Verifying a barrier algorithm with a mechanical theorem prover

Anton Smit

Supervisor: W.H. Hesselink

RuG

Computer Science

Masters thesis

Verifying a barrier algorithm with a mechanical theorem prover

Anton Smit

Supervisor: W.H. Hesselink

Rijksuniversiteit Groningen
Computer Science
Postbus 800
9700 AV Groningen

September 2001

Contents

Chapter 1

Introduction

A way to reduce the execution time of a program, is to distribute the tasks of the program over several processors. It is not easy to design these programs because there are many factors you have to take into account, like deadlock (tasks are waiting for each other) or the possibility that tasks come in a infinite loop.

To prove such a program's correctness is very complex. It is not recommendable to prove such a program by hand. A lot of administration is needed for all the different cases the program can go into during its execution. Also is it very time consuming and the chance of overlooking something or making a mistake is very high. An approach to prove the correctness of such a complex program is the use of a mechanical theorem prover. A mechanical theorem prover is a piece a software that does the proving for the designer. Learning how such a prover works can be very hard.

The assignment was to verify a barrier algorithm with such a theorem prover and will be discussed in this paper. The goal of this assignment is to learn more about mechanical theorem provers. But also how it is to be working independently on such a theoretical problem like this.

In chapter 2 we will lay out the theory that is needed to understand a concurrent program like the barrier. The barrier itself is also discussed in this chapter including a global proof. In chapter 3 we will discuss some aspects of theorem provers. Aspects like what they are and what they do, and also the advantages and disadvantages of theorem provers. We will conclude that chapter with some information about how to use the theorem prover NQTHM cf. [4] and why we have chosen this particular theorem prover for this assignment. In chapter 4 and 5 we give our implementation of the barrier in NQTHM code. In chapter 4 the implementation of the prelude we use for the barrier implementation is discussed and the barrier itself is layed out in chapter 5. Chapter 6 contains all the problems and obstacles we encountered during the proof. In chapter 7 we will discuss extensions that can be made to the barrier algorithm. We conclude this paper with concluding remarks.

Chapter 2

The Barrier

Iterative algorithms are used for solving many problems cf. [1]. Some of those programs needs to repeat the same calculations over and over again on several parts of a datastructure like for example all elements of a list. Instead of computing those executions successively, many programs can gain great performance by doing those executions parallel of each other. When such tasks are done in parallel almost always some kind of synchronization is needed. Otherwise some processes will run ahead and other processes will stay behind. This will mostly result in wrong answers. That is why we need some sort of barrier. The purpose of the barrier is to block all the processes until the last one has arrived. When it does it will let all the processes pass. In this way no processes will stay behind or will run too far ahead. Let us work out a simple example.

2.1 An example

Let us say we have the global array a with n elements. All elements are initially 0. We want to increment all these elements by one. We assign n processes ($q(1)$ to $q(n)$), one each element of the array. And a process p that prints the elements of the array to the screen or to a file.

The processes $q(i)$ will probably look like this:

```
q(i):
   loop
      a[i]++;
   end
```

When p wants to print the array after a few runs it will probably never have the same elements. It is unlikely that when $q(i)$ has executed its body x times and thus incremented `a[i]` with x, all other processes $q(j)$ also have incremented their array elements `a[j]` with x. If we want to print only arrays with the the same elements we have to insert a barrier in the processes:

```
q-with-barrier(i):
   loop
      a[i]++;
      barrier
   end
```

If the barrier is implemented well, none of the processes will continue before the last process has reached the barrier. When the final process has arrived, it can give p a signal and the array will be printed with identically elements.

2.2 Some theory and commands

Like all concurrent algorithms there is always the danger of processes interfering with each other. That is why we need to use terms like *critical sections*, *mutual exclusion*, *atomic actions* and *waiting queues*. Below we will give an explanation of these terms.

Critical Section: This is the part of the program that is sensitive for interference of other processes. If another process interferes it will result in deadlock (a process is waiting for a condition that will never happen) or will give an incorrect answer.

Mutual Exclusion: This is a solution to solve the critical section (from now on abbreviated as CS) problem. It means that only one process may execute, and the other processes have to wait until the process has done its executions in the CS. This can be done be defining a global data structure called *mutex* to represent which process is in the CS. We define the mutex as a record with the single field owner. Let m be a mutex. Then $m.owner = \perp$ means that no process has the mutex and every process can enter the critical section if it wants to. When $m.owner = p$ no other process but p can execute, all other processes have to wait until the mutex is released again (i.e. becomes $\perp$). To enter and leave a CS we use the following commands respectively cf. [2]:

lock(m): $\langle$**await** $m.owner = \perp$ **then** $m.owner := self\ \rangle$
unlock(m): $\langle$**await** $m.owner = self$ **then** $m.owner := \perp\rangle$

The $\langle\ \rangle$ brackets are called *atomicity brackets*. The commands within those brackets are executed without any interference by other processes. The **await** command just checks to see if its guard holds. If it does it executes the command next to **then** or else it just waits until its guard holds.

Waiting Queues: Processes that have reached the barrier before the last one have to be disabled until the last one arrive. This is performed by a waiting queue. The queue is a set of processes which are not allowed to perform executions. They are waiting for a signal given by another process to be activated again. Below is the command for entering the queue. This can also be found in [2]

wait(v, m):
  $\langle$**unlock**(m); *insert self in* $Q(v)\rangle$;
  lock(m)

This function releases the mutex and inserts the process executing **wait** into the waiting queue $Q(v)$. This happens in the same atomic action. If **wait** did not have an atomic action, the last process entering the barrier will never know that it is the last one. This is due to the fact that it is not certain that each previous process has entered the queue, which will lead to deadlock.

Note that the mutex is released in **wait** while entering the queue. Therefore other processes can continue their actions and the process that has executed **wait** needs to re-acquire the mutex when it is released again by another process.

To empty the queue we have the following command:

broadcast(v): $\langle$ *release all the processes from* $Q(v)\rangle$

When the waiting queue is empty, all the processes are active again. There is also a command for releasing some of the processes, namely **signal**(v). If the queue was empty already the signal commands are both equal to *skip*. All these commands like are already provided in the POSIX standard and languages like Modula-3 and Java.

2.3 The algorithm itself

Now that we have discussed all the concurrency theory that we need we can move on to the algorithm itself. Like we said at the beginning of this chapter the barrier algorithm is located

within a process. There are N processes and all the processes will execute the barrier. The shared variable `atbar` is initially 0 and the private ghost variable *cnt* is initially C for all N processes. The algorithm is almost the same as in [2]:

```
loop
    TNS;
    lock(m);
    cnt ++;
    if atbar = N − 1 then
        atbar := 0;
        broadcast(v);
    else
        atbar ++;
        wait(v, m);
    fi;
v:  unlock(m)
end
```

The algorithm begins with the *Terminating Noncritical Section* `TNS`. Almost everything can be implemented here as long it terminates. Also it does not modify the variables and data structures in the `barrier` code like the `a[`i`]++` command in our example.The algorithm does nothing more than checking if the process executing the algorithm is the last one: `atbar` $= N - 1$ where N is the number of processes. If it is, it releases all the processes in the waiting queue. If the process is not the last one, it puts itself in the waiting queue, releases the mutex m and waits for the queue to be emptied. This program line is labeled by the condition variable v and is called the *re-entry* location.

Note that the barrier is locked and unlocked by the mutex m. Without the mutual exclusion it will definitely go wrong. For example let us say that a process p has incremented `atbar` but the process is not yet inserted in the waiting queue $Q(v)$. When the last process arrives it will empty $Q(v)$. This will result in deadlock because when p puts itself in $Q(v)$ after the **broadcast** it will stay there forever because `atbar` will then never be incremented in p again. Then `atbar` will never be $N - 1$ with deadlock as result.

We changed the algorithm slightly during the proof. One of the changes we made is the relocation of the `atbar` incrementation. In the previous versions it was located next to the *cnt* incrementation. But we moved it to make proving one of the invariants later on simpler. More explanation about this will be given during the proof.

2.4 Global proof of the algorithm

This global proof is a just a summary of the proof in [2] but we need it here to make to mechanical proof more understandable.

As can be seen in the code the variable *cnt* is not in typewriter font. That is because it is a private ghost variable. It can be omitted in the code, but we need it to prove the algorithm. The purpose of the barrier can be seen in the following Hoare Triple where C is a constant and *self* is the process executing the commands :

(H) $\{\ cnt.self = C\ \}$ `barrier` $\{\ cnt.self = C + 1\ \}$

It is easy seen that it follows directly from the code. We want that the barrier guarantees that no process runs ahead of the others. This is expressed (BC). If process q runs ahead then $cnt.q > cnt.r$ for some process r, but (BC) then implies q **notin** `TNS`, so q is waiting.

(BC) q **in** `TNS` $\Rightarrow$ $cnt.q \leq cnt.r$

A process is only in the queue when it has executed **wait** and it can only be executed again when it is released from the queue. This is formulated in the following invariant:

(WL) $\quad q \in Q(v) \quad \Rightarrow \quad q$ **at** v

If we combine (WL) with (M0) defined below it will imply (BC).

(M0) $\quad q \notin Q(v) \quad \Rightarrow \quad cnt.q \leq cnt.r$

Since (BC) follows from (WL) and (M0) and since (WL) is an invariant, our main obligation is (M0). This will be proved as follows. (M0) will be threatened when q increases the counter but does not enter $Q(v)$. This only happens when q is the last one and executes **broadcast**. We need the following invariants to prove (M0).

(M1) $\quad \texttt{atbar} = |Q(v)|$
(M2) $\quad q \notin Q(v) \quad \wedge \quad r \in Q(v) \quad \Rightarrow \quad cnt.r = cnt.q + 1$

Now we know that when $\texttt{atbar} = N - 1$, the process executing the barrier algorithm is the last process that is not in the waiting queue according to (M1). We also know that after incrementing *cnt* all process counters will be equal because of (M2). To prove these invariants we use the mechanical theorem prover. Note that this (M2) is not equal to the (M2) presented in [2]. The reason why we changed it will be discussed in chapter 5.4.

Chapter 3

Mechanical theorem provers

Like we said in the introduction mechanical theorem provers can be used to verify concurrent algorithms like our barrier. In this chapter we will discuss various aspects of theorem provers. First we will explain what a theorem prover is and how it works. Then we will discuss the advantages and disadvantages of the use of a theorem prover. And finally we will tell more about the theorem prover we used. In that section we will also lay out the commands that are needed to understand the code we use throughout the paper.

3.1 What are theorem provers?

A theorem prover is a piece of software that needs certain definitions and lemmas as input. If these are correctly written so that the prover can understand them , the prover will then apply its own logic and base functions on these definitions and lemmas and try to prove them. To verify lemmas mechanical provers often use a lot of number theory and Boolean logic. If that does not prove a lemma, it tries to proof it by induction or generalization. How a theorem prover works exactly will not be discussed in this paper. It is too complex and each theorem prover uses a different approach to verify programs. Some global knowledge how a theorem prover works should be enough to understand this paper.

When the prover fails to prove a lemma it is your task as the designer of the definitions and lemmas to find out why the prover cannot verify it. There are two possibilities for the designer to consider.

- The lemma or the definitions the lemma uses just are not correct and the designer has to correct the mistake.
- The theorem prover is not able to prove the lemma and needs more information. This information can be given by designing additional lemmas to be done by the designer.

When the prover fails to verify a lemma the designer has to look through the generated code to see which of the two cases described above can be applied here. The process continues until the prover has verified all the lemmas the designer wanted to be verified.

3.2 Advantages and disadvantages

The main advantage of theorem provers is when the prover succeeds to verify the definitions and lemmas, the designer can be very certain that they are valid. If you want to verify a concurrent algorithm like our Barrier, it is almost impossible to do it by hand. It has a lot of different cases due to the multiple processes involved and this requires a lot of administration. A theorem prover can do all the administration for you.

Another great advantage of theorem provers is that the problem that you want to verify can change while you are verifying it. For example when you are verifying an problem, it is quite possible to make a mistake in designing the problem itself. If you are halfway verifying the proof when you discover it, large parts of the proof have to be done all over again. If you are verifying the problem by hand this can be an enormous task. With a theorem prover changing a couple of definitions or lemmas can be enough to correct the mistake.

It also has a great disadvantage. It is very difficult to learn theorem provers. A lot of experience is needed to recognize why a certain proof fails. If the reason for failure is found it has to be corrected. When you are experienced in using theorem provers, correcting the input is done in far less time than done by an inexperienced user.

Another disadvantage is that theorem provers differ a lot from each other. When you know one of them very well, it does not mean that learning another one is easy. Also theorem provers are not very common. Finding good information about them can be hard.

3.3 NQTHM

The theorem prover we use is NQTHM cf. [4]. The main reason we use it, is because we already have some experience with it. Also some preparation has been done for this assignment in the form of a NQTHM file that we have modified and used. More about this in chapter 4.

NQTHM is a layer on top of the language *Common Lisp*. And it also uses most of the LISP commands. LISP functions can very easily be recognized. Let us say we have a function `f` with two parameters, namely x and y. In LISP this will be written as: `(f` x y`)`. Below we will give the most common commands that we have used in NQTHM. This is needed to make the NQTHM code we used in this paper more understandable for those who are not familiar with LISP or NQTHM.

3.3.1 Boolean operators

Below are the Boolean operators we use. They are not difficult to comprehend.

`(implies` x y`)`	:	$x \Rightarrow y$
`(and` $x \cdots y$`)`	:	$x \wedge \cdots \wedge y$
`(not` x`)`		$\neg x$
`(if` x y z`)`	:	**if** x **then** y **else** z **fi**

Note that `and` can have more than two parameters. The predicates *true* and *false* are denoted as `(true)` and `(false)` respectively. They can be abbreviated as `t` and `f`.

3.3.2 Arithmetic

Below are some standard functions applied on natural numbers a and b

`(add1 a)`	=	a + 1
`(sub1 a)`	=	a − 1
`(plus a b)`	=	a + b
`(equal` x y`)`	:	$x = y$
`(lessp a b)`	=	a < b

As can be seen, NQTHM never uses infix-operators. They are all prefix.

3.3.3 Functions on lists

Lists are used frequently our algorithm and are constructed as follows:

```
(list a b c d)   ≡  (cons a (cons b (cons c (cons d nil))))
```

Whenever possible NQTHM will try to evaluate the elements of the list. That is not always what we want. To prevent this, we can mark the elements with a quote. This results in the following list construction that we will use in our code.

```
'(a b c d)   ≡  (cons 'a (cons 'b (cons 'c (cons 'd nil))))
```

Let us say that a list `l = (cons x (cons y nil))` is given. Then the following rules are valid:

```
(car l)        =  x
(cdr l)        =  (cons y nil)
(listp l)      =  t
(member x l)   =  x ∈ l
```

As can be seen, `car` returns the header of the list and `cdr` returns the tail of the list. These two functions can also be combined. For example `(cddr l)` is the same as `(cdr (cdr l))`, `(caar l)` is `(car (car l))`, `(cadr l)` is equal to `(car (cdr l))` and so on. The function `listp` is a check to see if its parameter is a list or not. The function `(nlistp l)` is its counterpart, it returns `t` if its parameter is not a list.

3.3.4 Definitions and lemmas

The contents of the NQTHM file mainly consists of definitions and lemmas. How a definition can be implemented is shown below

```
(defn name ( parameters )
   commando )
```

It looks like a definition is a very small thing due to only one command can be defined here. But this command can be any NQTHM command , like for example `if` or `implies`. Those commands can also have nested commands in them. In this way a command can be extended a lot. It is up to the designer to make his code readable by using carriage returns and tabbing in a decent way.

Proofs concerning these definitions are mostly done using the `proof-lemma` and `lemma` constructs. First we will discuss the first one:

```
(prove-lemma name (rewrite)
   command
   [ hints ] )
```

In our case the command is mostly an `implies`. NQTHM will try to prove the command using everything it knows. Including previous proven lemmas. The *hints* part is placed between brackets, because that part is optional. Sometimes NQTHM need a number of hints to proof the lemma. The two hints we used in our barrier algorithm are: `(do-not-induct t)` and `(do-not-generalize t)`. The first one becomes very useful when you know that the lemma does not need any induction. With this hint enabled NQTHM can discover that a lemma is not correct sooner and easier. NQTHM often splits a lemma into several cases. When it cannot verify a case it will save the case for later to try and prove it with induction and continues verifying the other cases. After all these other cases are verified it returns to the case that still needs to be proven. Mostly it then find out it cannot prove the case with induction and gives a failure. With the `(do-not-induct t)` hint given it will returns immediately with a failure when NQTHM tries to verify the case that fails and leaves all the other cases alone. This saves the designer some time. The second hint tells NQTHM that it should not try to generalize a problem, this will also help to improve the

performance of NQTHM. One of the lemmas we made, failed without this option. It failed after generalizing the problem, but it went well with the option enabled. The hints are used in almost all our lemmas but they are omitted in the code mentioned in this paper. Most of the time the hints are not needed. Especially when a lemma is valid. The hints are mostly used for debugging the output NQTHM gives when a lemma fails.

The `lemma` construct looks almost the same as `prove-lemma`. The main difference is that it is far more efficient than `prove-lemma`. This is because all the lemmas that NQTHM needs to prove the current lemma has to be given explicitly. It does not check all the lemmas that NQTHM knows at that time. Like `prove-lemma` does.

All `lemma` and `prove-lemma` have one parameter namely `rewrite` This parameter is the only one we use in our NQTHM files. It replaces complicated terms by hopefully simpler ones depending on the contents of *command*. When NQTHM cannot verify a lemma directly it checks if there are already other rewrite lemma proven that it can use. It replaces the old terms by new ones defined in a earlier proven rewrite lemma. After applying the rewrite lemma to the lemma that failed the failed lemma can then maybe be proven.

Lemmas can be enabled and disabled in NQTHM at any time, by the commands (`enable` *lemmas*) and (`disable` *lemmas*) respectively. This makes analyzing NQTHM's output easier. When NQTHM fails to verify a lemma, the output can be very complicated. When certain lemmas are disabled the output can be easier to understand, because unnecessary evaluations are eliminated.

Now that we have discussed all the theory we need to understand our NQTHM code we can move to our NQTHM design of the barrier algorithm.

Chapter 4

The event-file concprelude

This event file `concprelude.events` is a prelude that can be used for proving many concurrent algorithms and is made by Wim. H. Hesselink. The main function of this event-file is to recognize and evaluate the semantics of a simple concurrent program with both shared and private variables. The main function `exe` in this prelude changes the global state x depending on the atomic command a process executes.

The state x is modeled by a pair of lists: the list of private states and the list of shared variables. The first list has sublists as elements. The header of each of these sublists contains the name of the process and the tail contains the private variables of the process including their corresponding values. The second list of the pair representing state x is a list of the global variables with their corresponding values. A graphical representation is given in figure 4.1.

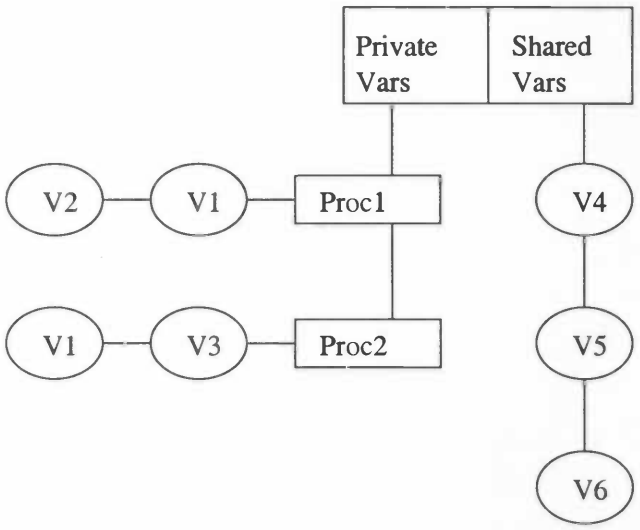

Figure 4.1: A graphical representation of state x

Note that *proc1* and `proc2` in figure 4.1 both contain a variable $v1$. But they are not the same. The variable $v1$ of `proc1`, denoted by $v1.proc1$ does not have to be equal to $v2.proc2$. They are completely different variables.

The prelude recognizes the following commands:

- `(put` v w`)`: assigns the value w to the variable v
- `(if` e a b`)`: if the expression e is valid execute a else execute b.
- If the command is not recognized, it will not change the state

The barrier algorithm needs two additional commands, being:

- `(enter wq)`: Puts the process that executes this command into the waiting queue wq.
- `(broadcast wq)` Empties the waiting queue wq completely.

This results in the following NQTHM code:

```
(defn exe (d q cmd x)
    (cond ((nlistp cmd) x)
          ((nlistp (car cmd))
           (case (car cmd)
                 (put (putgen d q (cadr cmd)
                              (ev d q (caddr cmd) x)
                              x ))
                 (if (if (ev d q (cadr cmd) x)
                         (exe d q (caddr cmd) x)
                         (exe d q (cadddr cmd) x) ))
                 (enter (putglobal q (cadr cmd)
                          (cons q (lookup (cadr cmd) (cdr x))) x))
                 (broadcast (putglobal q (cadr cmd) NIL x))
                 (otherwise x) ) )
          (t (exe d q (cdr cmd)
                  (exe d q (car cmd) x) )) ) )
```

We did not have to integrate the new commands `enter` and `broadcast` into `exe`. We also could have implemented them in the form of separate definitons. But we saw them as commands of the same level as `put`. That is why we have integrated them into `exe`. The code contains two NQTHM commands we did not explain before: `case` and `cond`. The first one is a standard *case*-construct, like `switch` is in C or `case` in PASCAL. The second command `cond` is a extended form of the *if*-construct.

Before `exe` can be executed it has to satisfy some properties. Firstly the command ready to be executed has to be valid of course and secondly the process should not be disabled. Disabling a process means that it is not allowed to perform an execution. In our case it is in the waiting queue. But we will discuss this in the next section. Concurrent programs can use more than one queue. This means that if we want to use this file for more concurrent programs we cannot define here when a process is enabled or not because disabling depends on the concurrent algorithm itself. The `enabled` function needs to be different defined if a program has two queues instead of one. To solve this we use the command `apply$` which has as arguments the function name `enabled` which is not defined yet and `enabled`'s own arguments: process q and program state x. The `apply$` contains a dollar sign because the developers of NQTHM are still not satisfied with this function. With the dollar sign they can easily locate the function in their source code cf. [4]. Here you can see the `apply$` function integrated in the already existing function `exepc`.

```
(defn exepc (d q cmd x)
    (let ((rule (assoc (pc q x) cmd)))
         (if (and rule
                  (apply$ 'enabled (list q x)) )
             (exe d q (cdr rule)
                  (putlocal q 'pc
                            (add1 (pc q x))
                            x ) )
             x ) ) )
```

Before executing the command the process counter has to be increased otherwise `exe` will execute the same command over and over again. The complete source code of `concprelude.events` can be found in appendix B.

Chapter 5

The main event-file barrier

Now that we have done some preparation in concprelude, we can try to verify the algorithm itself. We begin by rewriting the algorithm to make it understandable for NQTHM. Then we define functions to retrieve the values of the shared and private variables from the program state. After that we will define functions that model the changes of a variable in the state of the program. Each variable will have its own function. We need those functions to prove the invariants we have defined in chapter 2. We will finish this chapter with the initialization of the invariants.

5.1 The algorithm itself

The algorithm defined in chapter 2 has to be rewritten to make it understandable for the execute functions in concprelude. To achieve this we have to expand **lock**, **unlock** and **wait** and use the commands that the prelude can understand. To simulate loops and if-statements with more than one atomic action in the body we manipulate the process counter *pc* to jump through the program.
The algorithm rewritten:

```
(defn barrier ()
  '(;(0  TNS
     (1  (if (equal m-owner 'bot) (put m-owner self) (put pc 1)))
     (2  (put cnt (add1 cnt)) )
     (3  (if (equal atbar (sub1 Nproc)) (put pc 4) (put pc 7)))
     (4    (put atbar 0))
     (5    (broadcast Qv))
     (6    (put pc 10))
        ;else
     (7    (put atbar (add1 atbar)) )
     (8    (if (equal m-owner self)
              ((put m-owner 'bot) (enter Qv))
              (put pc 8) ) )
     (9    (if (equal m-owner 'bot) (put m-owner self) (put pc 9)))
     (10 (if (equal m-owner self) (put m-owner 'bot) (put pc 10)))
     (11 (put pc 1)) ) )
```

The algorithm starts at process counter 0 with a terminating non critical section. Almost everything can be implemented here as long it does not modify the variables atbar, Nproc, cnt, m-owner and Qv. To make the proof more convenient we leave the TNS alone, and degrade it to a comment. Officially at pc 11 it should jump back again to 0 instead of 1.

The symbol $\perp$ has been replaced by the the identifier 'bot. At process counter 8 in the then-part of the if-statement there is a tuple of commands instead of just one. The reason for it is that those commands have to be executed in one atomic statement like defined in **wait** of chapter 2,

otherwise the algorithm is incorrect. We found this error while trying to prove the algorithm in NQTHM.

We changed the algorithm a number of times to make the invariants more convenient and easier to prove for NQTHM. The changes we made in the algorithm during the proof will be mentioned further on in this paper.

5.2 Retrieving variables from the state

To retrieve the values belonging to the variable names we have created several lookup functions. Each variable has its own function. When a variable is a private one, the function searches for the sublist corresponding to the process name the variable belongs to. After the variable is located the function searches for it value in that sublist. Finding a global variable is easier. The function just looks for its value in the second list of x.

The lookup functions:

```
(defn m-owner (x) (lookup 'm-owner (cdr x)))
(defn Nproc   (x) (lookup 'Nproc (cdr x)))
(defn atbar   (x) (lookup 'atbar (cdr x)))
(defn Qv      (x) (lookup 'Qv (cdr x)))
(defn cnt   (q x) (lookup 'cnt (privstate q x)))
```

Here `(lookup e lst)` is a function defined in `concprelude.events` that returns the value of element `e` in list `lst`. Now that we have defined the lookup function for the waiting queue, we can determine when a process is disabled or not. I.e. :

```
(defn enabled (q x) (not (member q (Qv x))))
```

This function has to be defined in this event-file, otherwise when this event file calls `exec-pc` in `concprelude.events` it will give an error that `enabled` does not exist. This barrier algorithm only uses one waiting queue. What we want is that when a process is in the waiting queue it is not allowed to do anything. On the other hand the process is allowed to perform executions when it is not in the queue, hence the definition `enabled` above.

We first used another method of disabling processes. We used a second list called `disableds` that contains all the disabled processes, because we did not think about the `apply$` approach first. Introducing such a list would mean more administration. Every process that entered the waiting queue had to be inserted in `disableds` manually. Also more invariants need to be proven with this method. Lemmas such as

$$q \in Q(v) \Rightarrow q \in \texttt{disableds}$$

With the `apply$` approach all this is not needed.

5.3 Lemmas about an execution step

An invariant of a concurrent system is a predicate that remains valid after every possible execution step of the system. So we need some sort of step function which performs such a step in our barrier algorithm:

```
(defn step (p x)   (exepc (dcl-p) p (barrier) x))
```

Here `exepc` is a function declared in the prelude. It gives the new state in terms of the old state x when process p takes a step in the algorithm specified by `barrier` and `dcl-p`. It checks if the process that wants to execute a step is enabled and if the command is valid. Before a call to `exe` it increases the process counter by one. `dcl-p` is a list of all the global variables that barrier needs i. e. `'(m-owner Nproc atbar Qv)`

With this step function we can now determine changes of the variables during an execution step. For example, the variable `Nproc` isn't really a variable but a constant. To prove that it is

a constant we need to prove that at each execution step that is possible in state x the value of Nproc does not change:

```
(prove-lemma Nproc-step (rewrite)
  (equal (Nproc (step p x)) (Nproc x)) )
```

Step-lemmas for variables that can be changed are a little different. For example, one can easily see that the variable *cnt* is only modified at instruction 2. In that instruction *cnt* is incremented by one. This is expressed in lemma:

```
(prove-lemma cnt-step (rewrite)
  (equal (cnt q (step p x))
         (if (and (equal p q)
                  (equal (pc p x) 2)
                  (enabled p x) )
             (add1 (cnt q x) )
             (cnt q x)) ) )
```

As can be seen `(cnt q x)` is only increased when it satisfies the following properties:

1. When the process that does the step is the same as q.
2. When the process is at process counter where the variable is going to be modified.
3. When the process is enabled.

In all the other cases, `(cnt q x)` is not changed. Every shared and private variable has such a step lemma and proving them is the same as we did with `cnt`, as can be found in appendix C

5.4 The invariants

We will now try to prove the invariants that we introduced in chapter 2.

Proving (WL)

First we want to prove (WL). Rewriting (WL) in NQTHM results in the following:

```
(defn wl (q x)
  (implies (member q (Qv x)) (equal (pc q x) 9)) )
```

To prove that this invariant is correct, (WL) has to stay valid on every possible execution step the concurrent program can make:

```
(prove-lemma wl-kept-valid (rewrite)
    (implies (wl q x)
             (wl q (step p x)) )
    ((do-not-induct t)) )
```

Here p can be any process. NQTHM did not have any problems verifying this invariant.

Proving Mutual Exclusion

It is very important to know the program's locations where the process can be when it holds the mutex. Indeed, from the code, one can see that the process holds the mutex, if and only if it is one of the locations 2 to 8 or 10. This is expressed in this invariant:

(JQ1) $\quad m\_owner = q \quad \equiv \quad pc.q \in \{2, 3, 4, 5, 6, 7, 8, 10\}$

Rewriting it to NQTHM-code and proving it:

```
(defn jq1 (q x)
    (equal (equal (m-owner x) q)
           (member (pc q x) '(2 3 4 5 6 7 8 10))) )

(prove-lemma jq1-kept-valid (rewrite)
    (implies (and (jq1 q x)
                  (not (equal q 'bot)))
             (jq1 q (step p x)) ) )
```

This invariant proves mutual exclusion and we will show this by example. If $pc.q = 2$ then it follows from JQ1 that q has the mutex. If q has it, another process p cannot have it, this implies to that $pc.p$ can only be at 1, 9 or 11 and cannot do anything until the mutex is released again.

Proving (M1)

The first invariant that is more difficult to prove is (M1). We need to change the invariant a little. When a process performs an execution step. The invariant can be violated. This is not something to worry about as long as the invariant stays valid, when no process performs any executions. In other words instead of proving (M1) we want to prove

(JQ2) $\quad m.owner = \bot \quad \Rightarrow \quad \texttt{atbar} = |Q(v)|$

A simple `m1-kept-valid` is not sufficient to prove this invariant. We know by (JQ1) that when $m.owner = \bot$ the process is at instruction 1, 9 or 11. Again (JQ2) is proven valid if it holds for every possible execution step in state x. But we do not know nothing about what will happen with $|Q(v)|$ and `atbar` when there is some other process for example at process counter 10 before the execution step. That is why we have to introduce some other invariants to handle those cases

(JQ2a) $\quad pc.q \in \{2,3,4,6,7,10\} \quad \Rightarrow \quad \texttt{atbar} = |Q(v)|$
(JQ2b) $\quad pc.q = 8 \quad \Rightarrow \quad \texttt{atbar} = |Q(v)| + 1$
(JQ2c) $\quad pc.q = 5 \quad \Rightarrow \quad \texttt{atbar} = 0$

We can now prove (JQ2) with these new subinvariants, but only if they themselves are valid. For proving each subinvariant in NQTHM, NQTHM needs a little help: it needs to distinguish whether process q does the step or another process does. Without this proving such a subinvariant is too difficult. To make it somewhat clearer we will lay out the proof for the invariant (JQ2a).

The first thing we need to do is rewriting it to NQTHM-code

```
(defn jq2a (q x)
  (implies (member (pc q x) '(2 3 4 6 7 10))
           (equal (atbar x) (len (Qv x))) ) )
```

This is pretty standard and the other subinvariants are rather similar. Those invariants are not given here, they can be found in appendix C. Now we want to prove the invariant when process p is the process that performs the execution step

```
(prove-lemma jq2a-eq (rewrite)
  (implies (and (jq2a q x)
                (jq2 x)
                (jq2b q x)
                (jq2c q x)
                (equal p q) )
           (jq2a q (step p x)) ) )
```

As it can be seen it needs all the subinvariants and the main invariant JQ2 to prove it. Now we will prove the case where q is not the process that performs the execution

```
(prove-lemma jq2a-dif (rewrite)
  (implies (and (jq2a q x)
                (jq1 p x)
                (jq1 q x)
                (not (equal p q)))
           (jq2a q (step p x)) )
  ((do-not-induct t)) )
```

As you can see we need (JQ1) here applied to both p and q. Mutual exclusion is needed because when both the processes are in the critical section (i.e. the section where a process has the mutex) the invariant will be invalidated. If (JQ2a) holds for process q it means that q has the mutex according to (JQ1). This means that process p, which is not equal to q, cannot modify `atbar` because it does not have the mutex. Thus is not at pc 4 or 7. NQTHM can now prove without any problem the following lemma

```
(lemma jq2a-kept-valid (rewrite)
  (implies (and (jq2a q x)
                (jq2 x)
                (jq2b q x)
                (jq2c q x)
                (jq1 p x)
                (jq1 q x)
                )
           (jq2a q (step p x)) )
    ((use (jq2a-dif) (jq2a-eq))
     (do-not-induct t)) )
```

The other subinvariants are proved in the same way. See for the code the appendix.

Proving (JQ2) itself:

```
(prove-lemma jq2-kept-valid (rewrite)
  (implies (and (jq2 x)
                (jq1 p x)
                (jq2a p x)
                (jq2b p x)
                (jq2c p x)
                (not (equal p 'bot)) )
           (jq2 (step p x)) )
  ((do-not-induct t)) )
```

Proving this main invariant is easier than its subinvariants. It involves only one process that can perform a step, since no process has the mutex according to (JQ1).

Proving (M2)

We now want to prove (M2) but here the problem starts. We have not been able to prove this invariant yet. We have tried many approaches. In chapter 6 we will discuss every approach we have tried and why they did not work. Below we will discuss what we have done so far. We are confident that this is the way to proof (M2). Only we did not have the time to complete the proof.

Like (M1) this invariant only needs to hold when no process has the mutex:

(JQ3') $\quad m.owner = \bot \quad \wedge \quad q \notin Q(v) \quad \wedge \quad r \in Q(v) \quad \Rightarrow \quad cnt.r = cnt.q + 1$

If we analyze this invariant we can simplify it by applying invariants we already know. We see that $m.owner = \bot$ needs to be valid. If it holds no process has the mutex.

With this fact known, according to (JQ1) and with the given fact that $q \notin Q(v)$ must hold also, we can rewrite (JQ3') to

(JQ3) $\quad pc.q \in \{1, 9, 11\} \quad \wedge \quad r \in Q(v) \quad \Rightarrow \quad cnt.r = cnt.q + 1$

The only thing we needed now are the additional invariants like we have done for (JQ2). We have tried to make those invariants, but we have run out of time at this point. The only thing we got so far is (JQ3) rewritten in NQTHM code

```
(defn jq3 (q r x)
  (implies (and (member (pc q x) '(1 9 11))
                (member r (Qv x)))
           (equal (cnt r x) (add1 (cnt q x))) ) )
```

Initialization

We have not done this part in NQTHM yet because we were not able to verify (M2). But we still can give a global view how it can be done. Proving that invariants are kept valid is not enough. They also have to be valid before the processes begin to execute. In other words they have to be probably initialized.

We will start with (WL). The only datastructure that needs to be initialized here is $Q(v)$. Before the executions start, all the processes are synchronized because no process can run ahead without doing execution steps. So all processes can be enabled without any problem. In other words $Q(v)$ can be initialized as an empty list and the mutex can be set to $\perp$. To see that (WL) will hold is trivial because since $Q(v)$ is empty the left-hand-side is false and false implies everything. The same goes for (JQ1) since the mutex is equal to $\perp$ it is not equal to any process q so (JQ1) also holds after initialization.

Because of $Q(v)$ being empty, `atbar` has to be initialized to 0. Otherwise (JQ2) will not be valid. The other JQ2 invariants will hold too because their left-hand-sides are all false. The invariant (JQ3) will also hold because r cannot be in $Q(v)$ initially.

The only datastructures that still need to be initialized are `Nproc` and *cnt*. `Nproc` is a constant representing the number of processes. As long `Nproc` is greater than 0, it should not cause any problems. In chapter 2 we introduced a hoare triple (H). To make it valid, we need to make every *cnt* of each process equal to a constant C.

We have now initialized all datastructures. Now we have to integrate them into NQTHM. What we want is that all invariants are valid with the right initialization. Thus we have to create an initial state `initstate` containing all the variables we need with their corresponding initial values. Let `jq*` be the conjunction of all the invariants introduced in `barrier.events` then for the initialization state `initstate` is correct if NQTHM verifies the following successfully.

```
(prove-lemma jq*-initstate (rewrite)
  (jq* initstate) )
```

But we cannot prove it at this point because we ran out of time.

Chapter 6

What went wrong?

In this chapter we will lay out all the obstacles we encountered during the proof of the barrier algorithm. At the beginning of this assignment, the barrier algorithm looked different than it does now. Below is the old algorithm given.

```
loop
    TNS;
    lock(m);
    cnt ++;
    atbar ++;
    if atbar = N then
        atbar := 0;
        while notEmpty(Q(v)) do
            signal(v);
        od
    else
        wait(v, m);
    fi;
v:  unlock(m)
end
```

6.1 Problems proving the (M1) invariant

One of the things that was not good about the old barrier algorithm, was the **while** statement. We used it because we did not realize `nil` existed in NQTHM. The loop did the same as a **broadcast**. We did not see a problem in that. We have implemented **signal** in such a way that it removed the first element of $Q(v)$. **signal** will only one element on each call. Because of this loop we had trouble defining the additional invariants for (JQ2). We were trying to design these invariants in terms of $Q(v)$ and `atbar`. But that did not turn out very well and took us quite some time.

But then it occurred to us that after the **while** is done $Q(v)$ must be empty because of the guard of the **if** statement. We know that $|\emptyset| = 0$ and also that `atbar` is equal to 0 at the same location of the algorithm. So then we came up with the `atbar` $= 0$ invariant if the program counter was located in the **while** statement, and completely ignored the $Q(v)$ which we first thought was impossible. We were very surprised to see that NQTHM did not have any problem verifying this new designed invariant at all. We did not think that NQTHM was smart enough to see that $Q(v)$ was empty after the loop. And we thought that we had to make that clear to NQTHM manually by leaving $Q(v)$ in all the additional invariants to prove (M1). If we would have implemented the **broadcast** command in right in the beginning of the program, we would have saved a lot of time.

6.2 Problems proving the (M2) invariant

This invariant is the biggest obstacle we have experienced in our assignment. We have been working on this invariant for months now and we have made a lot of mistakes while proving it. And we still have not proven it yet.

Proving the old (M2)

In the beginning of the assignment (M2) was different. Below is the old (M2) we now call (M2'):

(M2') $q \in Q(v) \Rightarrow cnt.q = (\texttt{MIN}\ r :: cnt.r) + 1$

Here the `MIN` quantifier returns the minimum of all the *cnt* variables in its domain. This quantifier has to be implemented in our NQTHM code. The first problem we encountered was that nothing is known about process r in the domain of `MIN`. That is why we created a new ghost variable called *plist*. It was a list that represented all the processes that were using the barrier algorithm. Introducing such a ghost variable means introducing more lemmas and definitions. First we needed a lemma that defined *plist*. We already know that `Nproc` is the number of processes so we can use

(PL1) $|plist| = \texttt{Nproc}$

We also now know that $Q(v)$ holds the processes that are currently waiting to be released. These processes must also be in *plist*. Thus

(PL2) $Q(v) \subseteq plist$

Then we defined the definition `mincount` to model the `MIN` quantifier.

```
(defn mincount (m lst)
  (if (nlistp lst) f
      (if (nlistp (cdr lst))
         (min (car lst) m)
         (mincount (min (car lst) m) (cdr lst))) ) ) )
```

After the construction of the new definitions and lemmas we created the additional invariants like we did for (JQ2) for all the possible *pc*'s process q can be in. After a while we were still no able to prove the invariants. The lemmas were too difficult for NQTHM, or there was an error somewhere that we have missed. A lot of time was used trying to find the right lemmas, but to no avail. We dropped this approach after finding the new (M2) we use now. Unfortunately we do not have the code anymore where this approach was implemented.

Proving (M2) using set theory

After hopelessly trying to verify (M2'), we analysed the invariant again and came to a new invariant (M2). This one looks very simple compared to (M2'). But regrettably that is not entirely true. The main problem is that there are two processes in the parameter list of the invariant. Our first reaction was that we need to design additional invariants for *every pc* of both processes. To write all these invariants down took quite some time. There are many cases that need to be written down. Thus we came to the following variants. Note that the invariants are based on the NQTHM definition `barrier` in chapter 5.

(JQ4a) $pc.q \in \{1,2,9,10,11\} \wedge q \notin Q(v) \wedge pc.r \in \{1,2,9,10,11\} \wedge r \notin Q(v) \Rightarrow cnt.r = cnt.q$

(JQ4b) $pc.q \in \{1,2,9,10,11\} \wedge q \notin Q(v) \wedge pc.r = 3 \wedge \texttt{atbar} = \texttt{Nproc} - 1 \Rightarrow cnt.r = cnt.q$

(JQ4c) $pc.q \in \{1,2,9,10,11\} \wedge q \notin Q(v) \wedge pc.r = 3 \wedge \texttt{atbar} \neq \texttt{Nproc} - 1 \Rightarrow cnt.r = cnt.q + 1$

(JQ4d) $pc.q \in \{1,2,9,10,11\} \wedge q \notin Q(v) \wedge pc.r \in \{7,8\} \Rightarrow cnt.r = cnt.q + 1$

(JQ4e) $pc.q \in \{1,2,9,10,11\} \wedge q \notin Q(v) \wedge pc.r \in \{5,6\} \Rightarrow cnt.r = cnt.q$

(JQ4f) $pc.q \in \{1,2,9,10,11\} \wedge q \notin Q(v) \wedge r \in Q(v) \Rightarrow cnt.r = cnt.q + 1$

With the old design of our algorithm we needed even more invariants. Let us call those invariants the (JQ4') invariants for convenience. In the (JQ4) invariants we only need the `atbar` $=$ `Nproc` -1 check in (JQ4b) and (JQ4c). On the other hand in the (JQ4') invariants, due to the location of the `atbar` incrementation, we had four invariants that needed to check the value of `atbar` instead of the two we have now. We needed two invariants depending on whether `atbar` $=$ `Nproc` -1 was valid or not when the process was at the `atbar ++` line. And two more invariants that checked `atbar` $=$ `Nproc` in the line after the `atbar` incrementation.

The first invariant that we tried to prove was (JQ4a'). It looked almost the same as (JQ4a) only the process counters were not the same. We were trying to prove the invariant where the process that performs the step was equal to q:

```
(prove-lemma jq4a'-eq1 (rewrite)
  (implies (and (jq4*' q r x)
                (jq4*' r q x)
                (equal p q) )
           (jq4a' q r (step p x)) ) )
```

Here is `jq4*'` nothing but a conjunction of the (JQ4') invariants. As can be seen we need two calls of `jq4*'` to get all the cases. After some effort, we managed to prove this. But it took more than 45 minutes on a 700 MHz system. In other words this lemma is proven but it took too much time. It would take almost a day to prove all the other invariants. That is why we redesigned the algorithm to what it is now. Especially relocating the `atbar` incrementation saves a lot of time because we now need less invariants. Proving (JQ4a) like we did with (JQ4a') can be done in about five minutes. That is still too long but acceptable.

Verifying (JQ4a) where process r performs a step also succeeded in about five minutes. But the proof where another process than q or r performs the step failed. Whatever we tried it kept failing, giving output like below:

```
(IMPLIES (AND (MEMBER Q (QV X))
              (EQUAL (PC (M-OWNER X) X) 5)
              (EQUAL (PC Q X) 9)
              (NOT (MEMBER R (QV X)))
              (NOT (EQUAL (M-OWNER X) R))
              (NOT (EQUAL (M-OWNER X) Q))
              (NOT (MEMBER (M-OWNER X) (QV X)))
              (EQUAL (PC R X) 1))
         (EQUAL (CNT R X) (CNT Q X))),

which we would normally try to prove by induction.  But since the
DO-NOT-INDUCT hint has been provided by the user, the proof attempt has

********** F  A  I  L  E  D **********
```

We have analyzed the code and have found what is wrong with it. When the pc is equal to 5 we know that it is the last process that enters the barrier. The guard of the **if** statement and according to (JQ3) all but the process that has the mutex should be in $Q(v)$. What NQTHM wants to prove here is not possible. If the running process is at pc 5, all the others should be at 9 according to (WL). We tried solving this problem using a new lemma:

$$\text{(SM)} \qquad |A| + 1 = |B| \ \wedge \ A \subseteq B \ \wedge \ x, y \in B \ \wedge \ x, y \notin A \ \Rightarrow \ x = y$$

We had to re-introduce *plist* again if we want to apply this lemma. Also we needed to prove (PL1) and (PL2). If we can prove (SM) the NQTHM failure shown above should not be possible because if the mutex owner is at pc 5, $|Q(v)| + 1 = |plist|$ should hold. Process r cannot be at pc 1. It can only be at pc 5 because according to (SM) r and the mutex owner should be the same. This makes the left-hand-side of the *implies* false, thus making the code above true.

The human mind can easily see that (SM) is true. Only NQTHM needs far more information to proof it. We needed a lot of additional lemmas which are not so easy to design for those who are inexperienced with NQTHM.

Some of them were very simple like

$$A \subseteq B \quad \Rightarrow \quad (\texttt{cdr}\ A) \subseteq B$$

Others needed far more work like

$$|A| + 1 = |B| \quad \wedge \quad A \subseteq B \quad \Rightarrow \quad |B \setminus A| = 1$$

After some effort and with the help of Wim Hesselink we could verify the following lemma.

```
(lemma sm (rewrite)
    (implies (and (isset x) (isset y) (subset x y)
                  (equal (add1 (len x)) (len y))
                  (member a y) (member b y)
                  (not (member a x)) (not (equal a b)) )
             (member b x) )
    ((use (length-1 (x (setminus x y)))) (enable memberSetMinus crit-1)) )
```

Note that the NQTHM code is a little different than (SM) itself but they are equivalent. We have rewritten it, because we need the `rewrite` parameter. Rewriting variables is not allowed in NQTHM. That is what happens when you leave $x = y$ in the right-hand-side of (SM). If this is true NQTHM wants to replace y by x and that is not allowed.

We verified (SM) in a separate event-file first. But when we wanted to integrate the code into our barrier file it gave some errors. The two definitions we made NQTHM already knew, namely `delete` and `subset`. But if we deleted those, it still could not prove it. We discovered that those two functions were defined in `weakfairness.events` and we did not understand why they did not work. After some research we discovered that they were disabled in `weakfairness.events`. After enabling the two definitions again in `barrier.events` recognizes the `subset` and `delete` commands fine.

Still we needed to integrate (SM) into (JQ4a). Unfortunately we did not succeed in doing that. After a while we tried another completely different approach to solve the problem we had with proving (JQ4a). The code for this approach can be found in appendix D.

Proving (M2) using a new ghost variable

The new approach was introducing a new ghost variable called *actives* that represented a list containing all processes that are not in $Q(v)$. This meant changing our barrier algorithm again.

```
loop
    TNS;
    lock(m);
    cnt ++;
    if atbar = N − 1 then
        atbar := 0;
        broadcast(v);
        actives:= plist;
    else
        atbar ++;
        delete (self, actives);
        wait(v, m);
    fi;
v:  unlock(m)
end
```

All our step functions and invariants had to be changed to be valid again due to changes in the process counters and new step functions about the constant `plist` and the list *actives* had to be introduced.

The first lemma we wanted to prove was.

(AP) $|actives| + \texttt{atbar} = \texttt{Nproc}$

The proof failed at the point where the executing process does the broadcast. We did not found out why, because we discovered the method we have layed out in chapter 5. The NQTHM code for this approach can be found in appendix E.

In our point of view with so many approaches we should be able to prove the invariant. But with each approach we reached some point where we could not go on or we found another approach that looked easier to prove. We have no idea how far we could have gone with the approach we layed out in chapter 5. On first site it looks easier to verify than all the other approaches we have tried. But it can end like any other approaches we have tried.

Chapter 7

Extensions that can be made

In this chapter we will discuss some extensions that can be done to our barrier implementation. Unfortunately we did not have the time to implement it.

7.1 Incrementing *cnt*

There may be situations where you like to have the ghostvariable *cnt* implemented in the code, for debugging purposes for example. The barrier as it is implemented at the moment will run into an error if the program runs a very long time and *cnt* is included in the code. This error will be an overflow because of *cnt*. As can be seen *cnt* only increases. And never decreases. This can result into an overflow. A way of solving this problem is introducing the modulo function $\oplus$. The definition of $\oplus$ will then be in NQTHM code for a constant M:

```
(defn oplus1 (n)
  (if (equal n M) 0 (add1 n)) )
```

The barrier algorithm will not change much. We only have to replace the *cnt* ++ command by *cnt*:= *cnt* $\oplus$ 1. In `barrier.events` this is done by replacing `(add1 cnt)` by `(oplus1 cnt)` in the definition `barrier`. Also changes must be made to the invariants. Because *cnt* of each process must now be changed according to the new Hoare Triple

(H') { *cnt.self* } `Barrier` { *cnt.self* $\oplus$ 1 }

Which will also result in a change in (M2). At a first glance proving these changes will not be that difficult. Only when the barrier algorithm is proven first without these changes of course.

7.2 Leaking waiting queues

Sometimes waiting queues can be leaky. This means that while a process is in the waiting queue it may wake up occasionally cf. [3]. If this is the case our barrier algorithm is not correct anymore. Waking up processes before the broadcast will definitely result in situations where $cnt.q > cnt.r+1$. for given processes q and r. This will make the (BC) invalid. To make it valid again we need an extra flag that has to be checked before a process may continue its executions.

Below is the adjusted barrier algorithm with the flag implemented

```
loop
    TNS;
    lock(m);
    cnt ++;
    if atbar = N − 1 then
        atbar := 0;
        flag:= (flag + 1) mod 2;
        broadcast(v);
    else
        atbar ++;
        ownflag:= flag;
        repeat
            wait(v, m);
v:      until ownflag ≠ flag;
    fi;
    unlock(m)
end
```

Two new variables are introduced here: the shared variable `flag` and the private variable `ownflag`. Each time a process leaves the queue it is put back into the queue again if the flag is not changed before the broadcast. If after the process has been released from the queue the flag has been changed and the process may continue. Proving this is more complex. We have introduced two more variables, thus more step functions have to be implemented. Also the **mod** operator has to be implemented and some of the invariants have to be adjusted.

Looking at these extensions, it can be seen why using a theorem prover can save a lot of work. Proving these extensions by hand will take a lot of time. Almost all the proof you have done has to be done all over again, while proving it with a theorem prover changing a couple of lemmas and definitions can be enough to verify the new extensions.

Chapter 8

Conclusions

First of all we would like to say we regret it very much that we have not been able to verify the algorithm yet. A lot of time has been wasted on staring at lemmas we could not verify. The intention of the assignment was to prove the barrier algorithm. After that was done we would make some extensions on it, like the extensions we have discussed in chapter 7. But we did not even managed to prove the barrier algorithm itself. This is mostly due to lack of experience because NQTHM is very experience dependant. When you first encounter a problem, it can sometimes take weeks or even months before you have found the solution. An experienced user can sometimes solve the same problem in less than a couple of hours, or even in a couple of minutes.

When NQTHM fails to verify a lemma an experienced user can see very quick what went wrong, because he or she can often recognize the problem in the generated output sooner than an inexperienced user could. We have worked on this assignment for more more than six months now. We have noticed indeed that experience is very important in NQTHM. A lot of small problems we have encountered when we just started this assignment are now easier to solve.

An important thing we learned is always make a log of all the things you have done during an assignment and always keep the source files, even when you have dropped the approach you were working on. This is what we have found the hard way. While we were trying to verify (M2) we accidentally deleted the file that contained the `mincount` approach we used to verify the old (M2'). As a result we cannot give the source code of this approach in this paper. Also we have made several definitions and lemmas in that file, we later on could use again. We had to create these all over again.

When you are working a theory problem like we did. Do not be afraid to drop an approach you have been working on. Sometimes trying a new approach will help you to reach your goal sooner than when you keep on trying the old approach. We have done it a lot of times during this assignment, unfortunately to no avail. Let us hope that someone else is more successful in solving this problem.

Bibliography

[1] Andrews, G.R.: Concurrent Programming, principles and practice.

[2] Hesselink, W.H., Jonker E.J.: Pthreads and applications of mutex-abstraction (version 227).

[3] Hesselink, W.H., Jonker E.J.: Pthreads and applications of mutex-abstraction (version 242).

[4] Boyer, R.S., Moore, S.J.: A Computational Logic Handbook.

Appendix A

A solution verifying (M2)

After finishing this paper Wim H. Hesselink came with a solution. The solution was found after finishing the assignment. That is why we discuss it in the appendix. The main problem with (M2) was the complexity of the (JQ4) invariants. One of the problems was that some of the cases will not even occur. For example let $pc.q$ be at 3 and let `atbar` be equal to `Nproc - 1`. This means that q is the last process entering the barrier, thus all the other processes are in the waiting queue. This will imply that (JQ4b) will never apply.

The solutions is not very complex. As can be seen in the code all processes q where $pc.q \in \{0, 1, 2, 6, 9, 10, 11\}$ holds or when q is not in the waiting queue, have their cnt variables equal. In all the other cases $cnt.q$ is increased by one. This results in the following:

$$\begin{aligned} \text{CorrCnt}(q) = \quad & \textbf{if } q \in Q(v) \vee pc.q \in \{3, 4, 5, 7, 8\} \\ & \textbf{then } cnt.q \textbf{ else } cnt.q + 1 \end{aligned}$$

(KQ0) $\quad \text{CorrCnt}(q) = \text{CorrCnt}(r)$

Let q be the process that is in `TNS`, this implies that $pc.q$ is equal to 0, and let process r be in the waiting queue. According to (KQ0) $cnt.q + 1 = cnt.r$ will hold in this case, thus (KQ0) will imply (M2). For the proof an additional invariant had to be created for the case when a process is located at process counter 4 or 5.

(KQ1) $\quad pc.q \in \{4, 5\} \quad \Rightarrow \quad q = r \quad \vee \quad r \in Q(v)$

For the proof these invariants are rewritten in NQTHM code. They needed several additional lemmas, like (SM). But they are now successfully verified.

Appendix B

Source code of the prelude

```
#| Prelude for concurrency:
   Sequential processes communicating by shared
   variables
   A preparation, e.g., for Peterson's algorithm
   for mutual exclusion.
   Wim H. Hesselink, September 8, 2000 |#

(note-lib "weakfairness")
(disable-theory t)
(enable-theory ground-zero)

(defn putassoc (var w x)
      (if (nlistp x) (cons (cons var w) nil)
          (if (equal var (caar x))
              (cons (cons var w) (cdr x))
              (cons (car x) (putassoc var w (cdr x))) ) ) )

(prove-lemma assoc-put (rewrite)
    (equal (assoc key (putassoc var w x))
           (if (equal key var) (cons var w)
               (assoc key x) ) ) )

(defn cleanalist (d x)
    (if (nlistp d) nil
        (cons (cons (car d) (cdr (assoc (car d) x)))
              (cleanalist (cdr d) x) ) ) )

(defn proper (x)
    (if (nlistp x) nil
        (cons (car x) (proper (cdr x))) ) )

(lemma strip-cars-cleanalist (rewrite)
    (equal (strip-cars (cleanalist d x))
           (proper d) )
    ((enable proper cleanalist)) )

(lemma assoc-cleanalist (rewrite)
    (equal (assoc var (cleanalist d x))
           (if (member var d) (cons var (cdr (assoc var x)))  f ) )
    ((enable cleanalist)) )

(defn shared (d x)    (cleanalist d (cdr x)) )
(defn privstate (q x) (cdr (assoc q (car x))) )

(defn ev (d q exp x)
   (eval$ t exp
          (append (shared d x)
                  (cons (cons 'self q) (privstate q x)) ) ) )
```

```
(defn putlocal (q var w x)
    (cons (putassoc q (putassoc var w
                                (privstate q x) )
                    (car x))
          (cdr x) ) )

(prove-lemma private-vars-putlocal (rewrite)
    (implies (not (equal p q))
             (equal (privstate q
                               (putlocal p var w x) )
                    (privstate q x) ) ) ) ; triv

(defn putglobal (q var w x)
    (cons (car x)
          (putassoc var w (cdr x)) ) )

(defn putgen (d q var w x)
    (if (member var d)
        (putglobal q var w x)
        (putlocal q var w x) ) )

(defn lookup (a x)  (cdr (assoc a x)))
(defn pc (q x)      (lookup 'pc (privstate q x)))

(defn exe (d q cmd x)
    (cond ((nlistp cmd) x)
          ((nlistp (car cmd))
           (case (car cmd)
                 (put (putgen d q (cadr cmd)
                              (ev d q (caddr cmd) x)
                              x ))
                 (if (if (ev d q (cadr cmd) x)
                         (exe d q (caddr cmd) x)
                         (exe d q (cadddr cmd) x) ))
                 (enter (putglobal q (cadr cmd)
                          (cons q (lookup (cadr cmd) (cdr x))) x))
                 (broadcast (putglobal q (cadr cmd) NIL x))
                 (otherwise x) ) )
          (t (exe d q (cdr cmd)
                  (exe d q (car cmd) x) )) ) )

(prove-lemma private-vars-exe (rewrite)
    (implies (not (equal p q))
             (equal (privstate q (exe d p cmd x) ) (privstate q x) ) ) )


(defn exepc (d q cmd x)
    (let ((rule (assoc (pc q x) cmd)))
         (if (and rule
                  (apply$ 'enabled (list q x)) )
             (exe d q (cdr rule)
                  (putlocal q 'pc
                            (add1 (pc q x))
                            x ) )
             x ) ) )

(lemma private-vars-exepc (rewrite)
    (implies (not (equal p q))
             (equal (privstate q
                               (exepc d p cmd x) )
                    (privstate q x) ) )
    ((enable exepc private-vars-exe  private-vars-putlocal)) )

(make-lib "concprelude")
```

Appendix C

Source code of the barrier algorithm

```
(note-lib "concprelude")

(defn barrier ()
  '(;(0  TNS
     (1  (if (equal m-owner 'bot) (put m-owner self) (put pc 1)))
     (2  (put cnt (add1 cnt)) )
     (3  (if (equal atbar (sub1 Nproc)) (put pc 4) (put pc 7)))
     (4    (put atbar 0))
     (5    (broadcast Qv))
     (6    (put pc 10))
        ;else
     (7    (put atbar (add1 atbar)) )
     (8    (if (equal m-owner self)
              ((put m-owner 'bot) (enter Qv))
              (put pc 8) ) )
     (9    (if (equal m-owner 'bot) (put m-owner self) (put pc 9)))
     (10 (if (equal m-owner self) (put m-owner 'bot) (put pc 10)))
     (11 (put pc 1)) ) )

(defn dcl-p () '(m-owner Nproc atbar Qv))

(defn m-owner (x) (lookup 'm-owner (cdr x)))
(defn Nproc   (x) (lookup 'Nproc (cdr x)))
(defn atbar   (x) (lookup 'atbar (cdr x)))
(defn Qv      (x) (lookup 'Qv (cdr x)))
(defn cnt   (q x) (lookup 'cnt (privstate q x)))

(defn enabled (q x) (not (member q (Qv x))))

(defn step (p x)   (exepc (dcl-p) p (barrier) x))

(defn new-pc (q x)
    (case (pc q x)
          ;(0 1)
          (1 (if (equal (m-owner x) 'bot) 2 1))
          (2 3)
          (3 (if (equal (atbar x) (sub1 (Nproc x))) 4 7))
          (4 5)
          (5 6)
          (6 10)
          (7 8)
          (8 (if (equal (m-owner x) q) 9 8))
          (9 (if (equal (m-owner x) 'bot) 10 9))
          (10 (if (equal (m-owner x) q) 11 10))
          (11 1)
          (otherwise (pc q x)) ) )
```

```
(prove-lemma pc-step-eq (rewrite)
    (equal (pc p (step p x))
           (if (not (enabled p x))
             (pc p x)
             (new-pc p x)) )
    ((do-not-induct t)) )

(prove-lemma pc-step-dif (rewrite)
    (implies (not (equal p q))
             (equal (pc q (step p x))
                    (pc q x) ) )
    ((do-not-induct t)) )

(lemma pc-step (rewrite)
    (equal (pc q (step p x))
           (if (and (equal p q)
                    (enabled p x) )
               (new-pc p x)
               (pc q x) ) )
    ((enable pc-step-eq) (use (pc-step-dif))
     (do-not-induct t)) )

(prove-lemma cnt-step (rewrite)
  (equal (cnt q (step p x))
         (if (and (equal p q)
                  (equal (pc p x) 2)
                  (enabled p x) )
             (add1 (cnt q x) )
             (cnt q x)) )
  ((do-not-induct t)) )

(prove-lemma atbar-step (rewrite)
  (equal (atbar (step p x))
         (if (and (equal (pc p x) 7)
                  (enabled p x) )
             (add1 (atbar x) )
             (if (and (equal (pc p x) 4)
                      (enabled p x) )
                 0
                 (atbar x)) ) )
  ((do-not-induct t)) )

(prove-lemma Nproc-step (rewrite)
  (equal (Nproc (step p x)) (Nproc x))
  ((do-not-induct t)) )

(prove-lemma m-owner-step (rewrite)
  (equal (m-owner (step p x))
         (if (enabled p x)
             (case (pc p x)
                   (1 (if (equal (m-owner x) 'bot)
                          p
                          (m-owner x)))
                   (8 (if (equal (m-owner x) p)
                          'bot
                          (m-owner x)))
                   (9 (if (equal (m-owner x) 'bot)
                          p
                          (m-owner x)))
                   (10 (if (equal (m-owner x) p)
                           'bot
                           (m-owner x)))
                   (otherwise (m-owner x)))
             (m-owner x)))
  ((do-not-induct t)) )
```

```
(prove-lemma Qv-step (rewrite)
  (equal (Qv (step p x))
     (if (enabled p x)
       (case (pc p x)
         (5 NIL)
         (8 (if (equal (m-owner x) p)
                (cons p (Qv x))
                (Qv x) ))
         (otherwise (Qv x)) )
       (Qv x) ) )
  ((do-not-induct t)) )

(defn len (lst)
  (if (listp lst)
      (add1 (len (cdr lst)))
     0))

(disable-theory
    (pc-step-dif pc-step-eq pc step m-owner Nproc atbar cnt Qv) )

(defn wl (q x)
  (implies (member q (Qv x)) (equal (pc q x) 9)) )

(prove-lemma wl-kept-valid (rewrite)
    (implies (wl q x)
             (wl q (step p x)) )
    ((do-not-induct t)) )

(defn jq1 (q x)
    (equal (equal (m-owner x) q)
           (member (pc q x) '(2 3 4 5 6 7 8 10))) )

(prove-lemma jq1-kept-valid (rewrite)
    (implies (and (jq1 q x)
                  (not (equal q 'bot)))
             (jq1 q (step p x)) )
    ((do-not-induct t)) )

(defn jq2 (x)
  (implies (equal (m-owner x) 'bot)
      (equal (atbar x) (len (Qv x))) ) )

(defn jq2a (q x)
  (implies (member (pc q x) '( 2 3 4 6 7 10))
           (equal (atbar x) (len (Qv x))) ) )

(defn jq2b (q x)
  (implies (member (pc q x) '(8))
           (equal (atbar x) (add1 (len (Qv x)))) ) )

(defn jq2c (q x)
  (implies (member (pc q x) '(5))
           (equal (atbar x) 0) ) )

(prove-lemma jq2a-eq (rewrite)
  (implies (and (jq2a q x) (jq2 x) (jq2b q x) (jq2c q x)
                (equal p q) )
           (jq2a q (step p x)) )
  ((do-not-induct t)) )

(prove-lemma jq2a-dif (rewrite)
  (implies (and (jq2a q x)
                (jq1 p x) (jq1 q x)
                (not (equal p q)))
           (jq2a q (step p x)) )
  ((do-not-induct t)) )
```

```
(lemma jq2a-kept-valid (rewrite)
  (implies (and (jq2a q x) (jq2 x) (jq2b q x) (jq2c q x)
                (jq1 p x) (jq1 q x))
           (jq2a q (step p x)) )
    ((use (jq2a-dif) (jq2a-eq))
     (do-not-induct t)) )

(prove-lemma jq2b-eq (rewrite)
  (implies (and (jq2b q x) (jq2 x) (jq2a q x) (jq2c q x)
                (equal p q) )
           (jq2b q (step p x)) )
  ((do-not-induct t)) )

(prove-lemma jq2b-dif (rewrite)
  (implies (and (jq2b q x)
                (jq1 p x) (jq1 q x)
                (not (equal p q)))
           (jq2b q (step p x)) )
  ((do-not-induct t)) )

(lemma jq2b-kept-valid (rewrite)
  (implies (and (jq2b q x) (jq2 x) (jq2a q x) (jq2c q x)
                (jq1 p x) (jq1 q x) )
           (jq2b q (step p x)) )
  ((use (jq2b-dif) (jq2b-eq))
   (do-not-induct t)) )

(prove-lemma jq2c-eq (rewrite)
  (implies (and (jq2c q x) (jq2 x) (jq2a q x) (jq2b q x)
                (equal p q) )
           (jq2c q (step p x)) )
  ((do-not-induct t)) )


(prove-lemma jq2c-dif (rewrite)
  (implies (and (jq2c q x)
                (jq1 p x) (jq1 q x)
                (not (equal p q)))
           (jq2c q (step p x)) )
  ((do-not-induct t)) )

(lemma jq2c-kept-valid (rewrite)
  (implies (and (jq2c q x) (jq2 x) (jq2a q x) (jq2b q x)
                (jq1 p x) (jq1 q x) )
           (jq2c q (step p x)) )
  ((use (jq2c-dif) (jq2c-eq))
   (do-not-induct t)) )

(prove-lemma jq2-kept-valid (rewrite)
  (implies (and (jq2 x) (jq2a p x) (jq2b p x) (jq2c p x)
                (jq1 p x)
                (not (equal p 'bot)) )
           (jq2 (step p x)) )
  ((do-not-induct t)) )

(defn jq3 (q r x)
  (implies (and (member (pc q x) '(1 9 11))
                (member r (Qv x)))
           (equal (cnt r x) (add1 (cnt q x))) ) )

#| Here the proof ends |#
```

Appendix D

Source code of the barrier algorithm using (SM)

Here is only the part given that is different from the code given in C

```
(enable subset) (enable delete)
#| They are located in weakfairness.events |#

(defn isSet (x)
  (if (nlistp x) t
      (and (not (member (car x) (cdr x)))
           (isSet (cdr x))) ) )

(defn setMinus (x y)
  (if (nlistp x) y
      (delete (car x) (setMinus (cdr x) y)) ) )

(prove-lemma len-of-non-empty-list (rewrite)
  (implies (listp y) (not (lessp (len y) 1))) )

(prove-lemma len-of-delete (rewrite)
  (implies (and (member x y) (isSet y))
           (equal (add1 (len (delete x y)))
                  (len y) ) ) )

(prove-lemma isSetdelete (rewrite)
  (implies (isSet y)
           (isSet(delete a y)) )
  ((do-not-generalize t)) )

(prove-lemma isSetcdr (rewrite)
  (implies (isSet y)
           (isSet (cdr y)) )
  ((do-not-generalize t)) )

(prove-lemma isSetsetMinus (rewrite)
    (implies (isset y)
             (isset (setminus x y)) ) )

(prove-lemma subsetdelete (rewrite)
  (implies (and (not (member a x)) (subset x y) )
           (subset x (delete a y)) )
  ((do-not-generalize t)) )

(prove-lemma memberSetminus (rewrite)
  (implies (and (not (member a x))
                (member a y) )
           (member a (setminus x y)) )
  ((do-not-generalize t)) )
```

```
(prove-lemma subsetcdr (rewrite)
  (implies (subset x y)
           (subset (cdr x) y) )
  ((do-not-generalize t)) )

(prove-lemma len-of-empty-list (rewrite)
  (equal (equal (len y) 0)
         (nlistp y) ) )

(prove-lemma len-of-deleteplus (rewrite)
    (implies (and (member a x)
                  (isset x) )
             (equal (add1 (plus w (len (delete a x))))
                    (plus w (len x)) ) ) )

(prove-lemma lenght-of-setMinus (rewrite)
  (implies (and (isSet x) (isSet y)
                (subset x y) )
           (equal (plus (len x)
                        (len (setMinus x y)))
                  (len y) ) )
  ((do-not-generalize t)) )

#| made by Wim H. Hesselink |#
(lemma crit-1 (rewrite)
    (implies (and (isset x) (isset y)
                  (subset x y)
                  (equal (add1 (len x)) (len y)) )
             (equal (len (setminus x y)) 1) )
    ((use (lenght-of-setminus))
     (do-not-induct t) ) )

(prove-lemma length-1 (rewrite)
    (implies (and (member a x) (member b x)
                  (not (equal a b)) )
             (lessp 1 (len x)) ) )

(lemma sm (rewrite)
    (implies (and (isset x) (isset y)
                  (subset x y)
                  (equal (add1 (len x)) (len y))
                  (member a y) (member b y)
                  (not (member a x))
                  (not (equal a b)) )
             (member b x) )
    ((use (length-1 (x (setminus x y))))
     (enable memberSetMinus crit-1)
     (do-not-induct t) ) )
#| end of Hesslinks code |#


(defn jq4a (q r x)
  (implies (and (member (pc q x) '(1 2 9 10 11))
                (not (member q (Qv x)))
                (member (pc r x) '(1 2 9 10 11))
                (not (member r (Qv x))) )
           (equal (cnt r x) (cnt q x)) ) )

(defn jq4b (q r x)
  (implies (and (member (pc q x) '(1 2 9 10 11))
                (not (member q (Qv x)))
                (equal (pc r x) 3)
                (equal (atbar x) (sub1 (Nproc x))) )
           (equal (cnt r x) (cnt q x)) ) )
```

```
(defn jq4c (q r x)
  (implies (and (member (pc q x) '(1 2 9 10 11))
                (not (member q (Qv x)))
                (equal (pc r x) 3)
                (not (equal (atbar x) (sub1 (Nproc x)))) )
           (equal (cnt r x) (add1 (cnt q x))) ) )

(defn jq4d (q r x)
  (implies (and (member (pc q x) '(1 2 9 10 11))
                (not (member q (Qv x)))
                (member (pc r x) '(7 8)) )
           (equal (cnt r x) (add1 (cnt q x))) ) )

(defn jq4e (q r x)
  (implies (and (member (pc q x) '(1 2 9 10 11))
                (not (member q (Qv x)))
                (member (pc r x) '(5 6))
                (not (member r (Qv x))) )
           (equal (cnt r x) (cnt q x)) ) )

(defn jq4f (q r x)
  (implies (and (member (pc q x) '(1 2 9 10 11))
                (not (member q (Qv x)))
                (member r (Qv x)) )
           (equal (cnt r x) (add1 (cnt q x))) ) )


(defn jq4* (q r x)
  (and (jq4a q r x) (jq4b q r x) (jq4c q r x) (jq4d q r x)
       (jq4e q r x) (jq4f q r x) ) )

(prove-lemma jq4a-eq1 (rewrite)
  (implies (and (jq4* q r x) (jq4* r q x)
                (equal p q) )
           (jq4a q r (step p x)) )
  ((do-not-induct t)) )

(prove-lemma jq4a-eq2 (rewrite)
  (implies (and (jq4* q r x) (jq4* r q x)
                (equal p r) )
           (jq4a q r (step p x)) ) )


(prove-lemma jq4a-dif (rewrite)
;this one fails
  (implies (and (jq4a q r x)
                (member q plist) (member r plist)
                (isSet plist) (isSet (Qv x))
                (subset (Qv x) plist)
                (jq1 p x) (jq1 q x) (jq1 r x)
                (wl q x) (wl r x)
                (not (equal p r)) (not (equal p q)))
           (jq4a q r (step p x)) )
  ((do-not-induct t)) )
```

Appendix E

Source code of the barrier algorithm using a ghost variable

```
(note-lib "concprelude")

(defn barrier ()
  '(;(0  TNS
     (1  (if (equal m-owner 'bot) (put m-owner self) (put pc 1)))
     (2  (put cnt (add1 cnt)) )
     (3  (if (equal atbar (sub1 Nproc)) (put pc 4) (put pc 8)))
     (4    (put atbar 0))
     (5    (broadcast Qv))
     (6    (put actives procs))
     (7    (put pc 12))
        ;else
     (8    (put atbar (add1 atbar)) )
     (9    (put actives (delete self actives)))
     (10   (if (equal m-owner self)
             ((put m-owner 'bot) (enter Qv))
             (put pc 10) ) )
     (11    (if (equal m-owner 'bot) (put m-owner self) (put pc 11)))
     (12 (if (equal m-owner self) (put m-owner 'bot) (put pc 12)))
     (13 (put pc 1)) ) )

(defn dcl-p () '(m-owner Nproc atbar Qv procs actives))
(defn m-owner (x) (lookup 'm-owner (cdr x)))
(defn Nproc   (x) (lookup 'Nproc (cdr x)))
(defn atbar   (x) (lookup 'atbar (cdr x)))
(defn procs   (x) (lookup 'procs (cdr x)))
(defn Qv      (x) (lookup 'Qv (cdr x)))
(defn actives (x) (lookup 'actives (cdr x)))
(defn cnt   (q x) (lookup 'cnt (privstate q x)))
(defn enabled (q x) (not (member q (Qv x))))
(defn step (p x)   (exepc (dcl-p) p (barrier) x))

(defn new-pc (q x)
    (case (pc q x)
          ;(0 1)
          (1 (if (equal (m-owner x) 'bot) 2 1))
          (2 3)
          (3 (if (equal (atbar x) (sub1 (Nproc x))) 4 8))
          (4 5) (5 6) (6 7)
          (7 12)
          (7 8) (8 9) (9 10)
          (10 (if (equal (m-owner x) q) 11 10))
          (11 (if (equal (m-owner x) 'bot) 12 11))
          (12 (if (equal (m-owner x) q) 13 12))
          (13 1)
          (otherwise (pc q x)) ) )
```

```
(prove-lemma pc-step-eq (rewrite)
    (equal (pc p (step p x))
           (if (not (enabled p x))
             (pc p x)
             (new-pc p x)) )
    ((do-not-induct t)) )

(prove-lemma pc-step-dif (rewrite)
    (implies (not (equal p q))
             (equal (pc q (step p x))
                    (pc q x) ) )
    ((do-not-induct t)) )

(lemma pc-step (rewrite)
    (equal (pc q (step p x))
           (if (and (equal p q)
                    (enabled p x) )
               (new-pc p x)
               (pc q x) ) )
    ((enable pc-step-eq) (use (pc-step-dif))
     (do-not-induct t)) )

(prove-lemma cnt-step (rewrite)
  (equal (cnt q (step p x))
         (if (and (equal p q)
                  (equal (pc p x) 2)
                  (enabled p x) )
             (add1 (cnt q x) )
             (cnt q x)) )
  ((do-not-induct t)) )

(prove-lemma atbar-step (rewrite)
  (equal (atbar (step p x))
         (if (and (equal (pc p x) 8)
                  (enabled p x) )
             (add1 (atbar x) )
             (if (and (equal (pc p x) 4)
                      (enabled p x) )
                 0
                 (atbar x)) ) )
  ((do-not-induct t)) )

(prove-lemma Nproc-step (rewrite)
  (equal (Nproc (step p x)) (Nproc x))
  ((do-not-induct t)) )

(prove-lemma procs-step (rewrite)
  (equal (procs (step p x)) (procs x))
  ((do-not-induct t)) )  ; [ 0.0 2.5 0.2 ]

(prove-lemma m-owner-step (rewrite)
  (equal (m-owner (step p x))
         (if (enabled p x)
             (case (pc p x)
                   (1 (if (equal (m-owner x) 'bot)  p   (m-owner x)))
                   (10 (if (equal (m-owner x) p)   'bot (m-owner x)))
                   (11 (if (equal (m-owner x) 'bot) p   (m-owner x)))
                   (12 (if (equal (m-owner x) p)   'bot (m-owner x)))
                   (otherwise (m-owner x)))
             (m-owner x)))
  ((do-not-induct t)) )
```

```
(prove-lemma Qv-step (rewrite)
  (equal (Qv (step p x))
     (if (enabled p x)
       (case (pc p x)
         (5 NIL)
         (10 (if (equal (m-owner x) p)
                 (cons p (Qv x))
                 (Qv x) ))
         (otherwise (Qv x)) )
       (Qv x) ) )
  ((do-not-induct t)) )

(enable delete)

(prove-lemma actives-step (rewrite)
  (equal (actives (step p x))
    (if (enabled p x)
       (case (pc p x)
          (6 (procs x))
          (9 (delete p (actives x)))
          (otherwise (actives x)) )
       (actives x) ) )
  ((do-not-induct t)) )

#| definitions about sets |#
(enable subset)

(defn len (lst)
  (if (listp lst)  (add1 (len (cdr lst)))
     0))

(defn isSet (x)
  (if (nlistp x) t
      (and (not (member (car x) (cdr x)))
           (isSet (cdr x))) ) )

(disable-theory
    (pc-step-dif pc-step-eq pc step m-owner Nproc atbar cnt Qv procs actives) )

(defn procs-are-sets (x)
  (and (isSet (procs x))
       (isSet (Qv x))
       (isSet (actives x))
       (subset (Qv x) (procs x))
       (subset (actives x) (procs x)) ) )

(defn wl (q x)
  (implies (member q (Qv x)) (equal (pc q x) 11)) )

(prove-lemma wl-kept-valid (rewrite)
    (implies (wl q x)
                 ;(not (equal q 'bot)))
             (wl q (step p x)) )
    ((do-not-induct t)) )

(prove-lemma member-delete (rewrite)
    (equal (member y (delete x xs))
           (and (member y xs)
                (not (equal x y)) ) ) )

(defn qna (q x)
  (implies (and
       (isSet (procs x)) (isSet (Qv x)) (isSet (actives x))
       (subset (Qv x) (procs x))
       (subset (actives x) (procs x))
                (member q (actives x)) )
           (member (pc q x) '(1 2 3 4 5 6 7 8 9 12 13 )) ) )
```

```
(prove-lemma qna-kept-valid (rewrite)
    (implies (qna q x)
             (qna q (step p x)) )
    ((do-not-induct t)) )
;te bewijzen als q niet de muteex heeft dan pc q x != 2..8, 10

(defn jq1 (q x)
    (equal (equal (m-owner x) q)
           (member (pc q x) '(2 3 4 5 6 7 8 9 10 12))) )

(prove-lemma jq1-kept-valid (rewrite)
    (implies (and (jq1 q x)
                  (not (equal q 'bot)))
             (jq1 q (step p x)) )
    ((do-not-induct t)) )

(defn jq2 (x)
  (implies (equal (m-owner x) 'bot)
      (equal (atbar x) (len (Qv x))) ) )

(defn jq2a (q x)
  (implies (member (pc q x) '( 2 3 4 6 7 8 12))
           (equal (atbar x) (len (Qv x))) ) )

(defn jq2b (q x)
  (implies (member (pc q x) '(9 10))
           (equal (atbar x) (add1 (len (Qv x)))) ) )

(defn jq2c (q x)
  (implies (member (pc q x) '(5))
           (equal (atbar x) 0) ) )

(prove-lemma jq2a-eq (rewrite)
  (implies (and (jq2a q x) (jq2 x) (jq2b q x) (jq2c q x)
                (equal p q) )
           (jq2a q (step p x)) )
  ((do-not-induct t)) )

(prove-lemma jq2a-dif (rewrite)
  (implies (and (jq2a q x)
                (jq1 p x) (jq1 q x)
                (not (equal p q)))
           (jq2a q (step p x)) )
  ((do-not-induct t)) )

(lemma jq2a-kept-valid (rewrite)
  (implies (and (jq2a q x) (jq2 x) (jq2b q x) (jq2c q x)
                (jq1 p x) (jq1 q x) )
           (jq2a q (step p x)) )
    ((use (jq2a-dif) (jq2a-eq))
     (do-not-induct t)) )

(prove-lemma jq2b-eq (rewrite)
  (implies (and (jq2b q x) (jq2 x) (jq2a q x) (jq2c q x)
                (equal p q) )
           (jq2b q (step p x)) )
  ((do-not-induct t)) )

(prove-lemma jq2b-dif (rewrite)
  (implies (and (jq2b q x)
                (jq1 p x) (jq1 q x)
                (not (equal p q)))
           (jq2b q (step p x)) )
  ((do-not-induct t)) )
```

```
(lemma jq2b-kept-valid (rewrite)
  (implies (and (jq2b q x) (jq2 x) (jq2a q x) (jq2c q x)
                (jq1 p x) (jq1 q x) )
           (jq2b q (step p x)) )
  ((use (jq2b-dif) (jq2b-eq))
   (do-not-induct t)) )

(prove-lemma jq2c-eq (rewrite)
  (implies (and (jq2c q x) (jq2 x) (jq2a q x) (jq2b q x)
                (equal p q) )
           (jq2c q (step p x)) )
  ((do-not-induct t)) )


(prove-lemma jq2c-dif (rewrite)
  (implies (and (jq2c q x)
                (jq1 p x) (jq1 q x)
                (not (equal p q)))
           (jq2c q (step p x)) )
  ((do-not-induct t)) )

(lemma jq2c-kept-valid (rewrite)
  (implies (and (jq2c q x) (jq2 x) (jq2a q x) (jq2b q x)
                (jq1 p x) (jq1 q x) )
           (jq2c q (step p x)) )
  ((use (jq2c-dif) (jq2c-eq))
   (do-not-induct t)) )

;Proving the invariant itself.
(prove-lemma jq2-kept-valid (rewrite)
  (implies (and (jq2 x) (jq2a p x) (jq2b p x) (jq2c p x)
                (jq1 p x)
                (not (equal p 'bot)) )
           (jq2 (step p x)) )
  ((do-not-induct t)) )

(defn jq3 (x)
  (equal (Nproc x) (len (procs x))) )

(prove-lemma jq3-kept-valid (rewrite)
  (implies (jq3 x) (jq3 (step p x)))
  ((do-not-induct t)) )

(defn jq4 (x)
  (implies (equal (m-owner x) 'bot)
           (equal (plus (len (actives x)) (atbar x))
                  (Nproc x) ) ) )

(defn jq4a (q x)
  (implies (member (pc q x) '(2 3 4 7 8 10 12))
           (equal (plus (len (actives x)) (atbar x))
                  (Nproc x) ) ) )

(defn jq4b (q x)
  (implies (member (pc q x) '(5 6))
           (equal (atbar x) 0) ) )

(defn jq4c (q x)
  (implies (member (pc q x) '(9))
           (equal (plus (len (actives x)) (atbar x))
                  (add1(Nproc x)) ) ) )

(prove-lemma jq4a-eq (rewrite) ;this one fails
  (implies (and (jq4a q x) (jq4 x) (jq4b q x) (jq4c q x)
                (equal p q) )
           (jq4a q (step p x)) )
  ((do-not-induct t)) )
```