to create sets of those steps given sets of natural numbers:

$$\begin{aligned} \text{Start}(S) &= \{S_i | i \in S\} \\ \text{Crits}(S) &= \{C_i | i \in S\} \end{aligned}$$

Lastly, because a process can execute a Critical Section in a Mutexblock, and directly follow this up by a Start, we want to be able to create a set of those actions, that is, a set of lists consisting one C_i and one S_i:

$$\text{CS}(B) = Zip([C_i, S_i] \mid i \in B)$$

11.3.3 The Grammar

We can now use all functions to formulate the grammar for the executions of the programs by Udding and Morris:

$$
\begin{aligned}
\text{Execution} &= \mathit{Tail}(\emptyset) \\
\text{Tail}(S) &= \text{Eblock}(E)\ \ \text{Mblock}(M_E, M_L, M_R)\ \ \text{Tail}(M_E \cup M_R) \\
&\quad \text{With} : E = \text{Rnd}(S) \\
&\qquad\quad M_E = \text{Rnd}(S \cup E) \\
&\qquad\quad (M_L, M_R) = \text{Split}(S \cup R) \\
\text{Eblock}(S) &= \text{Permute}(\text{Starts}(S)) \\
\text{Mblock}(S, C, B) &= \text{Zip}(\{\text{Permute}(\text{Starts}(S) \cup \text{Crits}(C))\} \cup \text{CS}(B))
\end{aligned}
$$

We will not prove that this grammar is correct, but we make an argument:

First we show that a process indeeds the next chance it has. If a process enters in an Eblock, it leaves in the following Mblock. If the process enters in the Mblock, it is given as parameter to Tail, and exits the next Mblock. This is indeed correct behaviour.

We now show that each possible action is indeed modelled:

1. A process can enter in an Eblock. If it does so, it cannot do anything else anymore in the Eblock.
2. If a process enters in an Mblock, it cannot reach the critical section.
3. If a process executes the Critical Section in an Mblock there are two options after that:
   (a) The process does nothing.
   (b) The process re-enters.

All of those actions are indeed modelled. Since this list is complete, this means all possible actions are modelled.

Chapter 12

Dijkstra's Conjecture

In this chapter we (informally) prove the weakened form of Dijkstra's Conjecture. For this conjecture, we assume that we can only use semaphores[1] as concurrency technique. This means that we are assuming that we do not have atomic write actions (like atomic counters), because such an action would indeed make a counterexample possible (algorithm 4.2.1 is an example). With that requirement, we can formulate Dijkstra's conjecture:

Conjecture 12.1. *It is not possible to solve the mutual exclusion problem using only a fixed number of weak semaphores, while guaranteeing freedom from individual starvation.*

12.1 Proof

To prove that we cannot solve the mutual exclusion problem using only weak semaphores we only need to consider the first line of the algorithm. If we want to guarantee freedom from individual starvation, then a process needs to be registered in some way after executing the first line, otherwise the algorithm is unable to give it precedence in any way. Therefore we need to look at the possibilities for commands to execute on the first line:

1. A $\mathcal{P}$ operation:
   If the first operation is a $\mathcal{P}$ operation, there is no registration of newly arriving processes. Therefore, there is no way for the algorithm to make a difference between already blocked, and newly arriving processes. Therefore, starvation is impossible to prevent.

2. A non-semaphore operation:
   This is either a read-action, in which case the previous argument is valid, or a write action. Since we do not know the number of processes we do not have the option to assign each process its own variable, and therefore at least two processes will need to write to the same variable. This means that a reliable write to that variable is not guaranteed, since we have no atomic write actions at our disposal.

3. A $\mathcal{V}$ operation:
   The $\mathcal{V}$ operation does provide an option to register a process. In fact, it can be regarded as a counter. Therefore it seems a valid candidate.

[1] We do not however impose the restriction on the semaphores that they have to be binary semaphores.

12.2 The $\mathcal{V}$ Operation is no Valid Candidate

To be able to prove that the $\mathcal{V}$ operation is not a valid candidate, we first consider some executions including the $\mathcal{V}$ operation.

First we consider the following execution (using subscripts to denote the executing process):

$$\mathcal{P}_i(x)\mathcal{V}_j(x)$$

If this is a valid execution, then so is the following one:

$$\mathcal{V}_j(x)\mathcal{P}_i(x)$$

And both executions have the same end-state. Because if a process can execute a $\mathcal{P}$ before a $\mathcal{V}$, it can certainly do so after a $\mathcal{V}$, and the value of x will be the same in the end-state for both executions (since in both cases it is once incremented, and once decremented).[2]

Next, we consider the following execution:

$$\mathcal{V}_i(x)\mathcal{V}_j(x)$$

Since the $\mathcal{V}$ operation does in no way depend on which process executes it, we can interchange the operations, without changing anything to the execution.

Since non-semaphore operations do not change in any way because of the $\mathcal{V}$ operation, this means that we can move the $\mathcal{V}$ operation arbitrary far in backwards time. We are now able to prove that we cannot guarantee freedom from individual starvation using the $\mathcal{V}$ operation as first operation:

Consider an algorithm which starts with $\mathcal{V}(x)$. Since this algorithm is supposedly free from individual starvation, it has an upper time limit $t_{\max}$ that a process can stay in the algorithm. We now consider the following execution xs of the algorithm: a process p is idle between time t_0 and time $t_{\max}+1$, and then does the $\mathcal{V}(x)$ step which starts the algorithm at time $t_{\max}+1$. Then the sequence obtained from xs by moving the $\mathcal{V}(x)$ step to time t_0 is also a valid execution. This implies that we can construct an execution, in which process p stays for at least $t_{\max}+1$ steps in the algorithm, but this is a contradiction with our assumption that $t_{\max}$ is the maximum number of steps a process can stay in the algorithm.

This means that indeed (the weakened version of) Dijkstra's Conjecture is true, and it is not possible to solve the mutual exclusion problem using only weak semaphores.

[2] Note however, that if we would try this with a buffered semaphore, this would not be true: the execution of (the first part of) the $\mathcal{P}$ operation could result in entering the 'guarded region', instead of going into the blocked set, when an extra $\mathcal{V}$ is executed before the $\mathcal{P}$ is executed, therefore ending up in a different state.

Chapter 13

Future Work

Even though the basis of this research is rooted in relatively ancient papers, there are some interesting results, which might be further refined. In this chapter we indicate what further research we think to be interesting.

13.1 Dijkstra's Conjecture

The results we have now are that the original conjecture of Dijkstra was false, and that our weakened conjecture is true. But there is still quite a gap between a weak semaphore and a buffered semaphore, and it might be interesting to look further at what the weakest semaphore is with which we can implement mutual exclusion (using only a finite number of semaphores), and ensure freedom from individual starvation.

The polite semaphore comes to mind as a possible candidate. The proof of Dijkstra's Conjecture does not work when this semaphore is used. On the other hand, this semaphore does not work when used in the algorithms by Morris and Udding instead of the buffered semaphore (see appendix C).

This means that there might be an alternative algorithm where this semaphore does suffice. We believe that the polite semaphore, when used in the algorithm as proposed by Martin and Burch in [6], would be enough to ensure mutual exclusion without individual starvation. We have verified that the kind of counterexample as given in appendix C, would not work for this algorithm[1]. We are convinced that this means that this algorithm is indeed free from starvation with the Polite Semaphore. We have however decided not to write down a chapter on this for the following reasons:

1. The Polite Semaphore is definitely weaker than the Buffered Semaphore, but its definition is not very elegant (it uses the implicit $proc_{at}$). It would be interesting to look how one could formulate this type of semaphore as weak as possible (however, a set of blocked processes does seem necessary to use it in the algorithm by Martin and Burch, otherwise a counter-example can be constructed).

2. Our proof that the algorithm is starvation free is extremely hand-wavy, as the argument is based on our knowledge from the other algorithms, that the starvation could only occur on the first $\mathcal{P}$ operation. Furthermore, we have only constructed the (what we believe to be) only possible type of counter-example, to conclude that it cannot occur, but this is hardly a watertight proof.

[1] Because the sequence $\mathcal{V}(y)\mathcal{P}(y)\mathcal{V}(z)$ can only be executed if there are no processes blocked at the first $\mathcal{P}(y)$. If there are, instead the fp will block on its $\mathcal{P}(y)$, and not leave the first block.

So it would be interesting to see if someone could formally prove the algorithm by Martin and Burch, using the Polite Semaphore. Even more interesting would be to see if it would be possible to modify the algorithm to support an even weaker semaphore.

13.2 Comparison of Algorithms

In this document we provided a tool to compare the algorithms by Udding and Morris to other algorithms implementing a solution to the mutual exclusion problem: the set of possible executions. To compare the algorithms to the algorithm by Aravind and Hesselink we also need the set of possible executions of that algorithm. And if this set is indeed the same, it might be interesting to see if one can find one algorithm in the other by introducing ghost variables.

13.3 Automating the Proof of Invariants

We described the process of proving an invariant in section 6.3. We think most of those steps could, and should, be automated. We were shocked to see how large a percentage of time was wasted on trivial, automatable tasks.

Conceptually, we want to be able to do the following: formulate a list of invariants, press a button, get presented with a list of theorems of non-trivial steps which the human prover needs to prove, press another button, and get presented with the entire proof up to and including the proof of `iqall`.

Since PVS is not a dedicated invariant-prover, we do not expect such an interface soon. However, we should be able to implement it to quite a large extent.

For the first button, we propose the following pipeline. A script generates all needed list theorems based upon the list of invariants. Then it installs (for instance using ProofLite) a general proof for a list theorem on all those theorems. This proof will be something like a `listproof` followed by a `split` and a `grind`, followed by a piece of Lisp code, which writes to a file which branches (and therefore steps) failed.

As the next step, the script wil read said file, rewrite the list-proofs to exclude the steps which could not be proved, and creates a list of theorems for the non-trivial steps.

Next, the human prover can prove the non-trivial steps, after which the second script is invoked. This script then writes the `kept-valid` theorems by scraping the additional invariants from the step-theorems. It then proves the theorems as well (for instance by creating a proof and installing it using ProofLite). Lastly, it formulates and proves `iqall`.

Chapter 14

Conclusion

In this document we have presented the Mutual Exclusion problem, and shown why weak or buffered semaphores are no solution. Next we presented a general algorithm which does solve the problem, but is incomplete. From this algorithm we derived the algorithm by Udding and the algorithm by Morris. By presenting the algorithms in this way we have shown the similarity between the two.

We proved both algorithms with a theorem prover. Even though the algorithms were already proved, they were only proved by hand. And since algorithms involving concurrency are so complex, it is easy to make mistakes in their proofs. However, by using a theorem prover we have definitively proved the correctness of the algorithms.

We presented a way to compare the algorithms to other algorithms implementing Mutual Exclusion, by representing algorithms by their executions.

Lastly we took a conjecture of Dijkstra, which was disproved by the existence of the algorithms by Morris and Udding, and changed it to a slightly weaker theorem which is true, and which we have proved.

Concluding, we have conclusively proved it possible to implement mutual exclusion without individual starvation, using only a fixed number of buffered semaphores. We have also proved it impossible to implement using only weak semaphores. This means that we have made more precise, what guarantees a semaphore should offer, to be able to use it to implement mutual exclusion.

Bibliography

[1] Gregory R. Andrews. *Concurrent Programming: Principles and Practice.* Addison-Wesley Publishing Company, 1991.

[2] J.K. Annot, M.D. Janssens, and A.J. van de Goor. Comment on Morris's starvation-free solution to the mutual exclusion problem. *Information Processing Letters*, 23:91–97, 1986.

[3] Edsger W. Dijkstra. A strong P/V-implementation of conditional critical regions. EWD 651, circulated privately, 1977.

[4] Wim H. Hesselink and Alex A. Aravind. Queue based mutual exclusion with lineairly bounded overtaking. *Science of Computer Programming*, to appear.

[5] A.J. Martin and J.L.A. van de Snepscheut. Design of synchronization algorithms. In M. Broy, editor, *Constructive Methods in Computing Science*, pages 445–478. Springer, 1989.

[6] Alain J. Martin and Jerry R. Burch. Fair mutual exclusion with unfair p and v operations. *Inf. Process. Lett.*, 21(2):97–100, 1985.

[7] J.M. Morris. A starvation-free solution to the mutual exclusion problem. *Information Processing Letters*, 8:76–80, 1979.

[8] J.T. Udding. Absence of individual starvation using weak semaphores. *Information Processing Letters*, 23:159–162, 1986.

Appendix A

List of Invariants Used for the Proof of Udding's Algorithm

Utility Invariants

(U:I)
$$\begin{aligned} & pc(p) = 10 \lor pc(p) = 11 \lor pc(p) = 12 \lor pc(p) = 13 \lor pc(p) = 14 \\ & \lor pc(p) = 20 \lor pc(p) = 21 \lor pc(p) = 22 \lor pc(p) = 23 \lor pc(p) = 24 \\ & \lor pc(p) = 25 \lor pc(p) = 26 \lor pc(p) = 27 \lor pc(p) = 28 \lor pc(p) = 29 \\ & \lor pc(p) = 30 \lor pc(p) = 40 \lor pc(p) = 41 \lor pc(p) = 42 \lor pc(p) = 43 \\ & \lor pc(p) = 44 \lor pc(p) = 45 \end{aligned}$$

Defined here and called `inrange` in PVS.

Semaphore Invariants

(U:S1)
$$queue + \#proc_{queue} = 1$$

Defined on page 36 and called `sem_queue` in PVS.

(U:B1)
$$p \in blocked_{enter} \Longrightarrow pc(p) = 11 \lor pc(p) = 22$$

Defined on page 36 and called `blocked_enter_implies_11_22` in PVS.

(U:B2)
$$pc(p) = 11 \lor pc(p) = 22 \Longrightarrow enter = 0$$

Defined on page 38 and called `line11_22_implies_enter_zero` in PVS.

(U:S2)
$$enter + mutex + \#proc_{enter} + \#proc_{mutex} = 1$$

Defined on page 36 and called `sem_em` in PVS.

(U:M1)
$$p \in proc_{queue} \land q \in proc_{queue} \Longrightarrow p = q$$

Defined on page 36 and called `mutex_queue` in PVS.

(U:M2) $$\begin{array}{c}(p \in proc_{enter} \vee p \in proc_{mutex}) \\ \wedge (q \in proc_{enter} \vee q \in proc_{mutex}) \\ \Longrightarrow p = q\end{array}$$

Defined on page 36 and called `mutex_em` in PVS.

Invariants Regarding nm

(U:NM1) $$nm = \#between_{nm}$$

Defined on page 38 and called `nm_is_nrp` in PVS.

(U:O2) $$nm \geq 0$$

Defined here and called `nm_greq_zero` in PVS.

(U:NM3) $$(pc(p) = 24 \vee pc(p) = 42) \Longrightarrow tmp(p) = nm$$

Defined here and called `nm_is_tmp` in PVS.

Invariants Regarding ne

(U:NE2) $$ne = \#between_{ne}$$

Defined on page 39 and called `ne_is_nrp` in PVS.

(U:NE3) $$ne \geq 0$$

Defined here and called `ne_greq_zero` in PVS.

Invariants Regarding Temporary Values of Variables

(U:TMP1) $$pc(p) = 29 \Longrightarrow ne = 0$$

Defined here and called `at29_imp_ne_is_zero` in PVS.

(U:TMP2) $$pc(p) = 28 \Longrightarrow ne > 0$$

Defined here and called `at28_imp_ne_gr_zero` in PVS.

(U:TMP3) $$pc(p) = 45 \Longrightarrow nm = 0$$

Defined here and called `at45_imp_nm_is_zero` in PVS.

(U:TMP4) $$pc(p) = 44 \Rightarrow nm > 0$$

Defined here and called `at44_imp_nm_gr_zero` in PVS.

(U:TMP5) $$pc(p) = 13 \vee pc(p) = 26 \Rightarrow tmp(p) = ne$$

Defined here and called `ne_is_tmp` in PVS.

(U:TMP6) $$p \in proc_{mutex} \Longrightarrow ne = 0$$

Defined on page 48 and called `proc_mutex_implies_ne_zero` in PVS.

Invariants Regarding Which Section is Open

(U:O1) $$ne = 0 \wedge nm > 0 \Longrightarrow enter = 0$$

Defined on page 39 and called `ne0_nagr0_imp_enter0` in PVS.

(U:NM2) $$nm = 0 \Longrightarrow mutex = 0$$

Defined on page 38 and called `nm0_imp_mutex0` in PVS.

(U:P1) $$\text{pred-enter} \Longrightarrow mutex = 0$$

Defined on page 48 and called `pred_enter_implies_mutex0` in PVS.

(U:P2) $$\text{pred-enter} \Longrightarrow (\forall p : p \in proc_{mutex} \Rightarrow pc(p) \geq 43)$$

Defined on page 48 and called `pred_enter_implies_mutex_empty` in PVS.

(U:P3) $$\text{pred-mutex} \Longrightarrow enter = 0$$

Defined on page 48 and called `pred_mutex_implies_enter0` in PVS.

(U:P4) $$\text{pred-mutex} \Longrightarrow (\forall p : p \in proc_{enter} \Rightarrow pc(p) = 27 \vee pc(p) = 29)$$

Defined on page 48 and called `pred_mutex_implies_enter_empty` in PVS.

Appendix B

List of Invariants Used for the Proof of Morris' Algorithm

Utility Invariants

(M:I) $$\begin{aligned} & pc(p) = 10 \vee pc(p) = 11 \vee pc(p) = 12 \vee pc(p) = 13 \vee pc(p) = 14 \\ \vee\ & pc(p) = 20 \vee pc(p) = 21 \vee pc(p) = 22 \vee pc(p) = 23 \vee pc(p) = 24 \\ \vee\ & pc(p) = 25 \vee pc(p) = 26 \vee pc(p) = 27 \vee pc(p) = 28 \vee pc(p) = 31 \\ \vee\ & pc(p) = 29 \vee pc(p) = 30 \vee pc(p) = 40 \vee pc(p) = 41 \vee pc(p) = 42 \\ \vee\ & pc(p) = 43 \vee pc(p) = 44 \vee pc(p) = 45 \end{aligned}$$

Defined on page here and called `inrange` in PVS.

Semaphore Invariants

(M:S1) $$b + \#proc_b = 1$$

Defined on page on page 36 and called `sem_b` in PVS.

(M:B1) $$p \in blocked_b \Longrightarrow pc(p) = 11 \vee pc(p) = 24$$

Defined on page on page 36 and called `blocked_b_implies_11_24` in PVS.

(M:B2) $$pc(p) = 11 \vee pc(p) = 24 \Longrightarrow b = 0$$

Defined on page on page 40 and called `line11_24_implies_b_zero` in PVS.

(M:M1) $$p \in proc_b \wedge q \in proc_b \Longrightarrow p = q$$

Defined on page on page 36 and called `mutex_b` in PVS.

(M:S2) $$a + m + \#proc_a + \#proc_m = 1$$

Defined on page on page 36 and called `sem_am` in PVS.

(M:M2) $$\begin{aligned} & (p \in proc_a \vee p \in proc_m) \\ & \wedge (q \in proc_a \vee q \in proc_m) \\ & \Longrightarrow p = q \end{aligned}$$

Defined on page on page 36 and called `mutex_am` in PVS.

Obsolete Invariants

Because the proof of the algorithm by Morris was the first proof we did, our invariants were chosen a bit suboptimal. After using a set of invariants in the proof of the algorithm by Udding which were much more elegant, we decided to use those in this algorithm as well. However, entirely removing all old invariants is a lot of work, and since the proof does work, we decided to settle for removing most old invariants.

This means some remnants of the old proof remain. First there are three invariants: `semb1` and the old two mutual exclusion invariants. Those have been used in quite some proofs, so we decided to leave those in. We removed all other invariants, but since a lot of proofs depend on certain formulas being on certain sequent numbers, we decided to replace them by placeholders in the proof. Therefore we have three boolean functions `placeholder(x)`, `placeholder2(x)`, and `placeholder3(x)`. They are all defined as `TRUE` and therefore do not contribute anything to the proof, other than ensuring that each formula stays on the same sequent.

(M:SB1) $$inbrange(p) \vee pc(p) = 11 \vee pc(p) = 24 \Longrightarrow b = 0$$

Defined on page here and called `semb1` in PVS.

(M:MB) $$\begin{aligned}&(((x\text{`}pc(p) = 11 \vee x\text{`}pc(p) = 24) \wedge p \notin blocked_b) \vee inbrange(p)))\\ &\wedge(((pc(q) = 11 \vee pc(q) = 24) \wedge q \notin blocked_b) \vee inbrange(q))\\ &\Longrightarrow p = q\end{aligned}$$

Defined on page here and called `mutexb` in PVS.

(M:MAM)
$$(inarange(p) \vee inmrange(x)) \wedge (inarange(q) \vee inmrange(q)) \Longrightarrow p = q$$

Defined on page here and called `mutexam` in PVS.

Invariants Regarding The Value of na

(M:NA1) $$(pc(p) = 13 \vee pc(p) = 26) \Longrightarrow tmp(p) = na$$

Defined on page here and called `na_is_tmp` in PVS.

(M:NA2) $$na = \#between_{na}(x)$$

Defined on page here and called `na_is_nrp` in PVS.

(M:NA3) $$na \geq 0$$

Defined on page here and called `na_gr0` in PVS.

Invariants Regarding The Value of nm

(M:NM1) $$(pc(p) = 22 \vee pc(p) = 42) \Longrightarrow tmp(p) = nm$$

Defined on page here and called `nm_is_tmp` in PVS.

(M:NM2) $$nm = \#between_{nm}(x)$$

Defined on page here and called `nm_is_nrp` in PVS.

(M:NM3) $$nm \geq 0$$

Defined on page here and called `nm_gr0` in PVS.

Invariants Regarding The Value of nmq

(M:NMQ1) $$(nm = nmq) \vee (\exists r : 22 \leq pc(r) \leq 26)$$

Defined on page here and called `nm_is_nmq_strong` in PVS.

(M:NMQ2) $$inmrange(p, x) \Longrightarrow nm = nmq$$

Defined on page here and called `nm_is_nmq_lm` in PVS.

(M:NMQ3) $$nm \geq nmq$$

Defined on page here and called `nm_gr_nmq` in PVS.

(M:NMQ4) $$nmq \geq 0$$

Defined on page here and called `nmq_greq_zero` in PVS.

(M:NMQ5) $$nmq = \#between_{nmq}$$

Defined on page here and called `nmq_is_nrp` in PVS.

Invariants Regarding The Value of naq

(M:NAQ1) $$pc(p) = 27 \Longrightarrow na = naq$$

Defined on page here and called `na_is_naq_at_27` in PVS.

(M:NAQ2) $$naq \leq na$$

Defined on page here and called `naq_smaller_na` in PVS.

(M:NAQ3) $$naq \leq 0$$

Defined on page here and called `naq_greq_zero` in PVS.

(M:NAQ4) $$inmrange(p) \Longrightarrow naq = 0$$

Defined on page here and called `naq_is_zero_in_lm` in PVS.

(M:NAQ5) $$m > 0 \Longrightarrow naq = 0$$

Defined on page here and called `m_nonzero_imp_naq_zero` in PVS.

Invariants Regarding Temporary Values of Variables

(M:TMP1) $$(pc(p) = 29 \vee pc(p) = 30) \Longrightarrow na > 0$$

Defined on page here and called `na_gr0_at_29_30` in PVS.

(M:TMP2) $$(pc(p) = 44) \Longrightarrow nm = 0$$

Defined on page here and called `nm_zero_at44` in PVS.

(M:TMP3) $$(x`pc(p) = 45) \Longrightarrow nm > 0$$

Defined on page here and called `nm_gr0_at_45` in PVS.

(M:TMP4) $$(pc(p) = 29 \vee pc(p) = 30) \Longrightarrow naq > 0$$

Defined on page here and called `naq_is_gr_zero_at_29_30` in PVS.

(M:TMP5) $$(pc(p) = 28 \vee pc(p) = 31) \Longrightarrow naq = 0$$

Defined on page here and called `naq_is_zero_at_28_31` in PVS.

Invariants Regarding Which Section is Open

(M:O1) $$na = 0 \wedge nm > 0 \Longrightarrow a = 0$$

Defined on page on page 40 and called `na0_imp_a0` in PVS.

(M:O2) $$nm = 0 \Longrightarrow m = 0$$

Defined on page on page 40 and called `nm0_imp_m0` in PVS.

(M:P1) $$nmq > 0 \wedge naq = 0 \Longrightarrow a = 0$$

Defined on page here and called `predm_implies_lm_next` in PVS.

(M:P2) $$nmq = 0 \vee naq > 0 \Longrightarrow m = 0$$

Defined on page here and called `preda_implies_la_next` in PVS.

(M:P3) $$nmq > 0 \wedge naq = 0 \Longrightarrow \neg(20 < pc(r) < 27)$$

Defined on page here and called `predm_implies_la_empty` in PVS.

(M:P4) $$nmq = 0 \vee naq > 0 \Longrightarrow \neg(40 < pc(r) \leq 42)$$

Defined on page here and called `preda_implies_lm_empty` in PVS.

Appendix C

Polite Semaphore in the Algorithms by Udding and Morris

We choose to use the buffered semaphore for the algorithms by Udding and Morris. The motivation to choose that semaphore, and not the polite semaphore is given in this chapter. First we present the algorithm by Udding, but with the polite semaphore instead of the buffered semaphore. Next we give an execution of the program which proves it is not starvation free.

C.1 Motivation for Trying the Polite Semaphore

When there are only two processes involved, the buffered semaphore implements mutual exclusion without individual starvation. Morris presents his algorithm as a solution to the problem of the mutual exclusion problem with more than two processes. From this, we got the impression that the semaphore he used in his algorithm had to meet only one requirement: implement (fair) mutual exclusion for two processes. In chapter 3 we therefore proposed another semaphore which does this: the polite semaphore.

However, in chapter 4 we formulated a stronger demand: the semaphore needs to be a buffered semaphore. We need this to make the statement of the second case mentioned true: if the process executing $\mathcal{P}_2(X)$ can immediately continue (without blocking), this means that there are no processes blocked on $\mathcal{P}(X)$. This statement is not true for the polite semaphore. To show that this semaphore therefore cannot be used in the algorithm we devised an execution which exploits that aforementioned statement is not true.

C.2 Algorithm by Udding

The only difference between this algorithm and the algorithm as presented in chapter 5 is the $\mathcal{P}$ and $\mathcal{V}$ operations on *enter*. Again we don't write out the $\mathcal{V}$ operation, but for *enter* we will use the following $\mathcal{V}$ operation:

Algorithm C.2.1: $\mathcal{V}$ FOR THE POLITE SEMAPHORE

```
procedure V(enter)
 20     if proc_at(11) ≠ ∅
            then forbidden := self;
        enter := enter + 1;
```

The algorithm becomes:

Algorithm C.2.2: UDDING'S ALGORITHM

```
10      if enter > 0 ∧ self ≠ forbidden
          then
            enter := enter − 1; forbidden := −1;
            goto 12;
          else goto 11;                                  } REGISTRATION
11      await enter > 0 ∧ self ≠ forbidden;
        enter := enter − 1; forbidden := −1;
12      tmp := ne;
13      ne := tmp + 1;
14      V(enter);

20      P(queue);
21      if enter > 0 ∧ self ≠ forbidden
          then
            enter := enter − 1; forbidden := −1;
            goto 23;
          else goto 11;
22      await enter > 0 ∧ self ≠ forbidden;
        enter := enter − 1; forbidden := −1;
23      tmp := nm;                                     } ENTER
24      nm := tmp + 1;
25      tmp := ne;
26      ne := tmp − 1;
27      if ne > 0
28        then V(enter);
29        else V(mutex);
30      V(queue);

40      P(mutex);
41      tmp := nm;
42      nm := tmp − 1;
43      Critical Section;                              } MUTEX
        if nm > 0
44        then V(mutex);
45        else V(enter);
```

C.3 Can the Polite Semaphore be Used

Using the algorithm of Udding, with the polite semaphore, we hereby present a counterexample of the correctness of the algorithm of Udding, using 3 processes.[1] Processes 1 and 2 will infinitely cycle while process 3 will stay at line 11, blocked on *enter* (*enter*, *mutex* and *queue* are abbreviated as e, m and q):

action	pc_1	pc_2	pc_3	*forbidden*	e	m	q	ne	nm
	10	10	10	-1	1	0	1	0	0
$p_1 \to 21$	21	10	10	1	1	0	0	1	0
$p_2 \to 14$	21	14	10	-1	0	0	0	2	0
$p_3 \to 11$	21	14	11	-1	0	0	0	2	0
$p_2 \to 20$	21	20	11	2	1	0	0	2	0
$p_1 \to 40$	40	20	11	1	1	0	1	1	1
$p_2 \to 40$	40	40	11	2	0	1	1	0	2
$p_1, p_2 \to 10$	10	10	11	2	1	0	1	0	0
$p_1 \to 21$	21	10	11	1	1	0	0	1	0
$p_2 \to 14$	21	14	11	-1	0	0	0	2	0

Since the grey rows are exactly the same state, we can repeat the sequence in between indefinitely, and let p_3 starve. This means that the polite semaphore is not fit to be used instead of the buffered semaphore, in the algorithms by Udding and Morris.

Note that this counterexample would not work with a buffered semaphore, as the step $p_2 \to 40$ would not be possible: at that point p_3 would already been selected to enter the (first) guarded region of *enter* when p_1 executed its last $\mathcal{V}(enter)$.

[1] We do not prove this for the algorithm by Morris, but actually, the same example works (with adjustment of the line numbers, of course)