Journal of Universal Computer Science, vol. 11, no. 5 (2005), 744-770
submitted: 30/11/04, accepted: 31/1/05, appeared: 28/5/05 © J.UCS

Formal Construction of a Non-blocking Concurrent Queue Algorithm (a Case Study in Atomicity)

Jean-Raymond Abrial
(ETH Zürich, Switzerland
`jabrial@inf.ethz.ch`)

Dominique Cansell
(Université de Metz & LORIA, France
`cansell@loria.fr`)

Abstract: This paper contains a completely formal (and mechanically proved) development of some algorithms dealing with a linked list supposed to be shared by various processes. These algorithms are executed in a highly concurrent fashion by an unknown number of such independent processes. These algorithms have been first presented in [MS96] by M.M. Michael and M.L. Scott. Two other developments of the same algorithms have been proposed recently in [YS03] (using the 3VMC Model Checker developed by E. Yahav) and in [DGLM04] (using I/O Automata and PVS).

Key Words: atomicity, concurrency, refinement, formal proof, prover.
Category: D.1.3

1 Introduction

This paper[1] contains a case study in concurrency and atomicity. The example we study here has been published a long time ago by M.M. Michael and M.L. Scott in [MS96]. Their paper presented a number of algorithms for accessing and modifying some global data structures in a *concurrent fashion*. Among those, we have selected two very interesting algorithms whose simple purpose is to dequeue or enqueue nodes in a pointer constructed linear queue. The algorithms, which are presented in their paper in the form of C-like pseudo-code, seem to be extremely efficient (they give some statistical results supporting this assertion) although they appear to be not so obvious to fully understand (at least for us). Moreover, Michael and Scott gave some informal mathematical proofs of the correctness of their algorithms. As usual with proofs of that kind, it is rather hard to be certain to be completely convinced (at least, this is what happened to us). For all these reasons, we found it very interesting and challenging to see whether it would be possible to undertake a completely formal (and proved) re-construction of these algorithms. In other words, rather than verifying the algorithms as given by Michael and Scott, we redevelop them from scratch. As will be seen, we end up with some algorithms which are slightly different from theirs (we shall explain where and why). Correct algorithm construction is a task which is quite different from that of final verification. We

[1] This work has been partly supported by IST FP6 Rigorous Open Development Environment for Complex Systems (RODIN, IST-511599) Project.

think that it gives more insight on the very nature of the algorithm.

The paper is organized in the following way. Section 2 contains an informal definition of the global queue and of the two operations that could be performed on it. These definitions are given as if the operations were to be executed in a non-concurrent fashion. Section 3 contains the *concurrency and atomicity assumptions* which are to be followed in the final algorithms. Section 4 contains informal transformations of the definitions given in section 2 in order to take care of the assumptions that were defined in section 3. At this point the algorithms present themselves in the form of a number of successive atomic actions. So far nothing is formal, we just wanted to make very clear what we might eventually obtain. And, of course, nothing guarantees that the transformations done in section 4 are correct (in fact they are not). Section 5 contains a survey of the formal development technique which we shall use. Section 6 contains explanations of our development: it is done using 7 gradually refined models. Section 8 contains some comparisons of our work with similar recent studies done on the same example. Finally, section 9 concludes this study.

2 Defining the Queue and its Basic Operations

2.1 The Queue

The queue is made of a number of nodes linked by pointers (called $Next$) and forming a linear list. The last element of the queue is a “dummy” node called $Null$. We call $Null$ a dummy because it does not contain any useful information. The queue is accessed through two extra pointers called $Head$ (pointing to the first node of the queue) and $Tail$ (normally pointing to the node preceding $Null$ in the queue).

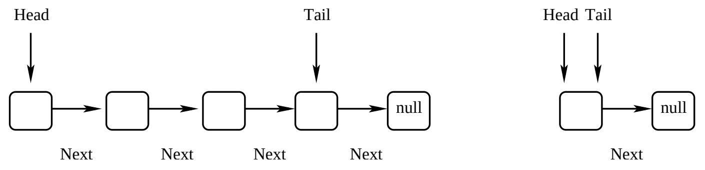

Figure 1: The Queue (normal and degraded)

The nodes belonging to the queue are those nodes connected by $Next$ ranging from the node pointed to by $Head$ to the $Null$ node. Notice however that the first node of the queue is in fact a dummy node (according to [MS96]). The queue could be in one of two states: normal (when $Head$ and $Tail$ are pointing to different nodes in the queue) or degraded (when both $Head$ and $Tail$ point to the last node of the queue) as indicated in Figure 1. In that latter case, the queue is indeed “empty”.

2.2 The Dequeue Operation

The queue can be modified by means of two operations called Dequeue and Enqueue. Operation Dequeue can result in a success or in a failure. It is a success in case it is performed on a normal queue, and it is a failure in case an attempt is made to perform it on a degraded queue (in which case the queue is left unchanged). Figure 2 shows a successful application of Dequeue: the first node in the queue is removed as a consequence of moving ahead the $Head$ pointer.

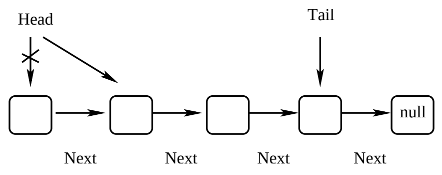

Figure 2: A successful Dequeue operation

2.3 The Enqueue Operation

Applying operation Enqueue results in adding an extra node by the end of the queue (preceding $Null$) as indicated in Figure 3. The extra node in question is one that is not in the queue: we suppose that we always have a sufficient supply of such extra nodes.

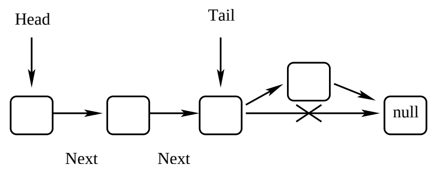

Figure 3: The Enqueue Operation (provided condition "$Next(Tail) = \mathsf{Null}$" holds)

After applying operation Enqueue, the $Tail$ pointer is not moved one step ahead. It results in the queue having one of the two shapes indicated in Figure 4. Operation Enqueue cannot be performed on a queue of these forms, but operation Dequeue can still be performed.

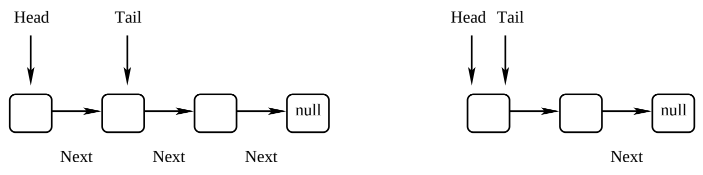

Figure 4: Possible queue shapes after an Enqueue operation

2.4 The Adapt Operation

A third operation, called Adapt, is thus necessary to move forward the $Tail$ pointer in case it does not point to the node preceding Null in the queue. Such an operation is illustrated in Figure 5. Thanks to operation Adapt, and Enqueue, $Tail$ always points either to the node n preceding $Null$ in the queue or to the node preceding n. Notice that in what follows, operation Adapt will be merged with operations Dequeue and Enqueue when necessary so that there will not exist such an independent Adapt operation.

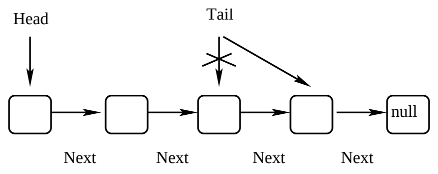

Figure 5: The Adapt Operation (provided condition “$Next(Tail) \neq \mathsf{Null}$” holds)

2.5 Non-concurrent versions of the Operations

The three operations we have just informally described and illustrated can be given more precise definitions by means of some “programs” written in a pseudo-code as indicated in Figure 6. Note that we also have an initialization defining an initial degraded queue.

Dequeue	Enqueue	Adapt
if $Head \neq Tail$ **then** $Head := Next(Head)$; **return**$(true)$ **else** **return**$(false)$ **end**	**if** $Next(Tail) = \mathsf{Null}$ **then** $nde :=$ new_node; $Next(nde) := \mathsf{Null}$; $Next(Tail) := nde$ **end**	**if** $Next(Tail) \neq \mathsf{Null}$ **then** $Tail := Next(Tail)$ **end**

Figure 6: The Three Operations

3 Concurrency and Atomicity Assumptions

The three operations mentioned in the previous section were defined as if activated by a single process. The problem, which was studied in the original paper [MS96], and which we shall study here again, is to have *several processes* willing to concurrently perform these operations. These concurrent processes perform these operations under certain assumptions which are the following:

1. There does not exist any critical section around the queue. In other words, processes can access the queue in a genuinely concurrent fashion by means of a succession of *elementary atomic actions*. An atomic action is an elementary modification of the queue and of its pointers $Head$ and $Tail$ in a way which is *guaranteed* to be done by a single process at a time. The global behavior of these processes is thus one where all such *atomic actions are interleaved*.

2. We have a number of global "variables" corresponding to the various nodes (inside and outside the queue) and to their connections (which we call $Next$ in Figures 1 to 5). Note that the two pointers $Head$ and $Tail$ are also global variables. As a practical convention, names of global variables all start with an upper case letter.

3. We have a number of other "variables" which are local to each process. Among these are local copies of $Head$ and $Tail$ and also pointers to new nodes to be enqueued. Again as a practical convention, names of local variables all start with a lower case letter.

4. We can assume two kind of *elementary atomic actions*: Reading and Compare-And-Swap (for short CAS).These two atomic actions are described in Figure 7.

5. Actions involving only local tests and local assignments can be performed concurrently by each process.

Reading	$local\_variable := global\_variable$
Compare-And-Swap	**if** $global\_variable = some\_local\_variable$ **then** $global\_variable := another\_local\_variable$ **end**

Figure 7: The Two Kinds of Elementary Atomic Actions

Notice that we shall extend in what follows the Compare-And-Swap atomic action to possibly contain a local test and a local action as indicated below:

```
**if** $local\_test \;\wedge\; global\_variable = some\_local\_variable$ **then**
   $global\_variable := another\_local\_variable;$
   $local\_action$
**end**
```

And it can also be simplified to the case where the body of the condition is just a local action:

```
**if** $local\_test \;\wedge\; global\_variable = some\_local\_variable$ **then**
   $local\_action$
**end**
```

Such extensions are just convenient ways to handle the atomic actions. They will be used in this paper to ease the reasoning. It is always possible to return to the unique basic Compare_And_Swap atomic action by adding *extra local assignments and tests*.

We shall also sometimes use the construct $CAS(glb, loc1, loc2)$ (where glb is supposed to be a global variable while $loc1$ and $loc2$ are local values) to represent a boolean expression ($glb = loc1$) with a possible side effect ($glb := loc2$). Such a construct can be used either: (1) as a condition within an **if** pseudo-code statement, or (2) as a simple statement. Example of such uses will be shown in Figure 12.

4 Informal Transformations of the Non-concurrent Operations

In this section, we gradually informally transform each operation Dequeue and Enqueue into a sequence of *elementary atomic actions*. Before doing that however, we first merge the operation Adapt within Dequeue and Enqueue respectively. All this will be done (and informally validated) *as if the operations were executed in a purely non-concurrent fashion*. The idea is to obtain at the end of these transformations some operations that will be ready for the formalization and the proof (but probably not yet completely correct). The idea of the formal development is then to gradually "encode" the atomic actions in a mathematical framework (Event-B) allowing us to prove that they indeed refine the initial operations Dequeue and Enqueue.

4.1 Merging Adapt within the Main Operations

In Figure 8, you can see how operation Adapt (which is boxed) is merged with operations Dequeue and Enqueue.

The merging with Dequeue and Enqueue can be further extended by providing a loop as indicated in Figure 9. The reason for introducing these loops is to move towards the concurrent algorithm. In a non-concurrent case, as we are supposed to be here, these loops are at most executed twice.

```
if  $Head \neq Tail$  then
   $Head := Next(Head)$;
   return(true)
elsif  $Next(Tail) \neq$ Null  then
   $Tail := Next(Tail)$;
   $Head := Next(Head)$;
   return(true)
else
   return(false)
end
```

```
$nde :=$ new_node;
$Next(nde) :=$ Null;
if  $Next(Tail) \neq$ Null  then
   $Tail := Next(Tail)$
end ;
$Next(Tail) := nde$
```

Figure 8: Merging Adapt with Dequeue and Enqueue

```
loop
   if  $Head \neq Tail$  then
      $Head := Next(Head)$;
      return(true)
   elsif  $Next(Tail) \neq$ Null  then
      $Tail := Next(Tail)$
   else
      return(false)
   end
end
```

```
$nde :=$ new_node;
$Next(nde) :=$ Null;
loop
   if  $Next(Tail) \neq$ Null  then
      $Tail := Next(Tail)$
   else
      $Next(Tail) := nde$;
      break
   end
end
```

Figure 9: Introducing a loop in Dequeue and in Enqueue

4.2 Introducing Local Process Variables

In Figure 10, local variables (all starting with lower case letters) are introduced within the previous versions of operations Dequeue and Enqueue. On operation Dequeue, you can see that expression $Next(Tail)$ is replaced by nxd which is clearly equal to $Next(Head)$ in a non-concurrent execution. Notice that it is correct to do these replacements because they are made in places where $Head$ and $Tail$ are equal. Likewise, in operation Enqueue you can see that expression $Next(Tail)$ is replaced by nxe, and that an assignment to $Next(Tail)$ is replaced by an assignment to $Next(tle)$. Again, such replacements are certainly correct in a non-concurrent execution.

```
loop
    $hdd := Head$;
    $tld := Tail$;
    $nxd := Next(hdd)$;
    if  $hdd \neq tld$  then
        $Head := nxd$;
        return(true)
    elsif  $nxd \neq$ Null  then
        $Tail := nxd$
    else
        return(false)
    end
end
```

```
$nde :=$ new_node;
$Next(nde) :=$ Null;
loop
    $tle := Tail$;
    $nxe := Next(tle)$;
    if  $nxe \neq$ Null  then
        $Tail := nxe$
    else
        $Next(tle) := nde$;
        break
    end
end
```

Figure 10: Introducing local variables within Dequeue and Enqueue

4.3 Preparing for Concurrency

In Figure 11, we prepare for concurrency in operations Dequeue and Enqueue by adding some extra tests which are certainly useless within a non-concurrent execution but will be important within a concurrent one because they will allow us to define elementary atomic actions. In Figure 12, we show the same pseudo-code with an explicit usage of the CAS atomic operation.

4.4 Final Atomic Actions

In Figures 13 and 14, we show the final (for the moment) situation with the decomposition of each operation in a sequence of elementary atomic actions. As can be seen, they are all simply connected by sequential composition. As a consequence, some local tests are repeated. We have adopted this form because it simplifies the analysis. Each atomic action is either a simple assignment or a possibly degenerated CAS. Each of them is represented in a *named box* followed by some "..." to indicate that some other atomic actions can take place before execution of the same operation can resume. The other atomic actions which can take place are other instantiations of the same or the other operation executed by different processes. Notice that the elementary atomic actions named Deq_Loop and Enq_Loop, which have no contents in these figures, correspond to the deallocation of the local variables created within the loop body. As can be seen, atomic actions are all of the forms presented in section 3. We have so far depicted 13 elementary atomic actions (7 for Dequeue and 6 for Enqueue). If we suppose that 20 different processes are acting concurrently, then we have 13^{20} different situations to consider (processes executing Dequeue can be in 7 distinct waiting situ-

```
loop
  $hdd := Head$;
  $tld := Tail$;
  $nxd := Next(hdd)$;
  if $hdd \neq tld$ then
    if $Head = hdd$ then
      $Head := nxd$;
      return(true)
    end
  elsif $nxd \neq$ Null then
    if $Tail = tld$ then
      $Tail := nxd$
    end
  elsif $Tail = tld$ then
    return(false)
  end
end
```

```
$nde :=$ new_node;
$Next(nde) :=$ Null;
loop
  $tle := Tail$;
  $nxe := Next(tle)$;
  if $nxe \neq$ Null then
    if $Tail = tle$ then
      $Tail := nxe$
    end
  elsif $Next(tle) = nxe$ then
    $Next(tle) := nde$;
    break
  end
end
```

Figure 11: Preparing for concurrency in Dequeue and in Enqueue

ations while processes executing Enqueue can be in 6 waiting situations). Testing is clearly impossible.

5 A Survey of the Event-B Formal Approach

5.1 Development Principles

We have no guarantee, of course, that the previous informal decomposition of each operation into a sequence of atomic actions is "correct". And, as we have just seen at the end of previous section, it is out of the question to perform any test. As a consequence, we think that a formal development (with proofs) is needed. Our approach consists of building several more and more accurate models of the concurrent execution. Each such model corresponds to *what can be observed* by a more and more accurate external observer.

5.2 The initial Model

In the first model, the observer can only see the "last" event (i.e. atomic action) of each operation invocation, that is: events Deq_true or Deq_false for operation Dequeue, and event Enq for operation Enqueue. In other words, the complete execution of each operation has thus been reduced to a single "point". Note that this does not mean that

```
loop
  $hdd := Head$;
  $tld := Tail$;
  $nxd := Next(hdd)$;
  if $hdd \neq tld$ then
    if $CAS1$ then
      return(true)
    end
  elsif $nxd \neq$ Null then
    $CAS(Tail, tld, nxd)$
  elsif $Tail = tld$ then
    return(false)
  end
end
$CAS1$ stands for
$CAS(Head, hdd, nxd)$
```

```
$nde :=$ new_node;
$Next(nde) :=$ Null;
loop
  $tle := Tail$;
  $nxe := Next(tle)$;
  if $nxe \neq$ Null then
    $CAS(Tail, tle, nxe)$
  elsif $CAS2$ then
    break
  end
end
$CAS2$ stands for
$CAS(Next(tle), nxe, nde)$
```

Figure 12: Final versions (so far) of Dequeue and Enqueue with explicit CAS

these operations are performed sequentially: in fact, their hidden events can be interleaved. It only means that the end point of each of them occur in a certain order. In Figure 15, we have represented an execution sample, as "seen" by the observer of the first model. The horizontal line is a time axis. Moreover, A stands for events Adapt_Deq or Adapt_Enq, E stands for event Enq, and D stand for event Deq_true. What is imposed here by this view is that these operations *are serializable*, as would be the case if the queue would have been protected by a global critical section. To put it in another way, the observer closes eyes most of the time: he only opens them when an operation terminates and he thus believes that the operation has been performed "just now". Notice that, at this stage, we also have two identical events Adapt_Deq and Adapt_Enq for operation Adapt.

5.3 Subsequent Models

In subsequent models these timeless events will be gradually "stretched" by means of more events corresponding to the various atomic actions and loops that were developed informally in the previous section. In Figure 16, the observer can see more events making the Dequeue operation. Operation Enqueue is still abstract (performed in one shot). On this figure, d1 to d3 stands for events Deq1 to Deq3. In Figure 17, operation Enqueue is stretched by means of events Enq1, Enq2, Enq3, and Adapt-Enq. Note that among the two executions of operation Enqueue, the one which terminated last, was started first.

```
loop
  $\boxed{hdd := Head}$;                                              Deq1
  ...
  $\boxed{tld := Tail}$;                                              Deq2
  ...
  $\boxed{nxd := Next(hdd)}$;                                         Deq3
  ...
  if  $hdd \neq tld \wedge \underline{Head} = hdd$  then
      $\underline{Head} := nxd$;
      return(true)                                     ;              Deq_true
  end
    ...
  if  $hdd = tld \wedge nxd \neq$ Null $\wedge \underline{Tail} = tld$  then
      $\underline{Tail} := nxd$                                 ;     Adapt_Deq
  end
    ...
  if  $hdd = tld \wedge nxd =$ Null $\wedge Tail = tld$  then
      return(false)                                         ;        Deq_false
  end
    ...
end                                                                   Deq_Loop
```

Figure 13: Showing concurrent atomic actions in Dequeue

5.4 Development Technique

The development technique which we shall use here is called Event-B. It has already been described in various papers such as [ACM03] and [Abr03], so that we shall not repeat it here. Roughly speaking, an Event-B development is made of a sequence of more refined models. Each such model contains a state description with some invariant properties and various events corresponding to transitions. Correctness imply proving invariant preservation and correct refinement of abstract events. In a refined model new events can be introduced: they are supposed to refine implicit events doing nothing. We shall only give in what follows a summary of the different models that have been written and proved. The interested reader can download the entire development from [AC03b]. He can also see and redo the proofs with the *Click'n'Proof* tool [AC03a, AC03b, Cle04].

```
$nde :=$ new_node;
$Next(nde) :=$ Null ;                                          Enq1
  . . .
loop
   $tle := Tail$;                                              Enq2
     . . .
   $nxe := Next(tle)$;                                         Enq3
     . . .
   if  $nxe \neq$ Null $\wedge$ $\underline{Tail} = tle$  then
       $\underline{Tail} := nxe$                          ;    Adapt_Enq
   end
     . . .
   if  $nxe =$ Null $\wedge$ $\underline{Next(tle)} = nxe$  then
       $\underline{Next(tle)} := nde$;
       break                                           ;       Enq
   end
     . . .
end                                                        Enq_Loop
```

Figure 14: Showing concurrent atomic actions in Enqueue

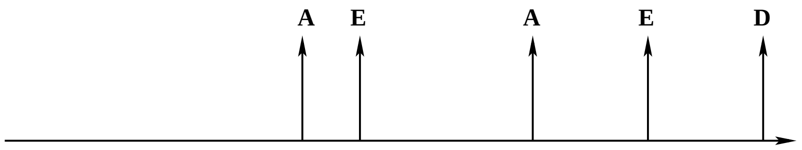

Figure 15: An abstract execution sample: the observer only sees a very few events

6 Summary of Event-B Development

6.1 Model 0: Introducing the Queue and its Basic Operations

In this first model, we introduce the basic global variables of the problem: the function $Next$, the set $Queue$, and the three nodes $Head$, $Tail$ and $Null$. They are all defined in terms of the abstract set N (for nodes). The invariants of Model 0 are shown in Figure 18. On this figure, $Head$ and $Tail$ always point to the $Queue$ (invariant inv0_3 and inv0_4) and they are different from $Null$ (invariant inv0_5). $Head$ points to the first element of the $Queue$ (since it does not belong to the range of $Next$ restricted to $Queue$ according to invariant inv0_8). Moreover $Tail$ points to the node n preceding $Null$ or to the node preceding n (inv0_7). Notice that we say nothing about the structure of the

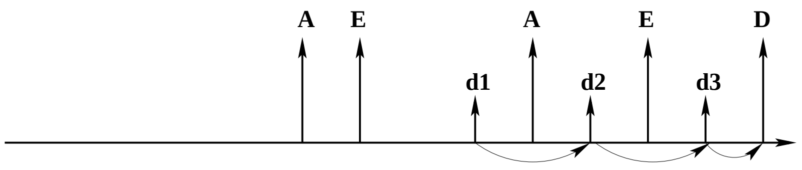

Figure 16: The observer now sees most of the atomic actions of Dequeue

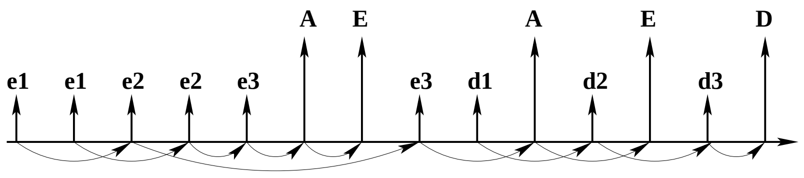

Figure 17: The observer now sees the atomic actions of Enqueue

function $Next$ outside the queue. Within the queue however, we express that $Next$ is a bijection (inv0_8). Finally, we express that $Queue$ is inductive (inv0_9). That is: every property of $Queue$ (represented by a subset s of $Queue$), which is true at $Head$ and also at $Next(x)$ if true at x, is then true for all members of $Queue$. The connectivity of the $Queue$ from $Head$ to $Null$ follows from invariants inv0_8 and inv0_9. As a consequence three of the five properties that were mentioned in [MS96] are certainly fulfilled in Model 0 (the other two follows from the events as will be seen below), namely:

– The linked list is always connected.

– Head always points to the first element in the linked list.

– Tail always points to a node in the linked list.

The four events are straightforward formalizations of the pseudo-code programs defined in Figure 6. They are defined in Figures 19 and 20[2]. It can be seen from the description of events Deq_true and Enq that the two other properties mentioned in [MS96] are indeed fulfilled:

– Nodes are only inserted after the last node in the linked list.

– Nodes are only deleted from the beginning of the linked list.

[2] Our events are defined by a guard (situated between the keywords **when** and **then**) and an action (situated between the keywords **then** and **end**). The guard expresses the *necessary condition* for the event to occur. And the action is either a *multiple* deterministic assignment (in all events shown here except Enq) or a multiple non-deterministic assignment introduced by an **any** construct (in event Enq).

inv0_1 : $Next \in N \rightarrow\!\!\!\!\!\!+\; N$

inv0_2 : $Queue \subseteq N$

inv0_3 : $Head \in Queue$

inv0_4 : $Tail \in Queue$

inv0_5 : $\text{Null} \in Queue \setminus \{Head, Tail\}$

inv0_6 : $Return \in BOOL$

inv0_7 : $Next(Tail) = \text{Null} \;\vee\; Next(Next(Tail)) = \text{Null}$

inv0_8 : $Queue \lhd Next \;\in\; Queue \setminus \{\text{Null}\} \;\twoheadrightarrow\!\!\!\!\!\!\rightarrowtail\; Queue \setminus \{Head\}$

inv0_9 : $\forall s \cdot \begin{pmatrix} s \subseteq Queue \\ Head \in s \\ Next[s] \subseteq s \\ \Rightarrow \\ Queue \subseteq s \end{pmatrix}$

Figure 18: The Invariant of Model 0

The only aspect that remained rather “vague” in Figure 6 was the nature of the new_node chosen by the event Enq. What we say for the moment in event Enq is that our filter for enqueuing a new node is that it does not belong to the $Queue$. The filter is thus the set $N \setminus Queue$ as shown in Figure 20. Mechanically proving that these events all maintain the invariants is easy.

The proof of this model required 50 lemmas of which 7 were proved interactively.

6.2 Model 1: Refining the Dequeue Event

In Model 1, we introduce the abstract set P of processes. We also define the local variables hdd, tld, and nxd as they were used in Figure 10. They are defined here as partial functions from P to N. The constraints on the domains of these functions allows one to take care of the *order* in which these local variables are assigned by their corresponding atomic actions (see Figure 13): first hdd, then tld, and finally nxd. The invariants of this model are presented in Figure 21.

```
Deq_true  $\widehat{=}$
  when
    $Head \neq Tail$
  then
    $Head := Next(Head)$
    $Queue := Queue \setminus \{Head\}$
    $Return :=$ true
  end
```

```
Deq_false  $\widehat{=}$
  when
    $Head = Tail$
  then
    $Return :=$ false
  end
```

Figure 19: The Deq_true and Deq_false Events of Model 0

```
Enq  $\widehat{=}$
when
  $Next(Tail) =$ Null
then
  any  $n$  where
    $n \in N \setminus Queue$
  then
    $Next := Next \lhd\!\!- \{Tail \mapsto n, n \mapsto \text{Null}\}$
    $Queue := Queue \cup \{n\}$
  end
end
```

```
Adapt  $\widehat{=}$
when
  $Next(Tail) \neq$ null
then
  $Tail := Next(Tail)$
end
```

Figure 20: The Enq and Adapt Events of Model 0

The definition of the events Deq1 to Deq3 are straightforward formalizations of the corresponding atomic actions defined in Figure 13. The new versions of the three events Deq_true, Deq_false, Adapt_Deq must refine their abstract counterparts defined in Model 0. In Figures 22 to 24 we show the abstract and concrete versions of each of these events together with the invariants which we have to introduce in order to ensure the correct refinements of the concrete event versions with regard to the more abstract ones. We remind the reader that the guards of a refined event must be *stronger* than that of its more abstract one (this requirement is a consequence of the refinement theory).

As can be seen in Figure 22 (event Deq_true), the invariant we introduce is universally quantified over processes. The antecedent of the quantified implication of this invariant is the same as the guard of the refined event (up to a change of variables, which is introduced in order to ease the reading). The consequent of the implication contains exactly what we need in order to prove that the concrete version of the event refines its

$$hdd \in P \nrightarrow N$$
$$tld \in P \nrightarrow N$$
$$nxd \in P \nrightarrow N$$

$$\mathsf{dom}\,(nxd) \subseteq \mathsf{dom}\,(tld)$$
$$\mathsf{dom}\,(tld) \subseteq \mathsf{dom}\,(hdd)$$

Figure 21: Parts of the Invariant of Model 1

more abstract version, namely (1) the condition $Head \neq Tail$, which is exactly the abstract guard, and (2) the condition $nxd(q) = Next(Head)$, which allows us to replace $Next(Head)$ in the abstract version by $nxd(p)$ in the concrete one. Similar comments

```
Deq_true  ≙
  when
    $Head \neq Tail$
  then
    $Head := Next(Head)$
    $Queue := Queue \setminus \{Head\}$
    $Return :=$ true
  end
```

$$\forall q \cdot \begin{pmatrix} q \in \mathsf{dom}\,(nxd) \\ tld(q) \neq hdd(q) \\ hdd(q) = Head \\ \Rightarrow \\ Head \neq Tail \\ nxd(q) = Next(Head) \end{pmatrix}$$

```
Deq_true  ≙
  any  $p$  where
    $p \in \mathsf{dom}\,(nxd)$
    $tld(p) \neq hdd(p)$
    $hdd(p) = Head$
  then
    $Head := nxd(p)$
    $tld := \{p\} \lhd tld$
    $hdd := \{p\} \lhd hdd$
    $nxd := \{p\} \lhd nxd$
    $Queue := Queue \setminus \{Head\}$
    $Return :=$ true
  end
```

Figure 22: Refinement of Event Deq_true with abstraction and invariant

as the one we have made for event Deq_true can be made on the two other invariants we introduce together with the refinements of events Deq_false and Adapt (see figures 23 and 24). Note that the antecedent of the invariant given in figure 23 is weaker than the guard of event Deq_false (since $\mathsf{dom}\,(nxd)$ is included in $\mathsf{dom}\,(tld)$ according to the invariant given in Figure 21), but it is indeed sufficient to prove the refinement of this event.

All these invariants are introduced by looking at the reason why some refinement proofs fail. As a matter of fact, some specific statements that cannot be proven are candidate new invariants.

```
Deq_false  $\widehat{=}$
  when
    $Head = Tail$
  then
    $Return :=$ false
  end
```

$$\forall q \cdot \begin{pmatrix} q \in \mathsf{dom}\,(tld) \\ tld(q) = hdd(q) \\ tld(q) = Tail \\ \Rightarrow \\ Head = Tail \end{pmatrix}$$

```
Deq_false  $\widehat{=}$
  any  $p$  where
    $p \in \mathsf{dom}\,(nxd)$
    $tld(p) = hdd(p)$
    $tld(p) = Tail$
    $nxd(p) =$ null
  then
    $tld := \{p\} \lhd tld$
    $hdd := \{p\} \lhd hdd$
    $nxd := \{p\} \lhd nxd$
    $Return :=$ false
  end
```

Figure 23: Refinement of Event Deq_false with abstraction and invariant

```
Adapt_Deq  $\widehat{=}$
  when
    $Next(Tail) \neq$ null
  then
    $Tail := Next(Tail)$
  end
```

$$\forall q \cdot \begin{pmatrix} q \in \mathsf{dom}\,(nxd) \\ tld(q) = hdd(q) \\ tld(q) = Tail \\ nxd(q) \neq \text{null} \\ \Rightarrow \\ nxd(q) = Next(Tail) \end{pmatrix}$$

```
Adapt_Deq  $\widehat{=}$
  ANY  $p$  WHERE
    $p \in \mathsf{dom}\,(nxd)$
    $tld(p) = hdd(p)$
    $tld(p) = Tail$
    $nxd(p) \neq$ null
  THEN
    $Tail := nxd(p)$
    $tld := \{p\} \lhd tld$
    $hdd := \{p\} \lhd hdd$
    $nxd := \{p\} \lhd nxd$
  END
```

Figure 24: Refinement of Events Adapt_Deq with abstraction and invariant

After introducing these invariants, our task is not finished: we have to prove them, that is to prove that they are maintained by all our events. Unfortunately, such proofs failed. The reason for this is what is called in [MS96] the ABA problem. This is a problem which arises frequently in concurrent applications where several processes are competing for a shared resource. More precisely, this problem occurs when a node n (pointed to by $Head$) is removed from the $Queue$ by an instance of event Deq_true working for a certain process p. Suppose it is done while another process q points to $Head$ with its local variable hdd. If q still points to n while n is re-enqueued by an instance of event Enq (it is possible because n is not any more in the $Queue$), then the invariant is broken. For example the following invariant was introduced with Deq_true.

$$\forall q \cdot \begin{pmatrix} q \in \mathsf{dom}\,(nxd) \\ tld(q) \neq hdd(q) \\ hdd(q) = Head \\ \Rightarrow \\ Head \neq Tail \end{pmatrix}$$

When trying to prove that Deq_true maintains it, we are led to prove the following (after some simplifications):

$$\begin{array}{l} p \in \mathsf{dom}\,(hdd) \\ q \in \mathsf{dom}\,(hdd) \\ hdd(p) = Head \\ hdd(q) = Next(Head) \\ \Rightarrow \\ Next(Head) \neq Tail \end{array}$$

In Figure 25 we generate a counter-example to the previous condition. In other words, we are going to generate a situation where the following three conditions are simultaneously true:

$$\begin{array}{l} hdd(p) = Head \\ hdd(q) = Next(Head) \\ Next(Head) = Tail \end{array}$$

We start in a situation where two processes, say r and q, have asked to perform a Dequeue, so that their hdd local variables both point to the first element of the queue also pointed to by the global variable $Head$. We suppose that process p wins the contest (Deq_true). As a consequence, the first element of the queue is removed and $Head$ moves forward. But process q still points to the removed node. We suppose then that an Enq event does occur, which choose precisely the just removed element to be enqueued (still process q points to it). Then an event Adapt takes place. Notice that now $Tail$ points to the same node as q does. Finally event Deq1 occurs for process p. Thus process p points to $Head$. Now, the counter-example is there, since we have the following three conditions simultaneously true:

$$\begin{array}{l} hdd(p) = Head \\ hdd(q) = Next(Head) \\ Next(Head) = Tail \end{array}$$

The solution is to add the following invariant stipulating that the range of hdd intersects $Queue$ at most on the $Head$ node:

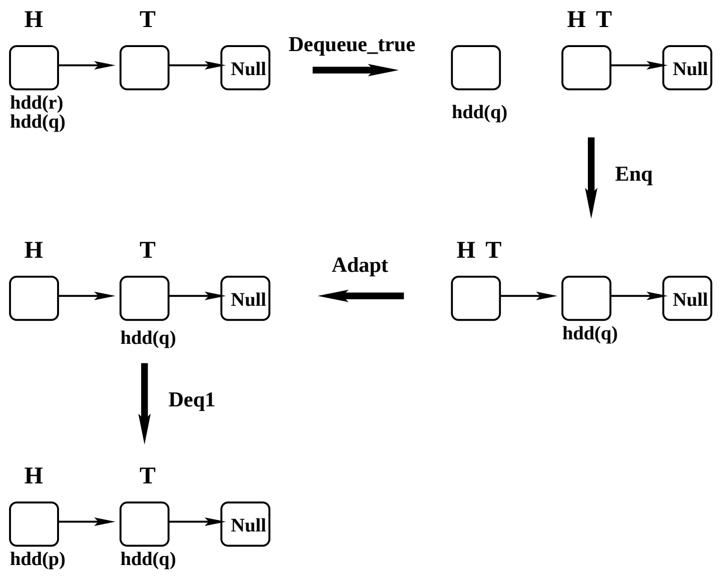

Figure 25: Generating a counter-example

$$\mathsf{ran}\,(hdd) \;\cap\; Queue \;\subseteq\; \{Head\}$$

In other words, local variables hdd must all point to $Head$ when pointing within the $Queue$. Note that the invariant given with Deq_false can now be deduced from this new invariant. Figure 26 shows the remaining invariants.

These invariants can all be proved provided we modify event Enq (Figure 27) by ensuring that the new node, which is chosen for enqueuing is not in the range of hdd. In other words, no process has its local variable pointing to the chosen node. Of course, this solution is not satisfactory (since Enq has to check all hdd local variables) but *at this abstract stage* it is sufficient to allow all the proofs to be done. To summarize, our filter is now $N \setminus (Queue \cup \mathsf{ran}\,(hdd))$.

We touch here a very important aspect of our refinement approach. At some stage, we define some abstract solutions where things are *correct but not practically implementable*. The situation is then improved at a later refinement stage. And the refinement proof will then ensure that the more concrete version will behave as the abstract one but with a practical and implementable solution. Another aspect that is important in our design approach is our usage of the prover. It helps us discovering the invariant we need. This is a consequence of its failure: it often indicates that the invariants are not strong enough.

The proof of this model required 79 lemmas of which 16 were proved interactively.

$$\forall q \cdot \begin{pmatrix} q \in \mathsf{dom}\,(tld) \\ tld(q) \neq hdd(q) \\ hdd(q) = Head \\ \Rightarrow \\ Head \neq Tail \end{pmatrix} \qquad \forall q \cdot \begin{pmatrix} q \in \mathsf{dom}\,(nxd) \\ tld(q) \neq hdd(q) \\ hdd(q) = Head \\ \Rightarrow \\ nxd(q) = Next(Head) \end{pmatrix}$$

$$\mathsf{ran}\,(hdd) \cap Queue \subseteq \{Head\} \qquad \forall q \cdot \begin{pmatrix} q \in \mathsf{dom}\,(nxd) \\ tld(q) = hdd(q) \\ tld(q) = Tail \\ nxd(q) \neq \mathrm{null} \\ \Rightarrow \\ nxd(q) = Next(Tail) \end{pmatrix}$$

Figure 26: Summary of new invariants of Model 1

```
Enq  $\widehat{=}$
  **when**
    $Next(Tail) = \mathrm{Null}$
  **then**
    **any** $n$ **where**
      $n \in N \setminus (Queue \cup \mathsf{ran}\,(hdd))$
    **then**
      $Next := Next \lhd\!\!- \{Tail \mapsto n,\ n \mapsto \mathrm{Null}\}$
      $Queue := Queue \cup \{n\}$
    **end**
  **end**
```

Figure 27: The Refinement of Event Enq

6.3 Model 2: Introducing an Abstract Waste Basket

In this model, we introduce a waste basket, Bsk. This will certainly not constitute the final solution of the ABA problem mentioned in the previous section but will be a little more appropriate than the previous one. And thanks to the proposed invariant it will be a refinement of the previous model. The idea is that a node that is dequeued will be put in the waste basket by event Deq_true and will stay there *for ever* (we shall see in section 6.6 how to implement "for ever"). The invariant for Bsk are defined in Figure 28. One has to refine again event Enq. The new node to be enqueued has to be chosen outside Bsk. Then, according to the invariant, this node will be, a fortiori, outside $\mathsf{ran}\,(hdd)$ (as required by the abstraction). Our filter is thus now $N \setminus (Queue \cup Bsk)$. This filter

$$\boxed{\begin{array}{l} Bsk \;\subseteq\; N \\[2ex] \mathsf{ran}\,(hdd) \;\subseteq\; \{Head\} \;\cup\; Bsk \end{array}} \qquad \boxed{\begin{array}{l} Bsk \;\subseteq\; \mathsf{dom}\,(Next) \\[2ex] Bsk \;\cap\; Queue \;=\; \varnothing \end{array}}$$

Figure 28: Invariant of Model 2

has to replace $N \setminus (Queue \cup \mathsf{ran}\,(hdd))$ in event Enq.

The proof of this model required 22 lemmas of which one was proved interactively.

6.4 Model 3: Refining the Enqueue Event

In this model, the event Enq is refined by introducing the extra events of this operation as shown in Figure 14. Some local variables are defined as shown in Figure 29. Notice the injectivity of the function nde: this is because potential nodes to be enqueued should not be shared by different processes. The new events and the refinement of Enq are straightforward. Some more specific invariants have to be introduced however (these invariants are not presented in the paper). The filtering of new nodes to be enqueued, which was done so far by event Enq is now moved to the new event Enq1 (Figure 30). Notice that the filter is made stronger by excluding nodes in $\mathsf{ran}\,(nde)$ in order to preserve the injectivity of nde. Here is our new filter: $N \setminus (Queue \cup Bsk \cup \mathsf{ran}\,(nde))$ as indicated by event Enq1.

$$\boxed{\begin{array}{l} nde \in P \rightarrowtail\!\!\!\!\!\!\!\!\;+\; N \\[2ex] tle \in P \leftrightarrow\!\!\!\!\!+\; N \\[2ex] nxe \in P \leftrightarrow\!\!\!\!\!+\; N \end{array}} \qquad \boxed{\begin{array}{l} \mathsf{dom}\,(nxe) \;\subseteq\; \mathsf{dom}\,(tle) \\[2ex] \mathsf{dom}\,(tle) \;\subseteq\; \mathsf{dom}\,(nde) \\[2ex] \mathsf{ran}\,(nde) \;\cap\; (Queue \;\cup\; Bsk) \;=\; \varnothing \\[2ex] \mathsf{ran}\,(tle) \;\subseteq\; Queue \;\cup\; Bsk \end{array}}$$

Figure 29: Invariant of Model 3

The proof of this model required 83 lemmas of which 12 were proved interactively.

```
Enq1 $\widehat{=}$
  **any** $p, n$ **where**
    $p \in P$
    $p \notin \mathsf{dom}\,(nde)$
    $n \in N \setminus (Queue \;\cup\; Bsk \;\cup\; \mathsf{ran}\,(nde))$
  **then**
    $nde(p) := n$
    $Next(n) := \mathrm{Null}$
  **end**
```

Figure 30: Event Enq1 of Model 3

6.5 Model 4: Introducing a Free List

In this model, we introduce a free list $Free$ together with the properties described in the invariant shown in Figure 31. This allows us to strengthen and simplify the filtering

$$Free \;\subseteq\; N \qquad\qquad Free \;\cap\; Bsk = \varnothing$$

$$Free \;\cap\; Queue = \varnothing \qquad\qquad Free \;\cap\; \mathsf{ran}\,(nde) = \varnothing$$

Figure 31: Invariant of Model 4

mentioned in previous section by having Enq1 taking now the candidate node to be enqueued simply within $Free$. Notice that we still have the basket Bsk within which we throw away a node which is deleted from the $Queue$ by event Deq_true (Figure 32). In order to prepare a future implementation of Bsk in the next refinement step, we now suppose that event Deq_true also systematically puts a new node in the $Free$ list. This node is chosen in the set $N \setminus (Queue \cup Bsk \cup \mathsf{ran}\,(nde) \cup Free)$ in order to maintain the previous invariants. This action of Deq_true seems to be magic at this stage: we throw away (for ever) the removed node in Bsk and we simultaneously add a new node in $Free$. But, again, we are still in an abstraction.

The proof of this model required 31 lemmas of which one was done interactively.

6.6 Model 5: Introducing Concrete Nodes

We are now ready to implement the basket Bsk. In fact, it is not really an *implementation* of Bsk as it will completely disappear at this stage. Notice that it is absolutely necessary to have Bsk disappearing as it grows indefinitely. So, what we are giving

```
Deq_true  $\widehat{=}$
  any  $p, n$  where
    $p \in P$
    $tld(p) \neq hdd(p)$
    $hdd(p) = Head$
    $n \in N \setminus (Queue \cup Bsk \cup Free \cup \mathsf{ran}\,(nde))$
  then
    $Head := nxd(p)$
    $tld := \{p\} \lhd tld$
    $hdd := \{p\} \lhd hdd$
    $nxd := \{p\} \lhd nxd$
    $Queue := Queue \setminus \{Head\}$
    $Bsk := Bsk \cup \{Head\}$
    $Free := Free \cup \{n\}$
    $Return :=$ true
  end
```

Figure 32: Refinement of Event Deq_true in Model 4

here is a means of *calculating* Bsk rather than storing it. We shall also see how we can generate a new node in $Free$ while deleting a node from the $Queue$. The idea of this refinement is to introduce a set C of concrete nodes. We also define two total functions from N to C and from N to $\mathbb{N}$. The direct product of these two functions is injective: given a member c of C and a natural number we obtain at most one member n of N.

$$cnode \in N \rightarrow C$$

$$count \in N \rightarrow \mathbb{N}$$

$$cnode \otimes count \in N \rightarrowtail C \times \mathbb{N}$$

Figure 33: Some Invariant of Model 5

Notice that we might have a finite number of concrete nodes C, whereas the set N of abstract nodes is infinite. The next invariant shown in Figure 34 is fundamental: it defines Bsk for nodes with the same $cnode$ as nodes which are members of the set $Queue \cup Free \cup \mathsf{ran}\,(nde)$: nodes in Bsk are exactly those nodes with smaller $count$. The node n to be chosen by Deq_true (Figure 35) to be put in $Free$ is then one with the same $cnode$ as $Head$ but with a $count$ which is just $count(Head) + 1$.

According to the abstraction this node n must *not* be a member of the set $Queue \cup Bsk \cup Free \cup \mathsf{ran}\,(nde)$. Suppose it is a member of $Queue \cup Free \cup \mathsf{ran}\,(nde)$,

$$\forall\,(m,n)\cdot\begin{pmatrix} m \in N \\ n \in Queue \cup Free \cup \mathsf{ran}\,(nde) \\ cnode(m) = cnode(n) \\ \Rightarrow \\ m \in Bsk \Leftrightarrow count(m) < count(n) \end{pmatrix}$$

Figure 34: The fundamental invariant of Model 5

```
Deq_true  $\widehat{=}$
  any  $p, n$  where
    $p \in P$
    $tld(p) \neq hdd(p)$
    $hdd(p) = Head$
    $n \in N$
    $cnode(n) = cnode(Head)$
    $count(n) = count(Head) + 1$
  then
    $Head := nxd(p)$
    $tld := \{p\} \lhd\!\!\!- \; tld$
    $hdd := \{p\} \lhd\!\!\!- \; hdd$
    $nxd := \{p\} \lhd\!\!\!- \; nxd$
    $Queue := Queue \setminus \{Head\}$
    $Free := Free \cup \{n\}$
    $Return :=$ true
  end
```

Figure 35: Refinement of Event Deq_true in Model 5

then, according to the invariant, $Head$ (which has a smaller count that n) must be in Bsk, which is impossible. Therefore n is not a member $Queue \cup Free \cup \mathsf{ran}\,(nde)$. Now suppose that n is a member of Bsk, then since $Head$ is clearly a member of $Queue \cup Free \cup \mathsf{ran}\,(nde)$ then, according to the invariant, $count(n)$ must be strictly smaller than $count(Head)$, which is not the case. As a consequence, n is not a member of the set $Queue \cup Bsk \cup Free \cup \mathsf{ran}\,(nde)$: it is indeed consistent with the abstraction.

We must now prove that old $Head$ will "automatically" go in Bsk when n has been put into $Free$. This will certainly be the case because n is now in $Queue \cup Free \cup \mathsf{ran}\,(nde)$ and has a count that is larger than that of $Head$. As can be seen, the variable Bsk, which was a useful abstraction, has now *completely disappeared*.

The proof of this model required 35 lemmas of which 11 were done interactively.

6.7 Model 6: Removing Abstract Nodes

This last model is just a technical refinement where we now make the abstract set N completely disappearing. We only have concrete nodes, and each such node has a unique counter. In this refinement all variables are changed and become “concrete”. We now eventually re-translate the final obtained sequential atomic actions into pseudo-code programs (Figure 36). As can be seen in this figure, it may seem that $Ccount$ is only assigned but never used. In fact, it is not the case as the comparisons are now made in the various instances of CAS between concrete nodes together with their counters.

The proof of this model required 29 lemmas of which 16 were proved interactively.

7 Other Works

E. Yahav et M. Sagiv [YS03] have proved these algorithms by using a model checker based on Abstract Interpretation and a three-valued logic. The algorithms are encoded in Java together with the five properties that the queue must maintain. The connectivity property of the list is defined by saying the “Tail is reachable from Head”. It is not clear whether this property really states that all nodes in the queue can be reached from the Head. Such properties have been checked using the 3VMC Model Checker which was developed by E. Yahav. The initial configuration allows to specify the number of en-queuing processes and that of dequeuing processes. One of these two numbers has to be bounded whereas it is not necessary for the other. The advantage of their approach is that the verification is completely automatic on the the Java representation of the algorithms.

Another approach of the same algorithms has been undertaken by S. Doherty et al [DGLM04]. They used abstraction and proofs. They model the algorithms using two automata (within the framework of I/O Automata) with a state for each line of the original Michael and Scott pseudo-code programs. Then they construct an abstraction of these automata in order to realize the main steps of the algorithms: one enqueue with adapt and one dequeue. The queue is modeled in the form of a sliding window working with an infinite sequence. As a consequence, the ABA problem does not exist and the five basic properties of the queues are maintained. In order to prove the correctness of the concrete automata with respect of their respective abstractions, they used both techniques of backward and forward simulations. This generated 1900 lemmas, which were proved using PVS.

8 Conclusion

In this paper, we have presented a complete reconstruction of Michael and Scott concurrent queue algorithms as defined in [MS96]. Our final result slightly differs from theirs.

```
loop
    $chdd := CHead$;
    $ctld := CTail$;
    $cnxd := CNext(chdd)$;
    if  $chdd \neq ctld$  then
        if  $CAS(CHead, chdd, cnxd)$  then
            $Ccount(chdd) := Ccount(chdd) + 1$;
            $Cfree := Cfree \cup \{chdd\}$;
            return(true)
        end
    elsif  $cnxd \neq$ CNull  then
        $CAS(CTail, ctld, cnxd)$
    elsif  $CTail = ctld$  then
        return(false)
    end
end
```

```
$cnde :\in CFree$;
$CNext(cnde) :=$ CNull;
loop
    $ctle := CTail$;
    $cnxe := Next(ctle)$;
    if  $cnxe \neq$ CNull  then
        $CAS(CTail, ctle, cnxe)$
    elsif  $CAS(CNext(ctle), cnxe, cnde)$  then
        break
    end
end
```

Figure 36: Final versions of Dequeue and Enqueue using CAS

The difference is first on the Dequeue operation which could not be refined to exactly obtain their algorithm. This is because we impose an abstract serialization of the three operations Dequeue, Enqueue and Adapt corresponding to the *end of these operation executions*. In our version, the Dequeue operation returns false when indeed the queue is empty, whereas in Michael and Scott it is possible that the operation returns false while the queue is not empty anymore (although it was empty when the operation Dequeue was called). Note that the serialization of the end of the three operations is important in the proof process: it is this very property that is proved to be maintained across the various refinements done during the design. The second difference is that we link the counters to the nodes, not to the pointers to the nodes as do Michael and Scott. It results in only one incrementation of the counter, namely when a node is returned to the free list.

The overall development required to prove 329 lemmas of which 64 were proved interactively yielding 80% of automatic proofs.

Acknowledgments
We would like to thank R. Bornat for introducing us to this example. We had many discussions with him on this subject and more generally on Separation Logic. We also want to thank L. Voisin for his careful readings of several versions of this paper.

References

[Abr96] J.R. Abrial. *The B Book - Assigning Programs to Meanings*. Cambridge University Press, 1996. ISBN 0-521-49619-5.

[Abr03] J.R. Abrial. Event Based Sequential Program Development: Application to Constructing a Pointer Program. In Dino Mandrioli Keijiro Araki, Stefania Gnesi, editor, *FME 2003: Formal Methods*, volume 2805 of *Lecture Notes in Computer Science*, pages 51–74, Pisa, Sept 2003.

[AC03a] J.-R. Abrial and D. Cansell. Click'n'prove : Interactive proofs within set theory. In David Basin et Burkhart Wolff, editor, *16th International Conference on Theorem Proving in Higher Order Logics - TPHOLs'2003, Rome, Italy*, volume 2758 of *Lecture notes in Computer Science*, pages 1–24. Springer, Sep 2003.

[AC03b] J.-R. Abrial and D. Cansell. Click'n'prove ("la balbulette"), Sep 2003. http://www.loria.fr/∼cansell/cnp.htm.

[ACM03] J.-R. Abrial, D. Cansell, and D. Méry. A mechanically proved and incremental development of ieee 1394 tree identify protocol. *Formal Aspects of Computing*, 14(3):215–227, Apr 2003.

[Cle04] ClearSy. B4free. Feb 2004. http://www.b4free.com.

[DGLM04] S. Doherty, L. Groves, V. Luchangco, and M. Moir. Formal verification of a practical lock-free queue algorithm. In David de Frutos-Escrig and Manuel Núñez, editors, *Formal Techniques for Networked and Distributed Systems - FORTE 2004, 24th IFIP WG 6.1 International Conference, Madrid Spain, September 27-30, 2004, Proceedings*, volume 3235 of *Lecture Notes in Computer Science*, pages 97–114. Springer, 2004.

[MS96] M. M. Michael and M. L. Scott. Simple, Fast, and Practical Non-Blocking and Blocking Concurrent Queue Algorithms. In *Proceedings of the 15th Symposium on Principles of Distributed Computing*, pages 267–75, Philadelphia, Pennsylvania, May 1996.

[YS03] E. Yahav and M. Sagiv. Automatically verifying concurrent queue algorithms. In Byron Cook, Scott Stoller, and Willem Visser, editors, *Electronic Notes in Theoretical Computer Science*, volume 89. Elsevier, 2003.