Efficient almost wait-free parallel accesible dynamic hashtables

Citation for published version (APA):
Gao, H., Groote, J. F., & Hesselink, W. H. (2003). *Efficient almost wait-free parallel accesible dynamic hashtables*. (Computer science reports; Vol. 0303). Technische Universiteit Eindhoven.

Document status and date:
Published: 01/01/2003

Document Version:
Publisher's PDF, also known as Version of Record (includes final page, issue and volume numbers)

Please check the document version of this publication:

• A submitted manuscript is the version of the article upon submission and before peer-review. There can be important differences between the submitted version and the official published version of record. People interested in the research are advised to contact the author for the final version of the publication, or visit the DOI to the publisher's website.
• The final author version and the galley proof are versions of the publication after peer review.
• The final published version features the final layout of the paper including the volume, issue and page numbers.

Link to publication

General rights
Copyright and moral rights for the publications made accessible in the public portal are retained by the authors and/or other copyright owners and it is a condition of accessing publications that users recognise and abide by the legal requirements associated with these rights.

• Users may download and print one copy of any publication from the public portal for the purpose of private study or research.
• You may not further distribute the material or use it for any profit-making activity or commercial gain
• You may freely distribute the URL identifying the publication in the public portal.

If the publication is distributed under the terms of Article 25fa of the Dutch Copyright Act, indicated by the "Taverne" license above, please follow below link for the End User Agreement:

www.tue.nl/taverne

Take down policy
If you believe that this document breaches copyright please contact us at:

openaccess@tue.nl

providing details and we will investigate your claim.

Download date: 04. Oct. 2023

Efficient Almost Wait-free Parallel Accessible Dynamic Hashtables

Gao, H.[1], Groote, J.F.[2], Hesselink, W.H.[1]
[1] Department of Mathematics and Computing Science, University of Groningen, P.O. Box 800, 9700 AV Groningen, The Netherlands(Email: `{hui,wim}@cs.rug.nl`)
[2] Department of Mathematics and Computing Science, Eindhoven University of Technology, P.O. Box 513, 5600 MB Eindhoven, The Netherlands and CWI, P.O. Box 94079, 1090 GB Amsterdam, The Netherlands (Email: `jfg@win.tue.nl`)

Abstract

In multiprogrammed systems, synchronization often turns out to be a performance bottleneck and the source of poor fault-tolerance. Wait-free and lock-free algorithms can do without locking mechanisms, and therefore do not suffer from these problems. We present an efficient almost wait-free algorithm for parallel accessible hashtables, which promises more robust performance and reliability than conventional lock-based implementations. Our solution is as efficient as sequential hashtables. It can easily be implemented using C-like languages and requires on average only constant time for insertion, deletion or accessing of elements. Apart from that, our new algorithm allows the hashtables to grow and shrink dynamically when needed.

A true problem of lock-free algorithms is that they are hard to design correctly, even when apparently straightforward. Ensuring the correctness of the design at the earliest possible stage is a major challenge in any responsible system development. Our algorithm contains 81 atomic statements. In view of the complexity of the algorithm and its correctness properties, we turned to the interactive theorem prover PVS for mechanical support. We employ standard deductive verification techniques to prove around 200 invariance properties of our almost wait-free algorithm, and describe how this is achieved using the theorem prover PVS.

CR Subject Classification (1991): D.1 Programming techniques
AMS Subject Classification (1991): 68Q22 Distributed algorithms, 68P20 Information storage and retrieval
Keywords & Phrases: Hashtables, Distributed algorithms, Lock-free, Wait-free

1 Introduction

We are interested in efficient, reliable, parallel algorithms. The classical synchronization paradigms are not most suited for this, because synchronization often turns out a performance bottleneck, and failure of a single process can force all other processes to come to a halt. Therefore, wait-free, lock-free, or synchronization-free algorithms are of interest [11, 19, 13].

An algorithm is *wait-free* when each process can accomplish its task in a finite number of steps, independently of the activity and speed of other processes. An algorithm is *lock-free* when it guarantees that within a finite number of steps always some process will complete its tasks, even if other processes halt. An algorithm is *synchronization-free* when it does not contain synchronization primitives. The difference between wait-free and lock-free is that a lock-free process can be arbitrarily delayed by other processes that repeatedly start and accomplish tasks. The difference between synchronization-free and lock-free is that in a synchronization-free algorithm processes may delay each other arbitrarily, without getting closer to accomplishing their respective tasks. As we present a lock-free algorithm, we only speak about lock-freedom below, but most applies to wait-freedom or synchronization-freedom as well.

Since the processes in a lock-free algorithm run rather independently of each other, lock-free algorithms scale up well when there are more processes. Processors can finish their tasks on their own, without being blocked, and generally even without being delayed by other processes. So, there is no need to wait for slow or overloaded processors. In fact, when there are processors of differing speeds, or under different loads, a lock-free algorithm will generally distribute common tasks over all processors, such that it is finished as quickly as possible.

As argued in [13], another strong argument for lock-free algorithms is reliability. A lock-free algorithm will carry out its task even when all but one processor stops working. Without problem it can stand any pattern of processors being switched off and on again. The only noticeable effect of failing processors is that common tasks will be carried out somewhat slower, and the failing processor may have claimed resources, such as memory, that it can not relinquish anymore.

For many algorithms the penalty to be paid is minor; setting some extra control variables, or using a few extra pointer indirections suffices. Sometimes, however, the time and space complexities of a lock-free algorithm is substantially higher than its sequential, or 'synchronized' counterpart [7]. Furthermore, some machine architectures are not very capable of handling shared variables, and do not offer *compare-and-swap* or *test-and-set* instructions necessary to implement lock-free algorithms.

Hashtables are very commonly in use to efficiently store huge but sparsely filled tables. As far as we know, no wait- or lock-free algorithm for hashtables has ever been proposed. There are very general solutions for wait-free addresses in general [1, 2, 6, 9, 10], but these are not efficient. Furthermore, there exist wait-free algorithms for different domains, such as linked lists [19], queues [20] and memory management [8, 11]. In this paper we present an almost wait-free algorithm for hashtables. Strictly speaking, the algorithm is only lock-free, but wait-freedom is only violated when a hashtable is resized, which is a relatively rare operation. We allow fully parallel *insertion*, *deletion* and *finding* of elements. As a correctness notion, we take that the operations behave the same as for 'ordinary' hashtables, under some arbitrary serialization of these operations. So, if a *find* is carried out strictly after an *insert*, the inserted element is found. If *insert* and *find* are carried out at the same time, it may be that *find* takes place before *insertion*, and it is not determined whether an element will be returned.

An important feature of our hashtable is that it can dynamically grow and shrink when needed. This requires subtle provisions, which can be best understood by considering the following scenarios. Suppose that process A is about to (slowly) insert an element in a hashtable H_1. Before this happens, however, a fast process B has resized the hashtable by making a new hashtable H_2, and has copied the content from H_1 to H_2. If (and only if) process B did not copy the insertion of A, A must be informed to move to the new hashtable, and carry out the insertion there. Suppose a process C comes into play also copying the content from H_1 to H_2. This must be possible, since otherwise B can stop copying, blocking all operations of other processes on the hashtable, and thus violating the lock-free nature of the algorithm. Now the value inserted by A can but need not be copied by both B and/or C. This can be made more complex by a process D that attempts to replace H_2 by H_3. Still, the value inserted by A should show up exactly once in the hashtable, and it is clear that processes should carefully keep each other informed about their activities on the tables.

A true problem of lock-free algorithms is that they are hard to design correctly, which even holds for apparently straightforward algorithms. Whereas human imagination generally suffices to deal with all possibilities of sequential processes or synchronized parallel processes, this appears impossible (at least to us) for lock-free algorithms. The only technique that we see fit for any but the simplest lock-free algorithms is to prove the correctness of the algorithm very precisely, and to double check this using a proof checker or theorem prover.

Our algorithm contains 81 atomic statements. The structure of our algorithm and its correctness properties, as well as the complexity of reasoning about them, makes neither automatic nor manual verification feasible. We have therefore chosen the higher-order interactive theorem prover PVS [3, 18] for mechanical support. PVS has a convenient specification language and contains a proof checker which allows users to construct proofs interactively, to automatically execute trivial proofs, and to check these proofs mechanically.

Our solution is as efficient as sequential hashtables. It requires on average only constant time for insertion, deletion or accessing of elements.

Overview of the paper

Section 2 contains the description of the hashtable interface offered to the users. The algorithm is presented in Section 3. Section 4 contains a description of the proof of the safety properties of the algorithm: functional correctness, atomicity, and absence of memory loss. This proof is based on a list of around 200 invariants, presented in Appendix A, while the relationships between the invariants are given by a dependency graph in Appendix B. Progress of the algorithm is proved informally in Section 5. Conclusions are drawn in Section 6.

2 The interface

The aim is to construct a hashtable that can be accessed simultaneously by different processes in such a way that no process can passively block another process' access to the table.

A hashtable is an implementation of (partial) functions between two domains, here called *Address* and *Value*. The hashtable thus implements a modifiable shared variable $\mathtt{X} \in \mathit{Address} \to \mathit{Value}$. The domains *Address* and *Value* both contain special default elements $0 \in \mathit{Address}$ and $\mathbf{null} \in \mathit{Value}$. An equality $\mathtt{X}(a) = \mathbf{null}$ means that no value is currently associated with the address a. In particular, since we never store a value for the address 0, we impose the invariant

$$\mathtt{X}(0) = \mathbf{null} \ .$$

We use open addressing to keep all elements within the table. For the implementation of the hashtables we require that from every value the address it corresponds to is derivable. We therefore assume that some function $\mathit{ADR} \in \mathit{Value} \to \mathit{Address}$ is given with the property that

Ax1: $\qquad v = \mathbf{null} \ \equiv \ \mathit{ADR}(v) = 0$

Indeed, we need **null** as the value corresponding to the undefined addresses and use address 0 as the (only) address associated with the value **null**. We thus require the hashtable to satisfy the invariant

$$\mathtt{X}(a) \neq \mathbf{null} \quad \Rightarrow \quad \mathit{ADR}(\mathtt{X}(a)) = a \ .$$

Note that the existence of *ADR* is not a real restriction since one can choose to store the pair (a, v) instead of v. When a can be derived from v, it is preferable to store v, since that saves memory.

There are four principle operations: *find*, *delete*, *insert* and *assign*. The first one is to *find* the value currently associated with a given address. This operation yields **null** if the address has no associated value. The second operation is to *delete* the value currently associated with a given address. It fails if the address was empty, i.e. $\mathtt{X}(a) = \mathbf{null}$. The third operation is to *insert* a new value for a given address, provided the address was empty. So, note that at least one out of two consecutive *inserts* for address a must fail, except when there is a *delete* for address a in between them. The operation *assign* does the same as *insert*, except that it rewrites the value even if the associated address is not empty. Moreover, *assign* never fails.

We assume that there is a bounded number of processes that may need to interact with the hashtable. Each process is characterized by the sequence of operations

$$(\ \mathit{getAccess}\ ;\ (\mathit{find} + \mathit{delete} + \mathit{insert} + \mathit{assign})^*\ ;\ \mathit{releaseAccess})^\omega$$

A process that needs to access the table, first calls the procedure *getAccess* to get the current hashtable pointer. It may then invoke the procedures *find*, *delete*, *insert*, and *assign* repeatedly, in an arbitrary, serial manner. A process that has access to the table can call *releaseAccess* to log out. The processes may call these procedures concurrently. The only restriction is that every process can do at most one invocation at a time.

The basic correctness conditions for concurrent systems are functional correctness and atomicity, say in the sense of [16], Chapter 13. Functional correctness is expressed by prescribing how the procedures *find*, *insert*, *delete*, *assign* affect the value of the abstract mapping $\mathtt{X}$. Atomicity is expressed by the condition that the modification of $\mathtt{X}$ is executed atomically at some time between the invocation of the routine and its response. Each of these procedures has the precondition that the calling process has access to the table. In this specification, we use auxiliary private variables declared locally in the usual way. We give them the suffix S to indicate that the routines below are the specifications of the procedures. We use angular brackets $\langle$ and $\rangle$ to indicate atomic execution of the enclosed command.

$$
\begin{array}{ll}
 & \textbf{proc } \mathit{find}_S(a : \mathit{Address} \setminus \{0\}) : \mathit{Value} = \\
 & \quad \textbf{local } rS : \mathit{Value}; \\
\text{(fS)} & \quad \langle\ rS := \ \mathtt{X}(a)\ \rangle; \\
 & \textbf{return } rS.
\end{array}
$$

$$
\begin{array}{ll}
 & \textbf{proc } \mathit{delete}_S(a : \mathit{Address} \setminus \{0\}) : \mathit{Bool} = \\
 & \quad \textbf{local } \mathit{sucS} : \mathit{Bool}; \\
\text{(dS)} & \quad \langle\ \mathit{sucS} := (\mathtt{X}[a] \neq \textbf{null})\ ; \\
 & \quad\ \ \textbf{if } \ \mathit{sucS} \ \textbf{ then } \ \mathtt{X}[a] := \textbf{null} \ \textbf{ end}\ \rangle\ ; \\
 & \textbf{return } \mathit{sucS}.
\end{array}
$$

$$
\begin{array}{ll}
 & \textbf{proc } \mathit{insert}_S(v : \mathit{Value} \setminus \{\textbf{null}\}) : \mathit{Bool} = \\
 & \quad \textbf{local } \mathit{sucS} : \mathit{Bool}\ ;\ a : \mathit{Address} := \mathit{ADR}(v)\ ; \\
\text{(iS)} & \quad \langle\ \mathit{sucS} := (\mathtt{X}[a] = \textbf{null})\ ; \\
 & \quad\ \ \textbf{if } \ \mathit{sucS} \ \textbf{ then } \ \mathtt{X}[a] := v \ \textbf{ end}\ \rangle\ ; \\
 & \textbf{return } \mathit{sucS}.
\end{array}
$$

$$
\begin{array}{ll}
 & \textbf{proc } \mathit{assign}_S(v : \mathit{Value} \setminus \{\textbf{null}\}) = \\
 & \quad \textbf{local } a : \mathit{Address} := \mathit{ADR}(v)\ ; \\
\text{(aS)} & \quad \langle\ \mathtt{X}[a] := v\ \rangle\ ; \\
 & \textbf{end}.
\end{array}
$$

Note that, in all cases, we require that the body of the procedure is executed atomically at some moment between the beginning and the end of the call, but that this moment need not coincide with the beginning or end of the call. This is the reason that we do not (e.g.) specify *find* by the single line **return** $\mathtt{X}(a)$.

Due to the parallel nature of our system we cannot use pre and postconditions to specify it. For example, it may happen that *insert*(v) returns *true* while $\mathtt{X}(\mathit{ADR}(v)) = \textbf{null}$ since another process deletes $\mathit{ADR}(v)$ between the execution of (iS) and the response of *insert*.

We prove partial correctness by extending the implementation with the auxiliary variables and commands used in the specification. So, we regard $\mathtt{X}$ as a shared auxiliary variable and rS and *sucS* as private auxiliary variables; we augment the implementations of *find*, *delete*, *insert*, *assign* with the atomic commands (fS), (dS), (iS), (aS), respectively. We prove that the implementation of the procedure below executes its atomic specification command always precisely once and that the resulting value r or *suc* of the implementation equals the resulting value rS or *sucS* in the specification above. It follows that, by removing the implementation variables from the combined program, we obtain the specification. This removal may eliminate many atomic steps of the implementation. This is known as removal of stutterings in TLA [14] or abstraction from τ steps in process algebras.

3 The algorithm

An implementation consists of P processes along with a set of variables, for $P \geq 1$. Each process, numbered from 1 up to P, is a sequential program comprised of atomic statements. Actions on private variables can be added to an atomic statement, but all actions on shared variables must be separated into atomic accesses. Since auxiliary variables are only used to facilitate the proof of correctness, they can be assumed to be touched instantaneously without violation of the atomicity restriction.

3.1 Hashing

We implement function `X` via hashing with open addressing, cf. [15, 21]. We do not use direct chaining, where colliding entries are stored in a secondary list, because maintaining these lists in a lock-free manner is tedious [19], and expensive when done wait-free. A disadvantage of open addressing with deletion of elements is that the contents of the hashtable must regularly be refreshed by copying the non-deleted elements to a new hashtable. As we wanted to be able to resize the hashtables anyhow, we consider this less of a burden.

In principle, hashing is a way to store address-value pairs in an array (hashtable) with a length much smaller than the number of potential addresses. The indices of the array are determined by a hash function. In case the hash function maps two addresses to the same index in the array there must be some method to determine an alternative index. The question how to choose a good hash function and how to find alternative locations in the case of open addressing is treated extensively elsewhere, e.g. [15].

For our purposes it is convenient to combine these two roles in one abstract function *key* given by:

$$\mathit{key}(a : \mathit{Address},\ l : \mathit{Nat},\ n : \mathit{Nat}) : \mathit{Nat}\ ,$$

where l is the length of the array (hashtable), that satisfies

Ax2: $\qquad 0 \leq \mathit{key}(a, l, n) < l$

for all a, l, and n. The number n serves to obtain alternative locations in case of collisions: when there is a collision, we re-hash until an empty "slot" (i.e. **null**) or the same address in the table is found. The approach with a third argument n is unusual but very general. It is more usual to have a function *Key* dependent on a and l, and use a second function *Inc*, which may depend on a and l, to use in case of collisions. Then our function *key* is obtained recursively by

$$\mathit{key}(a, l, 0) = \mathit{Key}(a, l) \text{ and } \mathit{key}(a, l, n + 1) = \mathit{Inc}(a, l, \mathit{key}(a, l, n))\ .$$

We require that, for any address a and any number l, the first l keys are all different, as expressed in

Ax3: $\qquad 0 \leq k < m < l \quad \Rightarrow \quad \mathit{key}(a, l, k) \neq \mathit{key}(a, l, m)\ .$

3.2 Tagging of values

In hashtables with open addressing a deleted value cannot be replaced by **null** since **null** signals the end of the search. Therefore, such a replacement would invalidate searches for other values. Instead, we introduce an additional "value" **del** to replace deleted values.

Since we want the values in the hashtable to migrate to a bigger table when the first table becomes full, we need to tag values that are being migrated. We cannot simply remove such a value from the old table, since the migrating process may stop functioning during the migration. Therefore, a value being copied must be tagged in such a way that it is still recognizable. This is done by the function *old*. We thus introduce an extended domain of values to be called *EValue*, which is defined as follows:

$$\mathit{EValue} = \{\mathbf{del}\} \cup \mathit{Value} \cup \{\mathit{old}(v) \mid v \in \mathit{Value}\}$$

We furthermore assume the existence of functions $\mathit{val} : \mathit{EValue} \to \mathit{Value}$ and $\mathit{oldp} : \mathit{EValue} \to \mathit{Bool}$ that satisfy, for all $v \in \mathit{Value}$:

$$\begin{aligned}
&\mathit{val}(v) = v\\
&\mathit{val}(\mathbf{del}) = \mathbf{null}\\
&\mathit{val}(\mathit{old}(v)) = v\\
&\mathit{oldp}(v) = \mathit{false}\\
&\mathit{oldp}(\mathbf{del}) = \mathit{false}\\
&\mathit{oldp}(\mathit{old}(v)) = \mathit{true}
\end{aligned}$$

Note that the *old* tag can easily be implemented by designating one special bit in the representation of *Value*. In the sequel we write **done** for *old*(**null**). Moreover, we extend the function *ADR* to domain *EValue* by $ADR(v) = ADR(val(v))$.

3.3 Data structure

A *Hashtable* is either $\perp$, indicating the absence of a hashtable, or it has the following structure:

> `size` : *Nat*;
> `occ` : *Nat*;
> `dels` : *Nat*;
> `bound` : *Nat*;
> `table` : **array** 0 . . `size-1` **of** *EValue*.

The field `size` indicates the size of the hashtable, `bound` the maximal number of places that can be occupied before refreshing the table. Both are set when creating the table and remain constant. The variable `occ` gives the number of occupied positions in the table, while the variable `dels` gives the number of deleted positions. If h is a pointer to a hashtable, we write h.`size`, h.`occ`, h.`dels` and h.`bound` to access these fields of the hashtable. We write h.`table`[i] to access the i^{th} *EValue* in the table.

Apart from the *current* hashtable, which is the main representative of the variable `X`, we have to deal with *old* hashtables, which were in use before the current one, and *new* hashtables, which can be created after the current one.

We now introduce data structures that are used by the processes to find and operate on the hashtable and allow to delete hashtables that are not used anymore. The basic idea is to count the number of processes that are using a hashtable, by means of a counter `busy`. The hashtable can be thrown away when `busy` is set to 0. An important observation is that `busy` cannot be stored as part of the hashtable, in the same way as the variables `size`, `occ` and `bound` above. The reason for this is that a process can attempt to access the current hashtable by increasing its `busy` counter. However, just before it wants to write the new value for `busy` it falls asleep. When the process wakes up the hashtable might have been deleted and the process would be writing at a random place in memory.

This forces us to use separate arrays `H` and `busy` to store the pointers to hashtables and the `busy` counters. There can be $2P$ hashtables around, because each process can simultaneously be accessing one hashtable and attempting to create a second one. The arrays below are shared variables.

> `H` : **array** 1 . . $2P$ **of pointer to** *Hashtable* ;
> `busy` : **array** 1 . . $2P$ **of** *Nat* ;
> `prot` : **array** 1 . . $2P$ **of** *Nat* ;
> `next` : **array** 1 . . $2P$ **of** 0 . . $2P$.

As indicated, we also need arrays `prot` and `next`. The variable `next`[i] points to the next hashtable to which the contents of hashtable `H`[i] is being copied. If `next`[i] equals 0, this means that there is no next hashtable. The variable `prot`[i] is used to guard the variables `busy`[i], `next`[i] and `H`[i] against being reused for a new table, before all processes have discarded these.

We use a shared variable `currInd` to hold the index of the currently valid hashtable:

> `currInd` : 1 . . $2P$.

Note however that after a process copies `currInd` to its local memory, other processes may create a new hashtable and change `currInd` to point to that one.

3.4 Primary procedures

We first provide the code for the primary procedures, which match directly with the procedures in the interface. Every process has a private variable

index : 1 . . 2P;

containing what it regards as the currently active hashtable. At entry of each primary procedure, it must be the case that the variable `H`[*index*] contains valid information. In section 3.5, we provide procedure *getAccess* with the main purpose to guarantee this property. When *getAccess* has been called, the system is obliged to keep the hashtable at *index* stored in memory, even if there are no accesses to the hashtable using any of the primary procedures. A procedure *releaseAccess* is provided to release resources, and it should be called whenever the process will not access the hashtable for some time.

3.4.1 Syntax

We use a syntax analogous to Modula-3 [5]. We use := for the assignment. We use the C–operations `++` and `--` for atomic increments and decrements. The semicolon is a separator, not a terminator. The basic control mechanisms are

loop .. end is an infinite loop, terminated by **exit** or **return**
while .. do .. end and **repeat .. until ..** are ordinary repetitions
if .. then .. {elsif ..} [else ..] end is the conditional
case .. end is a case statement.

Types are slanted and start with a capital. Shared variables and shared data elements are in typewriter font. Private variables are slanted or in math italic.

3.4.2 The main loop

We model the clients of the hashtable in the following loop. Note that this is not an essential part of the algorithm, but it is needed in the PVS description, and therefore provided here.

```
        loop
0:         getAccess() ;
           loop
1:            choose call; case call of
                 (f, a) with  a ≠ 0 → find(a)
                 (d, a) with  a ≠ 0 → delete(a)
                 (i, v) with  v ≠ null → insert(v)
                 (a, v) with  v ≠ null → assign(v)
                 (r)  → releaseAccess(index); exit
              end
           end
        end
```

The main loop shows that each process repeatedly invokes its four principle operations with correct arguments in an arbitrary, serial manner. Procedure *getAccess* has to provide the client with a protected value for *index*. Procedure *releaseAccess* releases this value and its protection. Note that **exit** means a jump out of the inner loop.

3.4.3 Find

Finding an address in a hashtable with open addressing requires a linear search over the possible hash keys until the address or an empty slot is found. The kernel of procedure *find* is therefore:

```
n := 0 ;
repeat  r := h.table[key(a, l, n)] ;   n++ ;
until  r = null ∨ a = ADR(r) ;
```

The main complication is that the process has to join the migration activity by calling *refresh* when it encounters an entry **done** (i.e. *old*(**null**)).

Apart from a number of special commands, we group statements such that at most one shared variable is accessed and label these 'atomic' statements with a number. The labels are chosen identical to the labels in the PVS code, and therefore not completely consecutive.

In every execution step, one of the processes proceeds from one label to a next one. The steps are thus treated as atomic. The atomicity of steps that refer to shared variables more than once is emphasized by enclosing them in angular brackets. Since procedure calls only modify private control data, procedure headers are not always numbered themselves, but their bodies usually have numbered atomic statements.

```
        proc find(a : Address \ {0}) : Value =
            local r : EValue ; n, l : Nat ; h : pointer to Hashtable ;
5:          h := H[index] ; n := 0 ; {cnt := 0} ;
6:          l := h.size ;
            repeat
7:              ⟨ r := h.table[key(a, l, n)] ;
                  { if r = null ∨ a = ADR(r) then  cnt++ ; (fS)  end } ⟩ ;
8:              if  r = done  then
                    refresh() ;
10:                 h := H[index] ; n := 0 ;
11:                 l := h.size ;
                else  n++  end ;
13:         until r = null ∨ a = ADR(r) ;
14:     return val(r) .
```

In order to prove correctness, we add between braces instructions that only modify auxiliary variables, like the specification variables `X` and rS and other auxiliary variables to be introduced later. The part between braces is comment for the implementation, it only serves in the proof of correctness. The private auxiliary variable *cnt* of type *Nat* counts the number of times (fS) is executed and serves to prove that (fS) is executed precisely once in every call of *find*.

This procedure matches the code of an ordinary find in a hashtable with open addressing, except for the code at the condition $r =$ **done**. This code is needed for the case that the value r is being copied, in which case the new table must be located. Locating the new table is carried out by the procedure *refresh*, which is discussed in Section 3.5. In line 7, the accessed hashtable should be valid (see invariants *fi4* and *He4* in Appendix A). After *refresh* the local variables n, h and l must be reset, to restart the search in the new hashtable. If the procedure terminates, the specifying atomic command (fS) has been executed precisely once (see invariant *Cn1*) and the return values of the specification and the implementation are equal (see invariant *Co1*). If the operation succeeds, the return value must be a valid entry currently associated with the given address in the current hashtable. It is not evident but it has been proved that the linear search of the process executing *find* cannot be violated by other processes, i.e. no other process can *delete*, *insert*, or *rewrite* an entry associated with the same address (as what the process is looking for) in the region where the process has already searched.

We require that there exist at least one **null** entry or **done** entry in any valid hashtable, hence the local variable n in the procedure *find* will never go beyond the size of the hashtable (see invariants *Cu1*, *fi4*, *fi5* and axiom *Ax2*). When the `bound` of the new hashtable is tuned properly before use (see invariants *Ne7*, *Ne8*), the hashtable will not be updated too frequently, and termination of the procedure *find* can be guaranteed.

3.4.4 Delete

Deletion is similar to finding. Since r is a local variable to the procedure *delete*, we regard 18a and 18b as two parts of atomic instruction 18. If the entry is found in the table, then at line 18b this entry is overwritten with the designated element **del**.

```
         proc delete(a : Address \ {0}) : Bool =
             local r : EValue ; k, l, n : Nat ; h : pointer to Hashtable ; suc : Bool ;
15:          h := H[index] ; suc := false ; {cnt := 0} ;
16:          l := h.size ; n := 0 ;
             repeat
17:              k := key(a, l, n) ;
                 ⟨ r := h.table[k] ;
                   { if r = null then cnt++ ; (dS) end } ⟩ ;
18a:             if oldp(r) then
                     refresh() ;
20:                  h := H[index] ;
21:                  l := h.size ; n := 0 ;
                 elsif a = ADR(r) then
18b:                 ⟨ if r = h.table[k] then
                             suc := true ; h.table[k] := del ;
                             { cnt++ ; (dS) ; Y[k] := del }
                       end ⟩ ;
                 else n++ end ;
             until suc ∨ r = null ;
25:          if suc then h.dels++ end ;
26:      return suc .
```

In this procedure, there are two possibilities if r is not outdated in each loop: either deletion fails with $r = \mathbf{null}$ in 17 or deletion succeeds with $r = h.\mathtt{table}[k]$ in 18b. In the latter case, we have in one atomic statement a double access of the shared variable $h.\mathtt{table}[k]$. This is a so-called compare&swap instruction. Atomicity is needed here to preclude interference. The specifying command (dS) is executed either in 17 or in 18, and it is executed precisely once (see invariant *Cn2*), since in 18 the guard $a = ADR(r)$ implies $r \neq \mathbf{null}$ (see invariant *de1* and axiom *Ax1*).

In order to remember the address from the value rewritten to **done** after the value is being copied in the procedure *moveContents*, in 18, we introduce a new auxiliary shared variable `Y` of type array of *EValue*, whose contents equals the corresponding contents of the current hashtable almost everywhere except that the values it contains are not tagged to be *old* or rewritten to be **done** (see invariants *Cu9*, *Cu10*).

Since we postpone the increment of $h.$`dels` until line 25, the field `dels` is a lower bound of the number of positions deleted in the hashtable (see invariant *Cu4*).

3.4.5 Insert

The procedure for insertion in the table is given below. Basically, it is the standard algorithm for insertion in a hashtable with open addressing. Notable is line 28 where the current process finds the current hashtable too full, and orders a new table to be made. We assume that $h.$`bound` is a number less than $h.$`size` (see invariant *Cu3*), which is tuned for optimal performance. Furthermore, in line 35, it can be detected that values in the hashtable have been marked *old*, which is a sign that hashtable h is outdated, and the new hashtable must be located to perform the insertion.

```
         proc insert(v : Value \ {null}) : Bool =
             local r : EValue ; k, l, n : Nat ; h : pointer to Hashtable ;
                 suc : Bool ; a : Address := ADR(v) ;
27:          h := H[index] ; {cnt := 0} ;
28:          if h.occ > h.bound then
```

```
            newTable() ;
30:         h := H[index]  end ;
31:     n := 0 ; l := h.size ; suc := false ;
        repeat
32:         k := key(a, l, n) ;
33:         ⟨ r := h.table[k] ;
              { if  a = ADR(r)  then  cnt++ ;  (iS)  end } ⟩ ;
35a:        if  oldp(r)  then
                refresh() ;
36:             h := H[index] ;
37:             n := 0 ; l := h.size ;
            elseif  r = null then
35b:            ⟨ if  h.table[k] = null  then
                    suc := true ; h.table[k] := v ;
                    { cnt++ ; (iS) ; Y[k] := v }
                  end ⟩ ;
            else  n++  end  ;
        until  suc ∨ a = ADR(r) ;
41:     if  suc  then  h.occ++  end ;
42:   return  suc .
```

Instruction 35b is a test&set instruction, a simpler version of compare&swap. Procedure *insert* terminates successfully when the insertion to an empty slot is completed, or it fails when there already exists an entry with the given address currently in the hashtable (see invariant *Co3* and the specification of *insert*).

3.4.6 Assign

Procedure *assign* is almost the same as *insert* except that it rewrites an entry with a give value even when the associated address is not empty. We provide it without further comments.

```
      proc assign(v : Value \ {null})  =
        local r : EValue ; k, l, n : Nat ; h : pointer to Hashtable ;
            suc : Bool ; a : Address := ADR(v) ;
43:     h := H[index] ; cnt := 0;
44:     if  h.occ > h.bound  then
            newTable() ;
46:         h := H[index]  end ;
47:     n := 0 ; l := h.size ; suc := false ;
        repeat
48:         k := key(a, l, n) ;
49:         r := h.table[k] ;
50a:        if  oldp(r)  then
                refresh() ;
51:             h := H[index] ;
52:             n := 0 ; l := h.size ;
            elsif  r = null ∨ a = ADR(r)  then
50b:            ⟨ if  h.table[k] = r  then
                    suc := true ; h.table[k] := v ;
                    { cnt++ ;  (aS) ; Y[k] := v }
                  end ⟩
            else  n++  end ;
        until suc ;
57:     if  r = null  then  h.occ++  end ;
      end.
```

3.5 Memory management and concurrent migration

In this section, we provide the public procedures *getAccess* and *releaseAccess* and the auxiliary procedures *refresh* and *newTable*. Since *newTable* and *releaseAccess* have the responsibilities for allocations and deallocations, we begin with the treatment of memory by providing a model of the heap.

3.5.1 The model of the heap

We *model* the `Heap` as an infinite array of hashtables, declared and initialized in the following way:

```
Heap : array Nat of Hashtable := ([Nat]⊥) ;
H_index : Nat := 1 .
```

So, initially, `Heap`$[i] = \perp$ for all indices i. The indices of array `Heap` are the pointers to hashtables. We thus simply regard **pointer to** *Hashtable* as a synonym of *Nat*. Therefore, the notation h.`table` used elsewhere in the paper stands for `Heap`$[h]$.`table`. Since we reserve 0 (to be distinguished from the absent hashtable $\perp$ and the absent value **null**) for the null pointer (i.e. `Heap`$[0] = \perp$, see invariant *He1*), we initialize `H_index`, which is the index of the next hashtable, to be 1 instead of 0. Allocation of memory is modeled in

```
proc allocate(s, b : Nat) : Nat =
   ⟨ Heap[H_index] := blank hashtable with size = s, bound = b, occ = dels = 0 ;
     H_index++ ⟩ ;
return H_index ;
```

We assume that *allocate* sets all values in the hashtable `Heap[H_index]` to **null**, and also sets its fields `size`, `bound`, `occ` and `dels` appropriately. Deallocation of hashtables is modeled by

```
proc deAlloc(h : Nat) =
   ⟨ assert Heap[h] ≠ ⊥ ;   Heap[h] := ⊥ ⟩
end .
```

The **assert** in *deAlloc* indicates the obligation to prove that *deAlloc* is called only for allocated memory.

3.5.2 GetAccess

The procedure *getAccess* is defined as follows.

```
        proc getAccess() =
           loop
59:           index := currInd;
60:           prot[index]++ ;
61:           if index = currInd then
62:              busy[index]++ ;
63:              if index = currInd then return ;
                 else releaseAccess(index) end ;
65:           else prot[index]-- end ;
           end
        end.
```

This procedure is a bit tricky. When the process reaches line 62, the *index* has been protected not to be used for creating a new hashtable in the procedure *newTable* (see invariants *pr2*, *pr3* and *nT12*).

The hashtable pointer `H`[*index*] must contain the valid contents after the procedure *getAccess* returns (see invariants *Ot3*, *He4*). So, in line 62, `busy` is increased, guaranteeing that the hashtable will not inadvertently be destroyed (see invariant *bu1* and line 69). Line 63 needs to check the *index* again in case that instruction 62 has the precondition that the hashtable is not valid. Once some process gets hold of one hashtable after calling *getAccess*, no process can throw it away until the process releases it (see invariant *rA7*). Note that this is using *releaseAccess* implicitly done in *refresh*.

3.5.3 ReleaseAccess

The procedure *releaseAccess* is given by

```
    proc releaseAccess(i : 1 . . 2P) =
        local h : pointer to Hashtable ;
67:     h := H[i] ;
68:     busy[i]-- ;
69:     if  h ≠ 0  ∧  busy[i] = 0  then
70:         ⟨ if H[i] = h  then  H[i] := 0 ; ⟩
71:           deAlloc(h) ;
          end ;
        end ;
72:     prot[i]-- ;
    end.
```

Since *deAlloc* in line 71 accesses a shared variable, we have separated its call from 70. The counter busy[i] is used to protect the hashtable from premature deallocation. Only if busy[i]=0, H[i] can be released. The main problem of the design at this point is that it can happen that several processes concurrently execute *releaseAccess* for the same value of i, with interleaving just after the decrement of busy[i]. Then they all may find busy[i] $= 0$. Therefore, a bigger atomic command is needed to ensure that precisely one of them sets H[i] to 0 (line 70) and calls *deAlloc*. Indeed, in line 71, *deAlloc* is called only for allocated memory (see invariant *rA3*). The counter prot[i] can be decreased since position i is no longer used by this process.

3.5.4 NewTable

When the current hashtable has been used for some time, some actions of the processes may require replacement of this hashtable. Procedure *newTable* is called when the number of occupied positions in the current hashtable exceeds the bound (see lines 28, 44). Procedure *newTable* tries to allocate a new hashtable as the successor of the current one (i.e. the next current hashtable). If several processes call *newTable* concurrently, they need to reach consensus on the choice of an index for the next hashtable (line 78). A newly allocated hashtable that will not be used must be deallocated again.

```
    proc newTable() =
        local i : 1 . . 2P ; b, bb : Bool ;
77:     while next[index] = 0  do
78:         choose i ∈ 1 . . 2P ;
            ⟨ b := (prot[i] = 0) ;
              if  b  then prot[i] := 1  end ⟩ ;
            if  b  then
81:             busy[i] := 1 ;
82:             choose bound > H[index].bound − H[index].dels + 2P ;
                choose size > bound + 2P ;
                H[i] := allocate(size, bound) ;
83:             next[i] := 0 ;
84:             ⟨ bb := (next[index] = 0) ;
                  if  bb  then  next[index] := i  end ⟩ ;
                if  ¬bb  then  releaseAccess(i)  end  ;
```

end **end** ;
refresh() ;
end .

In command 82, we allocate a new blank hashtable (see invariant *nT8*), of which the `bound` is set greater than `H`[*index*].`bound` − `H`[*index*].`dels` + $2P$ in order to avoid creating a too small hashtable (see invariants *nT6*, *nT7*). The variables `occ` and `dels` are initially 0 because the hashtable is completely filled with the value **null** at this moment.

We require the `size` of a hashtable to be more than `bound`+$2P$ because of the following scenario: P processes find "h.`occ` > h.`bound`" at line 28 and call *newtable*, *refresh*, *migrate*, *moveContents* and *moveElement* one after the other. After moving some elements, all processes but process p sleep at line 126 with $b_{mE} = true$ (b_{mE} is the local variable b of procedure *moveElement*). Process p continues the migration and updates the new current index when the migration completes. Then, process p does several insertions to let the `occ` of the current hashtable reach one more than its `bound`. Just at that moment, $P-1$ processes wake up, increase the `occ` of the current hashtable to be $P-1$ more, and return to line 30. Since $P-1$ processes insert different values in the hashtable, after $P-1$ processes finish their insertions, the `occ` of the current hashtable reaches $2P-1$ more than its `bound`.

It may be useful to make `size` larger than `bound` + $2P$ to avoid too many collisions, e.g. with a constraint `size` $\geq \alpha \cdot$ `bound` for some $\alpha > 1$. If we did not introduce `dels`, every migration would force the sizes to grow, so that our hashtable would require unbounded space for unbounded life time. We introduced `dels` to avoid this.

Strictly speaking, instruction 82 inspects one shared variable, `H`[*index*], and modifies three other shared variables, viz. `H`[i], `Heap`[`H_index`], and `H_index`. In general, we split such multiple shared variable accesses in separate atomic commands. Here the accumulation is harmless, since the only possible interferences are with other allocations at line 82 and deallocations at line 71. In view of the invariant *Ha2*, all deallocations are at pointers $h <$ `H_index`. Allocations do not interfere because they contain the increment `H_index`++ (see procedure *allocate*).

The procedure *newTable* first searches for a free index i, say by round robin. We use a nondeterministic choice. Once a free index has been found, a hashtable is allocated and the index gets an indirection to the allocated address. Then the current index gets a `next` pointer to the new index, unless this pointer has been set already.

The variables `prot`[i] are used primarily as counters with atomic increments and decrements. In 78, however, we use an atomic test-and-set instruction. Indeed, separation of this instruction in two atomic instructions is incorrect, since that would allow two processes to grab the same index i concurrently.

3.5.5 Migrate

After the choice of the next current hashtable, the procedure *migrate* has the task to transfer the contents in the current hashtable to the next current hashtable by calling a procedure *moveContents* and update the current hashtable pointer afterwards. Migration is complete when at least one of the (parallel) calls to *migrate* has terminated.

proc *migrate*() =
    local $i : 0 \,..\, 2P$; h : **pointer to** *Hashtable* ; b : *Bool* ;
94: $i :=$ `next`[*index*];
95: `prot`[i]++ ;
97: **if** *index* ≠ `currInd` **then**
98:     `prot`[i]-- ;
    else
99:     `busy`[i]++ ;
100:     $h :=$ `H`[i] ;
101:     **if** *index* = `currInd` **then**
        moveContents(`H`[*index*], h) ;

```
103:            ⟨ b := (currInd = index) ;
                  if b then currInd := i ;
                     {Y := H[i].table }
                  end ⟩ ;
                if b then
104:               busy[index]-- ;
105:               prot[index]-- ;
                end ;
             end ;
             releaseAccess(i) ;
       end end .
```

According to invariants *mi4* and *mi5*, it is an invariant that $i = \texttt{next}(\mathit{index}) \neq 0$ holds after instruction 94.

Line 103 contains a compare&swap instruction to update the current hashtable pointer when some process finds that the migration is finished while `currInd` is still identical to its *index*, which means that i is still used for the next current hashtable (see invariant *mi5*). The increments of `prot`$[i]$ and `busy`$[i]$ here are needed to protect the next hashtable. The decrements serve to avoid memory loss.

3.5.6 Refresh

In order to avoid that a process starts migration of an old hashtable, we encapsulate *migrate* in *refresh* in the following way.

```
       proc refresh() =
90:       if index ≠ currInd then
             releaseAccess(index) ;
             getAccess() ;
          else migrate() end ;
       end.
```

When *index* is outdated, the process needs to call *releaseAccess* to abandon its hashtable and *getAccess* to acquire the present pointer to the current hashtable. Otherwise, the process can join the migration.

3.5.7 MoveContents

Procedure *moveContents* has to move the contents of the current table to the next current table. All processes that have access to the table, may also participate in this migration. Indeed, they cannot yet use the new table (see invariants *Ne1* and *Ne3*). We have to take care that delayed actions on the current table and the new table are carried out or aborted correctly (see invariants *Cu1* and *mE10*). Migration requires that every value in the current table be moved to a unique position in the new table (see invariant *Ne19*).

Procedure *moveContents* uses a private variable *toBeMoved* that ranges over sets of locations. The procedure is given by

```
       proc moveContents(from, to : pointer to Hashtable) =
          local i : Nat ; b : Bool ; v : EValue} ; toBeMoved : set of Nat ;
          toBeMoved := {0, ..., from.size − 1} ;
110:      while currInd = index ∧ toBeMoved ≠ ∅ do
111:         choose i ∈ toBeMoved ;
             v := from.table[i] ;
             if from.table[i] = done then
118:            toBeMoved := toBeMoved − {i} ;
             else
```

```
114:          ⟨ b := (v = from.table[i]) ;
                if b then from.table[i] := old(val(v))  end ⟩ ;
              if b then
116:              if  val(v) ≠ null  then  moveElement(val(v), to)  end ;
117:              from.table[i] := done ;
118:              toBeMoved := toBeMoved − {i} ;
      end end end ;
   end .
```

Note that the value is tagged as outdated before being duplicated (see invariant *mC11*). After tagging, the value cannot be deleted or assigned until the migration has been completed. Tagging must be done atomically, since otherwise an interleaving deletion may be lost. When indeed the value has been copied to the new hashtable, in line 117 that value becomes **done** in the hashtable. This has the effect that other processes need not wait for this process to complete procedure *moveElement*, but can help with the migration of this value if needed.

Since the address is lost after being rewritten to **done**, we had to introduce the shared auxiliary hashtable `Y` to remember its value for the proof of correctness. This could have been avoided by introducing a second tagging bit, say for "very old".

The processes involved in the same migration should not use the same strategy for choosing i in line 111, since it is advantageous that *moveElement* is called often with different values. They may exchange information: any of them may replace its set *toBeMoved* by the intersection of that set with the set *toBeMoved* of another one. We do not give a preferred strategy here, one can refer to algorithms for the *write-all* problem [4, 13].

3.5.8 MoveElement

The procedure *moveElement* moves a value to the new hashtable. Note that the value is tagged as outdated in *moveContents* before *moveElement* is called.

```
   proc moveElement(v : Value \ {null}, to : pointer to Hashtable) =
      local a : Address ; k, m, n : Nat ; w : EValue ; b : Bool ;
120:  n := 0 ; b := false ; a := ADR(v) ; m := to.size ;
      repeat
121:      k := key(a, m, n) ; w := to.table[k] ;
          if w = null then
123:          ⟨ b := (to.table[k] = null);
                if  b  then  to.table[k] := v  end ⟩ ;
          else n++  end ;
125:  until  b ∨ a = ADR(w) ∨ currInd ≠ index ;
126:  if  b  then  to.occ++  end
   end .
```

The value is only allowed to be inserted once in the new hashtable (see invariant *Ne19*), otherwise it will violate the main property of open addressing. In total, four situations can occur in the procedure moveElement:

- the current location k contains a value with an other address, the process will increase n and inspect the next location.
- the current location k contains a value with the same address, which means the value has been copied to the new hashtable already. The process therefore terminates.
- the current location k is an empty slot. The process inserts v and returns. If insertion fails, as an other process did fill the empty slot, the search is continued.
- when *index* happens to differ from `currInd`, the whole migration has been completed.

While the current hashtable pointer is not updated yet, there exists at least one **null** entry in the new hashtable (see invariants *Ne8*, *Ne22* and *Ne23*), hence the local variable n in the procedure *moveElement* never goes beyond the size of the hashtable (see invariants *mE3* and *mE8*), and the termination is thus guaranteed.

4 Correctness (Safety)

In this section, we describe the proof of safety of the algorithm. The main aspects of safety are functional correctness, atomicity, and absence of memory loss. These aspects are formalized in eight invariants described in section 4.1. To prove these invariants, we need many other invariants. These are listed in Appendix A. In section 4.2, we sketch the verification of some of the invariants by informal means. In section 4.3, we describe how the theorem prover PVS is used in the verification. As exemplified in 4.2, Appendix B gives the dependencies between the invariants.

Notational Conventions. Recall that there are at most P processes with process identifiers ranging from 1 up to P. We use p, q, r to range over process identifiers, with a preference for p. Since the same program is executed by all processes, every private variable name of a process $\neq p$ is extended with the suffix "." + "process identifier". We do not do this for process p. So, e.g., the value of a private variable x of process q is denoted by $x.q$, but the value of x of process p is just denoted by x. In particular, $pc.q$ is the program location of process q. It ranges over all integer labels used in the implementation.

When local variables in different procedures have the same names, we add an abbreviation of the procedure name as a subscript to the name. We use the following abbreviations: *fi* for *find*, *del* for *delete*, *ins* for *insert*, *ass* for *assign*, *gA* for *getAccess*, *rA* for *releaseAccess*, *nT* for *newTable*, *mig* for *migrate*, *ref* for *refresh*, *mC* for *moveContents*, *mE* for *moveElement*.

In the implementation, there are several places where the same procedure is called, say *getAccess*, *releaseAccess*, etc. We introduce auxiliary private variables *return*, local to such a procedure, to hold the return location. We add a procedure subscript to distinguish these variables according to the above convention.

If V is a set, $\sharp V$ denotes the number of elements of V. If b is a boolean, then $\sharp b = 0$ when b is false, and $\sharp b = 1$ when b is true. Unless explicitly defined otherwise, we always (implicitly) universally quantify over addresses a, values v, non-negative integer numbers k, m, and n, natural number l, processes p, q and r. Indices i and j range over $[1, 2P]$. We abbreviate `H(currInd).size` as *curSize*.

In order to avoid using too many parentheses, we use the usual binding order for the operators. We give "$\wedge$" higher priority than "$\vee$". We use parentheses whenever necessary.

4.1 Main properties

We have proved the following three safety properties of the algorithm. Firstly, the access procedures *find*, *delete*, *insert*, *assign*, are functionally correct. Secondly they are executed atomically. The third safety property is absence of memory loss.

Functional correctness of *find*, *delete*, *insert* is the condition that the result of the implementation is the same as the result of the specification (fS), (dS), (iS). This is expressed by the required invariants:

Co1: $pc = 14 \Rightarrow val(r_{fi}) = rS_{fi}$
Co2: $pc \in \{25, 26\} \Rightarrow suc_{del} = sucS_{del}$
Co3: $pc \in \{41, 42\} \Rightarrow suc_{ins} = sucS_{ins}$

Note that functional correctness of *assign* holds trivially since it does not return a result.

According to the definition of atomicity in chapter 13 of [16], atomicity means that each execution of one of the access procedures contains precisely one execution of the corresponding specifying action (fS), (dS), (iS), (aS). We introduced the private auxiliary variables *cnt* to count the number of times the specifying action is executed. Therefore, atomicity is expressed by the invariants:

Cn1: $pc = 14 \Rightarrow \mathit{cnt}_{\mathit{fi}} = 1$
Cn2: $pc \in \{25, 26\} \Rightarrow \mathit{cnt}_{del} = 1$
Cn3: $pc \in \{41, 42\} \Rightarrow \mathit{cnt}_{ins} = 1$
Cn4: $pc = 57 \Rightarrow \mathit{cnt}_{ass} = 1$

We interpret absence of memory loss to mean that the number of valid hashtables is bounded. More precisely, we prove that this number is bounded by $2P$. This is formalized in the invariant:

No1: $\sharp\{k \mid k < \texttt{H\_index} \ \wedge \ \texttt{Heap}(k) \neq \bot\} \leq 2P$

4.2 Intuitive proof

The eight correctness properties (invariants) mentioned above have been completely proved with the interactive proof checker of PVS. The use of PVS did not only take care of the delicate bookkeeping involved in the proof, it could also deal with many trivial cases automatically. At several occasions where PVS refused to let a proof be finished, we actually found a mistake and had to correct previous versions of this algorithm.

In order to give some feeling for the proof, we describe some proofs. For the complete mechanical proof, we refer the reader to [12]. Note that, for simplicity, we assume that all non-specific private variables in the proposed assertions belong to the general process p, and general process q is an active process that tries to threaten some assertion (p may equal q).

Proof of invariant *Co1* (as claimed in 4.1). According to Appendix B, the stability of *Co1* follows from the invariants *Ot3*, *fi1*, *fi10*, which are given in Appendix A. Indeed, *Ot3* implies that no procedure returns to location 14. Therefore all return statements falsify the antecedent of *Co1* and thus preserve *Co1*. Since r_{fi} and rS_{fi} are private variables to process p, *Co1* can only be violated by process p itself (establishing *pc at 14*) when p executes 13 with $r_{\mathit{fi}} = \textbf{null} \vee a_{\mathit{fi}} = \mathit{ADR}(r_{\mathit{fi}})$. This condition is abbreviated as $\mathit{Find}(r_{\mathit{fi}}, a_{\mathit{fi}})$. Invariant *fi10* then implies that action 13 has the precondition $\mathit{val}(r_{\mathit{fi}}) = rS_{\mathit{fi}}$, so then it does not violate *Co1*. In PVS, we used a slightly different definition of *Find*, and we applied invariant *fi1* to exclude that r_{fi} is **done** or **del**, though invariant *fi1* is superfluous in this intuitive proof. □

Proof of invariant *Ot3*. Since the procedures *getAccess*, *releaseAccess*, *refresh*, *newTable* are called only at specific locations in the algorithm, it is easy to list the potential return addresses. Since the variables *return* are private to process p, they are not modified by other processes. Stability of *Ot3* follows from this. As we saw in the previous proof, *Ot3* is used to guarantee that no unexpected jumps occur. □

Proof of invariant *fi10*. According to Appendix B, we only need to use *fi9* and *Ot3*. Let us use the abbreviation $k = \mathit{key}(a_{\mathit{fi}}, l_{\mathit{fi}}, n_{\mathit{fi}})$. Since r_{fi} and rS_{fi} are both private variables, they can only be modified by process p when p is executing statement 7. We split this situation into two cases

1. with precondition $\mathit{Find}(h_{\mathit{fi}}.\texttt{table}[k], a_{\mathit{fi}})$
   After execution of statement 7, r_{fi} becomes $h_{\mathit{fi}}.\texttt{table}[k]$, and rS_{fi} becomes $\texttt{X}(a_{\mathit{fi}})$. By *fi9*, we get $\mathit{val}(r_{\mathit{fi}}) = rS_{\mathit{fi}}$. Therefore the validity of *fi10* is preserved.
2. otherwise.
   After execution of statement 7, r_{fi} becomes $h_{\mathit{fi}}.\texttt{table}[k]$, which then falsifies the antecedent of *fi10*. □

Proof of invariant *fi9*. According to Appendix B, we proved that *fi9* follows from *Ax2*, *fi1*, *fi3*, *fi4*, *fi5*, *fi8*, *Ha4*, *He4*, *Cu1*, *Cu9*, *Cu10*, and *Cu11*. We abbreviate $\mathit{key}(a_{\mathit{fi}}, l_{\mathit{fi}}, n_{\mathit{fi}})$ as k. We deduce $h_{fi} = \mathtt{H}(\mathit{index})$ from *fi4*, $\mathtt{H}(\mathit{index})$ is not $\perp$ from *He4*, and k is below $\mathtt{H}(\mathit{index}).\mathtt{size}$ from *Ax2*, *fi4* and *fi3*. We split the proof into two cases:

1. $\mathit{index} \neq \mathtt{currInd}$: By *Ha4*, it follows that $\mathtt{H}(\mathit{index}) \neq \mathtt{H}(\mathtt{currInd})$. Hence from *Cu1*, we obtain $h_{fi}.\mathtt{table}[k] = \mathbf{done}$, which falsifies the antecedent of *fi9*.

2. $\mathit{index} = \mathtt{currInd}$: By premise $\mathit{Find}(h_{fi}.\mathtt{table}[k], a_{fi})$, we know that $h_{fi}.\mathtt{table}[k] \neq \mathbf{done}$ because of *fi1*. By *Cu9* and *Cu10*, we obtain $\mathit{val}(h_{fi}.\mathtt{table}[k]) = \mathit{val}(\mathtt{Y}[k])$. Hence it follows that $\mathit{Find}(\mathtt{Y}[k], a_{fi})$. Using *fi8*, we obtain

$$\forall m < n_{fi} : \neg \mathit{Find}(\mathtt{Y}[\mathit{key}(a_{fi}, \mathit{curSize}, m)], a_{fi})$$

   We get n_{fi} is below *curSize* because of *fi5*. By *Cu11*, we conclude

$$\mathtt{X}(a_{fi}) = \mathit{val}(h_{fi}.\mathtt{table}[k])$$

□

4.3 The model in PVS

Our proof architecture (for one property) can be described as a dynamically growing tree in which each node is associated with an assertion. We start from a tree containing only one node, the proof goal, which characterizes some property of the system. We expand the tree by adding some new children via proper analysis of an unproved node (top-down approach, which requires a good understanding of the system). The validity of that unproved node is then reduced to the validity of its children and the validity of some less or equally deep nodes.

Normally, simple properties of the system are proved with appropriate precedence, and then used to help establish more complex ones. It is not a bad thing that some property that was taken for granted turns out to be not valid. Indeed, it may uncover a defect of the algorithm, but in any case it leads to new insights in it.

We model the algorithm as a transition system [17], which is described in the language of PVS in the following way. As usual in PVS, states are represented by a record with a number of fields:

```
State : TYPE = [#
% global variables
   ...
   busy : [ range(2*P) → nat ],
   prot : [ range(2*P) → nat ],
   ...
% private variables:
   index : [ range(P) → range(2*P) ],
   ...
   pc : [ range(P) → nat ], % private program counters
   ...
% local variables of procedures, also private to each process:
% find
   a_find : [ range(P) → Address ],
   r_find : [ range(P) → EValue ],
   ...
% getAccess
   return_getAccess : [ range(P) → nat ],
   ...
#]
```

where *range*(P) stands for the range of integers from 1 to P.

Note that private variables are given with as argument a process identifier. Local variables are distinguished by adding their procedure's names as suffixes.

An action is a binary relation on states: it relates the state prior to the action to the state following the action. The system performed by a particular process is then specified by defining the precondition of each action as a predicate on the state and also the effect of each action in terms of a state transition. For example, line 5 of the algorithm is described in PVS as follows:

```
% corresponding to statement find5: h := H[index]; n := 0;
find5(i,s1,s2) : bool =
    pc(s1)(i)=5 AND
    s2 = s1 WITH [ (pc)(i) := 6,
                   (n_find)(i) := 0,
                   (h_find)(i) := H(s1)(index(s1)(i))
                 ]
    ...
```

where i is a process identifier, s1 is a pre-state, s2 is a post-state.

Since our algorithm is concurrent, the global transition relation is defined as the disjunction of all atomic actions.

```
% transition steps
step(i,s1,s2) : bool =
   find5(i,s1,s2) or find6(i,s1,s2) or ...
   delete15(i,s1,s2) or delete16(i,s1,s2) or ...
   ...
```

Stability for each invariant has been proved by a *Theorem* in PVS of the form:

```
% Theorem about the stability of invariant fi10
IV_fi10: THEOREM
  forall (u,v : state, q : range(P) ) :
    step(q,u,v) AND fi10(u) AND fi9(u) AND ot3(u)
    => fi10(v)
```

To ensure that all proposed invariants are stable, there is a global invariant `INV`, which is the conjunction of all proposed invariants.

```
% global invariant
INV(s:state) : bool =
  He3(s) and He4(s) and Cu1(s) and ...
  ...

% Theorem about the stability of the global invariant INV
IV_INV: THEOREM
  forall (u,v : state, q : range(P) ) :
    step(q,u,v) AND INV(u) => INV(v)
```

We define `Init` as all possible initial states, for which all invariants must be valid.

```
% initial state
Init: { s : state |
    (forall (p: range(P)):
      pc(s)(p)=0 and ...
      ...) and
    (forall (a: Address):
      X(s)(a)=null) and
    ...
    }


% The initial condition can be satisfied by the global invariant INV
IV_Init: THEOREM
  INV(Init)
```

The PVS code contains preconditions to imply well-definedness: e.g. in *find7*, the hashtable must be non-NIL and ℓ must be its size.

```
% corresponding to statement find7
find7(i,s1,s2) : bool =
    i?(Heap(s1)(h_find(s1)(i))) and
    l_find(s1)(i)=size(i_(Heap(s1)(h_find(s1)(i)))) and
    pc(s1)(i)=7 and
    ...
```

All preconditions are allowed, since we can prove lock-freedom in the following form. In every state $s1$ that satisfies the global invariant, every process q can perform a step, i.e., there is a state $s2$ with $(s1, s2) \in \mathit{step}$ and $pc(s1, q) \neq pc(s2, q)$. This is expressed in PVS by

```
% theorem for lock-freedom
IV_prog: THEOREM
  forall (u: state, q: range(P) ) :
    INV(u) => exists (v: state): pc(u)(q) /= pc(v)(q) and step(q,u,v)
```

5 Correctness (Progress)

In this section, we prove that our algorithm is lock-free and almost wait-free. Recall that an algorithm is called *lock-free* if some non-faulty process will finish its task in a finite number of steps, regardless of delays or failures by other processes. This means that no process can block the applications of further operations to the data structure, although any particular operation need not terminate since a slow process can be passed infinitely often by faster processes. An algorithm is called *wait-free* if every process is guaranteed to complete any operation in a finite number of its own steps, regardless of the schedule.

5.1 The easy part of progress

It is clear that *releaseAccess* is wait-free. It follows that the wait-freedom of *migrate* depends on wait-freedom of *moveContents*. If we assume that the choice of i in line 111 is fair, say by round robin, the loop of *moveContents* is bounded. So, wait-freedom of *moveContents* depends on wait-freedom of *moveElement*. It has been proved that n is bounded by m in *moveElement* (see invariants *mE3* and *mE8*). Since, moreover, $to.\mathtt{table}[k] \neq \mathbf{null}$ is stable, the loop of *moveElement* is also bounded. This concludes the sketch that *migrate* is wait-free.

5.2 Progress of newTable

The main part of procedure *newTable* is wait-free. This can be shown informally, as follows. Since we can prove the condition $\mathtt{next}(\mathit{index}) \neq 0$ is stable while process p stays in the region $[77, 84]$, once the condition $\mathtt{next}(\mathit{index}) \neq 0$ holds, process p will exit *newTable* in a few rounds.

Otherwise, we may assume that p has precondition $\mathtt{next}(\mathit{index}) = 0$ before executing line 78. By the invariant

Ne5: $\quad pc \in [1, 58] \vee pc \geq 62 \ \wedge\ pc \neq 65 \ \wedge\ \mathtt{next}(\mathit{index}) = 0 \ \Rightarrow\ \mathit{index} = \mathtt{currInd}$

we get that $\mathit{index} = \mathtt{currInd}$ holds and $\mathtt{next}(\mathtt{currInd}) = 0$ from the precondition. We define two sets of integers:

$$\begin{aligned}
\mathit{prSet1}(i) &= \{r \mid \mathit{index}.r = i \ \wedge\ pc.r \notin \{0, 59, 60\}\} \\
\mathit{prSet2}(i) &= \{r \mid \mathit{index}.r = i \ \wedge\ pc.r \in \{104, 105\} \\
&\qquad \vee\ i_{rA}.r = i \ \wedge\ \mathit{index}.r \neq i \ \wedge\ pc.r \in [67, 72] \\
&\qquad \vee\ i_{nT}.r = i \ \wedge\ pc.r \in [81, 84] \\
&\qquad \vee\ i_{mig}.r = i \ \wedge\ pc.r \geq 97 \ \}
\end{aligned}$$

and consider the sum $\sum_{i=1}^{2P}(\sharp(prSet1(i)) \ + \ \sharp(prSet2(i)))$. While process p is at line 78, the sum cannot exceed $2P-1$ because there are only P processes around and process p contributes only once to the sum. It then follows from the pigeon hole principle that there exists $j \in [1, 2P]$ such that $\sharp(prSet1(j)) \ + \ \sharp(prSet2(j)) = 0$ and $j \neq index.p$. By the invariant

pr1: $\quad \mathtt{prot}[j] = \sharp(prSet1(j)) \ + \ \sharp(prSet2(j)) + \sharp(\mathtt{currInd} = j) + \sharp(\mathtt{next}(\mathtt{currInd}) = j)$

we can get that $\mathtt{prot}[j] = 0$ because of $j \neq index.p = \mathtt{currInd}$.

While `currInd` is constant, no process can modify $\mathtt{prot}[j]$ for $j \neq \mathtt{currInd}$ infinitely often. Therefore, if process p acts infinitely often and chooses its value i in 78 by round robin, process p exits the loop of *newTable* eventually. This shows that the main part of *newTable* is wait-free.

5.3 The failure of wait-freedom

Procedure *getAccess* is not wait-free. When the active clients keep changing the current index faster than the new client can observe it, the accessing client is doomed to starvation.

It may be possible to make a queue for the accessing clients which is emptied by a process in *newTable*. The accessing clients must however also be able to enter autonomously. This would at least add another layer of complications. We therefore prefer to treat this failure of wait-freedom as a performance issue that can be dealt with in practice by tuning the sizes of the hashtables.

Of course, if the other processes are inactive, *getAccess* only requires constant time. Therefore, *getAccess* is lock-free. It follows that *refresh* and *newTable* are lock-free.

According to the invariants *fi5*, *de8*, *in8* and *as6*, the primary procedures *find*, *delete*, *insert*, *assign* are loops bounded by $n \leq h.\mathtt{size}$, so they are wait-free unless n is infinitely often reset to 0. This reset only occurs during migration.

Therefore, if we assume that `occ` is not increased too often beyond `bound` in *insert* and *assign*, the primary procedures are wait-free. Under these circumstances, *getAccess* is also wait-free, and then everything is wait-free.

6 Conclusions

Wait-free shared data objects are implemented without any unbounded busy-waiting loops or idle-waiting primitives. They are inherently resilient to halting failures and permit maximum parallelism. We have presented a new practical algorithm, which is almost wait-free, for concurrently accessible hashtables, which promises more robust performance and reliability than a conventional lock-based implementation. Moreover, the new algorithm is dynamic in the sense that it allows the hashtable to grow and shrink as needed.

The algorithm scales up linearly with the number of processes, provided the function *key* and the selection of i in line 111 are defined well. This is confirmed by some experiments where random values were stored, retrieved and deleted from the hashtable. These experiments indicated that 10^6 insertions, deletions and finds per second and per processor are possible on an SGI powerchallenge with 250Mhz R12000 processors. This figure should be taken as a rough indicator, as the performance of parallel processing is very much influenced by the machine architecture, the relative sizes of data structures compared to sizes of caches, and even the scheduling of processes on processors.

The correctness proof for our algorithm is noteworthy because of the extreme effort it took to finish it. Formal deduction by human-guided theorem proving can, in principle, verify any correct design, but doing so may require unreasonable amounts of effort, time, or skill. Though PVS provided great help for managing and reusing the proofs, we have to admit that the verification for our algorithm was very complicated due to the complexity of our algorithm. The total verification effort can roughly be estimated to consist of two man year excluding the effort in determining the algorithm and writing the documentation. The whole proof contains around 200 invariants. It takes an 1Ghz Pentium IV computer around two days to re-run an individual proof for one of the biggest invariants. Without suitable tool support like PVS, we even doubt if it would be possible to complete a reliable proof of such size and complexity.

Probably, it is possible to simplify the proof and reduce the number of invariants a little bit, but we did not work on this. The complete version of the PVS specifications and the whole proof scripts can be found at [12]. Note that we simplified some definitions in the paper for the sake of presentation.

A Invariants

We present here all invariants whose validity has been verified by the theorem prover PVS.

Conventions. We abbreviate

$$\begin{aligned}
&\mathit{Find}(\mathtt{r}, \mathtt{a}) = \quad \mathtt{r} = \mathbf{null} \ \vee \mathtt{a} = \mathit{ADR}(\mathtt{r}) \\
&\mathit{LeastFind}(a, n) = \quad (\forall m < n : \neg \mathit{Find}(\mathtt{Y}[\mathit{key}(a, \mathit{curSize}, m)], a)) \\
&\qquad\qquad\qquad\qquad \wedge \mathit{Find}(\mathtt{Y}[\mathit{key}(a, \mathit{curSize}, n)], a)) \\
&\mathit{LeastFind}(h, a, n) = \quad (\forall m < n : \neg \mathit{Find}(h.\mathtt{table}[\mathit{key}(a, h.\mathtt{size}, m)], a)) \\
&\qquad\qquad\qquad\qquad \wedge \mathit{Find}(h.\mathtt{table}[\mathit{key}(a, h.\mathtt{size}, n)], a))
\end{aligned}$$

Axioms on functions *key* and *ADR*

Ax1: $v = \mathbf{null} \quad \equiv \quad \mathit{ADR}(v) = \mathbf{0}$
Ax2: $0 \leq \mathit{key}(a, l, k) < l$
Ax3: $0 \leq k < m < l \quad \Rightarrow \quad \mathit{key}(a, l, k) \neq \mathit{key}(a, l, m)$

Main correctness properties

Co1: $\mathit{pc} = 14 \Rightarrow \mathit{val}(r_{\mathit{fi}}) = rS_{\mathit{fi}}$
Co2: $\mathit{pc} \in \{25, 26\} \Rightarrow \mathit{suc}_{del} = \mathit{sucS}_{del}$
Co3: $\mathit{pc} \in \{41, 42\} \Rightarrow \mathit{suc}_{ins} = \mathit{sucS}_{ins}$
Cn1: $\mathit{pc} = 14 \Rightarrow \mathit{cnt}_{\mathit{fi}} = 1$
Cn2: $\mathit{pc} \in \{25, 26\} \Rightarrow \mathit{cnt}_{del} = 1$
Cn3: $\mathit{pc} \in \{41, 42\} \Rightarrow \mathit{cnt}_{ins} = 1$
Cn4: $\mathit{pc} = 57 \Rightarrow \mathit{cnt}_{ass} = 1$

The absence of memory loss is shown by

No1: $\sharp(nbSet1) \leq 2 * P$
No2: $\sharp(nbSet1) = \sharp(nbSet2)$

where $nbSet1$ and $nbSet2$ are sets of integers, characterized by

$$\begin{aligned}
\mathit{nbSet1} &= \{k \mid k < \mathtt{H_index} \ \wedge \ \mathtt{Heap}(k) \neq \bot\} \\
\mathit{nbSet2} &= \{i \mid \mathtt{H}(i) \neq 0 \vee (\exists r : \mathit{pc}.r = 71 \wedge i_{rA}.r = i)\}
\end{aligned}$$

Further, we have the following definitions of sets of integers:

$$\begin{aligned}
\mathit{deSet1} &= \{k \mid k < \mathit{curSize} \ \wedge \ \mathtt{Y}[k] = \mathbf{del}\} \\
\mathit{deSet2} &= \{r \mid \mathit{index}.r = \mathtt{currInd} \ \wedge \ \mathit{pc}.r = 25 \wedge \mathit{suc}_{del}.r\} \\
\mathit{deSet3} &= \{k \mid k < \mathtt{H}(\mathtt{next}(\mathtt{currInd})).\mathtt{size} \ \wedge \ \mathtt{H}(\mathtt{next}(\mathtt{currInd})).\mathtt{table}[k] = \mathbf{del}\}
\end{aligned}$$

$$
\begin{array}{lll}
\mathit{ocSet1} & = & \{r \mid \mathit{index}.r \neq \texttt{currInd} \\
& & \quad \vee\ \mathit{pc}.r \in [30,41] \vee \mathit{pc}.r \in [46,57] \\
& & \quad \vee\ \mathit{pc}.r \in [59,65] \ \wedge\ \mathit{return}_{gA}.r \geq 30 \\
& & \quad \vee\ \mathit{pc}.r \in [67,72] \\
& & \quad\quad \wedge\ (\mathit{return}_{rA}.r = 59 \ \wedge\ \mathit{return}_{gA}.r \geq 30 \\
& & \quad\quad\quad\quad \vee\ \mathit{return}_{rA}.r = 90 \ \wedge\ \mathit{return}_{ref}.r \geq 30) \\
& & \quad \vee\ (\mathit{pc}.r = 90 \vee \mathit{pc}.r \in [104,105]) \ \wedge\ \mathit{return}_{ref}.r \geq 30\} \\
\mathit{ocSet2} & = & \{r \mid \mathit{pc}.r \geq 125 \ \wedge\ b_{mE}.r \ \wedge\ \mathit{to}.r = \texttt{H}(\texttt{currInd})\} \\
\mathit{ocSet3} & = & \{r \mid \mathit{index}.r = \texttt{currInd} \ \wedge\ \mathit{pc}.r = 41 \ \wedge\ \mathit{suc}_{ins}.r \\
& & \quad \vee\ \mathit{index}.r = \texttt{currInd} \ \wedge\ \mathit{pc}.r = 57 \ \wedge\ r_{ass}.r = \mathbf{null}\} \\
\mathit{ocSet4} & = & \{k \mid k < \mathit{curSize} \ \wedge\ \mathit{val}(\texttt{Y}[k]) \neq \mathbf{null}\} \\
\mathit{ocSet5} & = & \{k \mid k < \texttt{H}(\texttt{next}(\texttt{currInd})).\texttt{size} \\
& & \quad \wedge\ \mathit{val}(\texttt{H}(\texttt{next}(\texttt{currInd})).\texttt{table}[k]) \neq \mathbf{null}\} \\
\mathit{ocSet6} & = & \{k \mid k < \texttt{H}(\texttt{next}(\texttt{currInd})).\texttt{size} \\
& & \quad \wedge\ \texttt{H}(\texttt{next}(\texttt{currInd})).\texttt{table}[k] \neq \mathbf{null}\} \\
\mathit{ocSet7} & = & \{r \mid \mathit{pc}.r \geq 125 \ \wedge\ b_{mE}.r \ \wedge\ \mathit{to}.r = \texttt{H}(\texttt{next}(\texttt{currInd}))\}
\end{array}
$$

$$
\begin{array}{lll}
\mathit{prSet1}(i) & = & \{r \mid \mathit{index}.r = i \ \wedge\ \mathit{pc}.r \notin \{0,59,60\}\} \\
\mathit{prSet2}(i) & = & \{r \mid \mathit{index}.r = i \ \wedge\ \mathit{pc}.r \in \{104,105\} \\
& & \quad \vee\ i_{rA}.r = i \ \wedge\ \mathit{index}.r \neq i \ \wedge\ \mathit{pc}.r \in [67,72] \\
& & \quad \vee\ i_{nT}.r = i \ \wedge\ \mathit{pc}.r \in [81,84] \\
& & \quad \vee\ i_{mig}.r = i \ \wedge\ \mathit{pc}.r \geq 97\} \\
\mathit{prSet3}(i) & = & \{r \mid \mathit{index}.r = i \ \wedge\ \mathit{pc}.r \in [61,65] \cup [104,105] \\
& & \quad \vee\ i_{rA}.r = i \ \wedge\ \mathit{pc}.r = 72 \\
& & \quad \vee\ i_{nT}.r = i \ \wedge\ \mathit{pc}.r \in [81,82] \\
& & \quad \vee\ i_{mig}.r = i \ \wedge\ \mathit{pc}.r \in [97,98]\} \\
\mathit{prSet4}(i) & = & \{r \mid \mathit{index}.r = i \ \wedge\ \mathit{pc}.r \in [61,65] \\
& & \quad \vee\ i_{mig}.r = i \ \wedge\ \mathit{pc}.r \in [97,98]\}
\end{array}
$$

$$
\begin{array}{lll}
\mathit{buSet1}(i) & = & \{r \mid \mathit{index}.r = i \\
& & \quad \wedge\ (\mathit{pc}.r \in [1,58] \cup (62,68] \ \wedge\ \mathit{pc}.r \neq 65 \\
& & \quad\quad \vee\ \mathit{pc}.r \in [69,72] \ \wedge\ \mathit{return}_{rA}.r > 59 \\
& & \quad\quad \vee\ \mathit{pc}.r > 72)\} \\
\mathit{buSet2}(i) & = & \{r \mid \mathit{index}.r = i \ \wedge\ \mathit{pc}.r = 104 \\
& & \quad \vee\ i_{rA}.r = i \ \wedge\ \mathit{index}.r \neq i \ \wedge\ \mathit{pc}.r \in [67,68] \\
& & \quad \vee\ i_{nT}.r = i \ \wedge\ \mathit{pc}.r \in [82,84] \\
& & \quad \vee\ i_{mig}.r = i \ \wedge\ \mathit{pc}.r \geq 100\}
\end{array}
$$

We have the following invariants concerning the `Heap`

He1: $\texttt{Heap}(0) = \perp$

He2: $\texttt{H}(i) \neq 0 \equiv \texttt{Heap}(\texttt{H}(i)) \neq \perp$

He3: $\texttt{Heap}(\texttt{H}(\texttt{currInd})) \neq \perp$

He4: $\mathit{pc} \in [1,58] \vee \mathit{pc} > 65 \ \wedge\ \neg(\mathit{pc} \in [67,72] \ \wedge\ i_{rA} = \mathit{index})$
$\Rightarrow \texttt{Heap}(\texttt{H}(\mathit{index})) \neq \perp$

He5: $\texttt{Heap}(\texttt{H}(i)) \neq \perp \Rightarrow\ \texttt{H}(i).\texttt{size} \geq P$

He6: $\texttt{next}(\texttt{currInd}) \neq 0 \Rightarrow \texttt{Heap}(\texttt{H}(\texttt{next}(\texttt{currInd}))) \neq \perp$

Invariants concerning hashtable pointers

Ha1: $\texttt{H\_index} > 0$

Ha2: $\texttt{H}(i) < \texttt{H\_index}$

Ha3: $i \neq j \ \wedge\ \texttt{Heap}(\texttt{H}(i)) \neq \perp \Rightarrow \texttt{H}(i) \neq \texttt{H}(j)$

Ha4: $index \neq \texttt{currInd} \Rightarrow \texttt{H}(index) \neq \texttt{H}(\texttt{currInd})$

Invariants about counters for calling the specification.

Cn5: $pc \in [6,7] \Rightarrow cnt_{fi} = 0$

Cn6:
$$\begin{aligned}&pc \in [8,13]\\ &\vee\ pc \in [59,65] \wedge return_{gA} = 10\\ &\vee\ pc \in [67,72] \wedge (return_{rA} = 59 \wedge return_{gA} = 10\\ &\qquad\qquad\qquad \vee\ return_{rA} = 90 \wedge return_{ref} = 10\\ &\vee\ pc \geq 90 \ \wedge\ return_{ref} = 10\\ &\Rightarrow cnt_{fi} = \sharp(r_{fi} = \mathbf{null} \vee a_{fi} = ADR(r_{fi}))\end{aligned}$$

Cn7:
$$\begin{aligned}&pc \in [16,21] \ \wedge\ pc \neq 18\\ &\vee\ pc \in [59,65] \wedge return_{gA} = 20\\ &\vee\ pc \in [67,72] \wedge (return_{rA} = 59 \wedge return_{gA} = 20\\ &\qquad\qquad\qquad \vee\ return_{rA} = 90 \wedge return_{ref} = 20\\ &\vee\ pc \geq 90 \ \wedge\ return_{ref} = 20\\ &\Rightarrow cnt_{del} = 0\end{aligned}$$

Cn8: $pc = 18 \quad \Rightarrow \quad cnt_{del} = \sharp(r_{del} = \mathbf{null})$

Cn9:
$$\begin{aligned}&pc \in [28,33]\\ &\vee\ pc \in [59,65] \wedge return_{gA} = 30\\ &\vee\ pc \in [67,72] \wedge (return_{rA} = 59 \wedge return_{gA} = 30\\ &\qquad\qquad\qquad \vee\ return_{rA} = 77 \wedge return_{nT} = 30\\ &\qquad\qquad\qquad \vee\ return_{rA} = 90 \wedge return_{ref} = 30\\ &\vee\ pc \in [77,84] \wedge return_{nT} = 30\\ &\vee\ pc \geq 90 \ \wedge\ return_{ref} = 30\\ &\Rightarrow cnt_{ins} = 0\end{aligned}$$

Cn10:
$$\begin{aligned}&pc \in [35,37]\\ &\vee\ pc \in [59,65] \wedge return_{gA} = 36\\ &\vee\ pc \in [67,72] \wedge (return_{rA} = 59 \wedge return_{gA} = 36\\ &\qquad\qquad\qquad \vee\ return_{rA} = 90 \wedge return_{ref} = 36\\ &\vee\ pc \geq 90 \ \wedge\ return_{ref} = 36\\ &\Rightarrow cnt_{ins} = \sharp(a_{ins} = ADR(r_{ins}) \vee suc_{ins})\end{aligned}$$

Cn11:
$$\begin{aligned}&pc \in [44,52]\\ &\vee\ pc \in [59,65] \wedge return_{gA} \in \{46,51\}\\ &\vee\ pc \in [67,72] \wedge (return_{rA} = 59 \wedge return_{gA} \in \{46,51\}\\ &\qquad\qquad\qquad \vee\ return_{rA} = 77 \wedge return_{nT} = 46\\ &\qquad\qquad\qquad \vee\ return_{rA} = 90 \wedge return_{ref} \in \{46,51\}\\ &\vee\ pc \in [77,84] \wedge return_{nT} = 46\\ &\vee\ pc \geq 90 \ \wedge\ return_{ref} \in \{46,51\}\\ &\Rightarrow cnt_{ass}sign = 0\end{aligned}$$

Invariants about old hashtables, current hashtable and the auxiliary hashtable `Y`. Here, we universally quantify over all non-negative integers $n < curSize$.

Cu1:
$$\begin{aligned}&\texttt{H}(index) \neq \texttt{H}(\texttt{currInd}) \ \wedge\ k < \texttt{H}(index).\texttt{size}\\ &\wedge\ (pc \in [1,58] \vee pc > 65 \ \wedge\ \neg(pc \in [67,72] \ \wedge\ i_{rA} = index)\\ &\Rightarrow \quad \texttt{H}(index).\texttt{table}[k] = \mathbf{done}\end{aligned}$$

Cu2: $\sharp(\{k \mid k < \mathit{curSize} \ \wedge\ \mathtt{Y}[k] \neq \mathbf{null}\}) < \mathit{curSize}$

Cu3: $\mathtt{H}(\mathtt{currInd}).\mathtt{bound} + 2 * P < \mathit{curSize}$

Cu4: $\mathtt{H}(\mathtt{currInd}).\mathtt{dels} + \sharp(\mathit{deSet}2) = \sharp(\mathit{deSet}1)$

Cu5: *Cu5* has been eliminated. The numbering has been kept, so as not to endanger the consistency with Appendix B and the PVS script.

Cu6: $\mathtt{H}(\mathtt{currInd}).\mathtt{occ} + \sharp(\mathit{ocSet}1) + \sharp(\mathit{ocSet}2) \leq \mathtt{H}(\mathtt{currInd}).\mathtt{bound} + 2 * P$

Cu7: $\sharp(\{k \mid k < \mathit{curSize} \ \wedge\ \mathtt{Y}[k] \neq \mathbf{null}\} = \mathtt{H}(\mathtt{currInd}).\mathtt{occ} + \sharp(\mathit{ocSet}2) + \sharp(\mathit{ocSet}3)$

Cu8: $\mathtt{next}(\mathtt{currInd}) = 0 \ \Rightarrow\ \neg\, \mathit{oldp}(\mathtt{H}(\mathtt{currInd}).\mathtt{table}[n])$

Cu9: $\neg(\mathit{oldp}(\mathtt{H}(\mathtt{currInd}).\mathtt{table}[n])) \Rightarrow \mathtt{H}(\mathtt{currInd}).\mathtt{table}[n] = \mathtt{Y}[n]$

Cu10: $\mathit{oldp}(\mathtt{H}(\mathtt{currInd}).\mathtt{table}[n]) \ \wedge\ \mathit{val}(\mathtt{H}(\mathtt{currInd}).\mathtt{table}[n]) \neq \mathbf{null}$
$\Rightarrow \mathit{val}(\mathtt{H}(\mathtt{currInd}).\mathtt{table}[n]) = \mathit{val}(\mathtt{Y}[n])$

Cu11: $\mathit{LeastFind}(a, n) \ \Rightarrow\ \mathtt{X}(a) = \mathit{val}(\mathtt{Y}[\mathit{key}(a, \mathit{curSize}, n)])$

Cu12: $\mathtt{X}(a) = \mathit{val}(\mathtt{Y}[\mathit{key}(a, \mathit{curSize}, n)]) \neq \mathbf{null} \ \Rightarrow\ \mathit{LeastFind}(a, n)$

Cu13: $\mathtt{X}(a) = \mathit{val}(\mathtt{Y}[\mathit{key}(a, \mathit{curSize}, n)]) \neq \mathbf{null} \ \wedge\ n \neq m < \mathit{curSize}$
$\Rightarrow \mathit{ADR}(\mathtt{Y}[\mathit{key}(a, \mathit{curSize}, m)]) \neq a$

Cu14: $\mathtt{X}(a) = \mathbf{null} \ \wedge\ \mathit{val}(\mathtt{Y}[\mathit{key}(a, \mathit{curSize}, n)]) \neq \mathbf{null}$
$\Rightarrow \mathit{ADR}(\mathtt{Y}[\mathit{key}(a, \mathit{curSize}, n)]) \neq a$

Cu15: $\mathtt{X}(a) \neq \mathbf{null}$
$\Rightarrow \exists m < \mathit{curSize} : \mathtt{X}(a) = \mathit{val}(\mathtt{Y}[\mathit{key}(a, \mathit{curSize}, m)])$

Cu16: $\exists (f : [\{m : 0 \leq m < \mathit{curSize}) \wedge \mathit{val}(\mathtt{Y}[m]) \neq \mathbf{null}\} \rightarrow$
$\{v : v \neq \mathbf{null} \wedge (\exists k < \mathit{curSize} : v = \mathit{val}(\mathtt{Y}[k]))\}]) :$
f `is bijective`

Invariants about `next` and `next(currInd)`:

Ne1: $\mathtt{currInd} \neq \mathtt{next}(\mathtt{currInd})$

Ne2: $\mathtt{next}(\mathtt{currInd}) \neq 0 \Rightarrow \mathtt{next}(\mathtt{next}(\mathtt{currInd})) = 0$

Ne3: $\mathit{pc} \in [1, 59] \vee \mathit{pc} \geq 62 \ \wedge\ \mathit{pc} \neq 65 \Rightarrow \mathit{index} \neq \mathtt{next}(\mathtt{currInd})$

Ne4: $\mathit{pc} \in [1, 58] \vee \mathit{pc} \geq 62 \ \wedge\ \mathit{pc} \neq 65 \Rightarrow \mathit{index} \neq \mathtt{next}(\mathit{index})$

Ne5: $\mathit{pc} \in [1, 58] \vee \mathit{pc} \geq 62 \ \wedge\ \mathit{pc} \neq 65 \ \wedge\ \mathtt{next}(\mathit{index}) = 0 \ \Rightarrow\ \mathit{index} = \mathtt{currInd}$

Ne6: $\mathtt{next}(\mathtt{currInd}) \neq 0$
$\Rightarrow \sharp(\mathit{ocSet6}) \leq \sharp(\{k \mid k < \mathit{curSize} \ \wedge\ \mathtt{Y}[k] \neq \mathbf{null}\} - \mathtt{H}(\mathtt{currInd}).\mathtt{dels} - \sharp(\mathit{deSet}2)$

Ne7: $\mathtt{next}(\mathtt{currInd}) \neq 0$
$\Rightarrow \mathtt{H}(\mathtt{currInd}).\mathtt{bound} - \mathtt{H}(\mathtt{currInd}).\mathtt{dels} + 2 * P \leq \mathtt{H}(\mathtt{next}(\mathtt{currInd})).\mathtt{bound}$

Ne8: $\mathtt{next}(\mathtt{currInd}) \neq 0$
$\Rightarrow \mathtt{H}(\mathtt{next}(\mathtt{currInd})).\mathtt{bound} + 2 * P < \mathtt{H}(\mathtt{next}(\mathtt{currInd})).\mathtt{size}$

Ne9: $\mathtt{next}(\mathtt{currInd}) \neq 0 \Rightarrow \mathtt{H}(\mathtt{next}(\mathtt{currInd})).\mathtt{dels} = \sharp(\mathit{deSet3})$

Ne9a: $\mathtt{next}(\mathtt{currInd}) \neq 0 \Rightarrow \mathtt{H}(\mathtt{next}(\mathtt{currInd})).\mathtt{dels} = 0$

Ne10: $\mathtt{next}(\mathtt{currInd}) \neq 0 \ \wedge\ k < h.\mathtt{size} \ \Rightarrow\ h.\mathtt{table}[k] \notin \{\mathbf{del}, \mathbf{done}\}$,
where $h = \mathtt{H}(\mathtt{next}(\mathtt{currInd}))$

Ne11: $\mathtt{next}(\mathtt{currInd}) \neq 0 \ \wedge\ k < \mathtt{H}(\mathtt{next}(\mathtt{currInd})).\mathtt{size}$
$\Rightarrow\ \neg \mathit{oldp}(\mathtt{H}(\mathtt{next}(\mathtt{currInd})).\mathtt{table}[k])$

Ne12: $k < \mathit{curSize} \ \wedge\ \mathtt{H}(\mathtt{currInd}).\mathtt{table}[k] = \mathbf{done} \ \wedge\ m < h.\mathtt{size} \ \wedge\ \mathit{LeastFind}(h, a, m)$
$\Rightarrow\ \mathtt{X}(a) = \mathit{val}(h.\mathtt{table}[\mathit{key}(a, h.\mathtt{size}, m)])$,
where $a = \mathit{ADR}(\mathtt{Y}[k])$ and $h = \mathtt{H}(\mathtt{next}(\mathtt{currInd})))$

Ne13: $k < \mathit{curSize} \ \wedge\ \mathtt{H}(\mathtt{currInd}).\mathtt{table}[k] = \mathbf{done} \ \wedge\ m < h.\mathtt{size}$
$\wedge\, \mathtt{X}(a) = \mathit{val}(h.\mathtt{table}[\mathit{key}(a, h.\mathtt{size}, m)]) \neq \mathbf{null}$
$\Rightarrow\ \mathit{LeastFind}(h, a, m)$,
where $a = \mathit{ADR}(\mathtt{Y}[k])$ and $h = \mathtt{H}(\mathtt{next}(\mathtt{currInd}))$

Ne14: $\mathtt{next}(\mathtt{currInd}) \neq 0 \ \wedge\ a \neq \mathbf{0} \ \wedge\ k < h.\mathtt{size}$

$\wedge\ \mathtt{X}(a) = val(h.\mathtt{table}[key(a, h.\mathtt{size}, k)]) \neq \mathbf{null}$
$\Rightarrow\quad LeastFind(h, a, k)$,
where $h = \mathtt{H}(\mathtt{next}(\mathtt{currInd}))$

Ne15: $k < curSize\ \wedge\ \mathtt{H}(\mathtt{currInd}).\mathtt{table}[k] = \mathbf{done}\ \wedge\ \mathtt{X}(a) \neq \mathbf{null}\ \wedge\ m < h.\mathtt{size}$
$\wedge\ \mathtt{X}(a) = val(h.\mathtt{table}[key(a, h.\mathtt{size}, m)])\ \wedge\ n < h.\mathtt{size}\ \wedge\ m \neq n$
$\Rightarrow\quad ADR(h.\mathtt{table}.[key(a, h.\mathtt{size}, n)]) \neq a$,
where $a = ADR(\mathtt{Y}[k])$ and $h = \mathtt{H}(\mathtt{next}(\mathtt{currInd}))$

Ne16: $k < curSize\ \wedge\ \mathtt{H}(\mathtt{currInd}).\mathtt{table}[k] = \mathbf{done}\ \wedge\ \mathtt{X}(a) = \mathbf{null}\ \wedge\ m < h.\mathtt{size}$
$\Rightarrow\quad val(h.\mathtt{table}[key(a, h.\mathtt{size}, m)]) = \mathbf{null}$
$\quad\vee\ ADR(h.\mathtt{table}[key(a, h.\mathtt{size}, m)]) \neq a$,
where $a = ADR(\mathtt{Y}[k])$ and $h = \mathtt{H}(\mathtt{next}(\mathtt{currInd}))$

Ne17: $\mathtt{next}(\mathtt{currInd}) \neq 0\ \wedge\ m < h.\mathtt{size}\ \wedge\ a = ADR(h.\mathtt{table}[m]) \neq 0$
$\Rightarrow\quad \mathtt{X}(a) = val(h.\mathtt{table}[m]) \neq \mathbf{null}$,
where $h = \mathtt{H}(\mathtt{next}(\mathtt{currInd}))$

Ne18: $\mathtt{next}(\mathtt{currInd}) \neq 0\ \wedge\ m < h.\mathtt{size}\ \wedge\ a = ADR(h.\mathtt{table}[m]) \neq 0$
$\Rightarrow\quad \exists n < curSize : val(\mathtt{Y}[n]) = val(h.\mathtt{table}[m])\ \wedge\ oldp(\mathtt{H}(\mathtt{currInd}).\mathtt{table}[n])$,
where $h = \mathtt{H}(\mathtt{next}(\mathtt{currInd}))$

Ne19: $\mathtt{next}(\mathtt{currInd}) \neq 0\ \wedge\ m < h.\mathtt{size}\ \wedge\ a = ADR(h.\mathtt{table}[key(a, h.\mathtt{size}, m)]) \neq 0$
$\wedge\ m \neq n < h.\mathtt{size}$
$\Rightarrow\quad ADR(h.\mathtt{table}[key(a, h.\mathtt{size}, n)]) \neq a$,
where $h = \mathtt{H}(\mathtt{next}(\mathtt{currInd}))$

Ne20: $k < curSize\ \wedge\ \mathtt{H}(\mathtt{currInd}).\mathtt{table}[k] = \mathbf{done}\ \wedge\ \mathtt{X}(a) \neq \mathbf{null}$
$\Rightarrow \exists m < h.\mathtt{size} : \mathtt{X}(a) = val(h.\mathtt{table}[key(a, h.\mathtt{size}, m)])$,
where $a = ADR(\mathtt{Y}[k])$ and $h = \mathtt{H}(\mathtt{next}(\mathtt{currInd}))$

Ne21: *Ne21* has been eliminated.

Ne22: $\mathtt{next}(\mathtt{currInd}) \neq 0 \Rightarrow \sharp(ocSet6) = \mathtt{H}(\mathtt{next}(\mathtt{currInd})).\mathtt{occ} + \sharp(ocSet7)$

Ne23: $\mathtt{next}(\mathtt{currInd}) \neq 0 \Rightarrow \mathtt{H}(\mathtt{next}(\mathtt{currInd})).\mathtt{occ} \leq \mathtt{H}(\mathtt{next}(\mathtt{currInd})).\mathtt{bound}$

Ne24: $\mathtt{next}(\mathtt{currInd}) \neq 0 \Rightarrow \sharp(ocSet5) \leq \sharp(ocSet4)$

Ne25: $\mathtt{next}(\mathtt{currInd}) \neq 0$
$\Rightarrow \exists(f : [\{m : 0 \leq m < h.\mathtt{size} \wedge val(h.\mathtt{table}[m]) \neq \mathbf{null}\} \to$
$\{v : v \neq \mathbf{null} \wedge (\exists k < h.\mathtt{size} : v = val(h.\mathtt{table}[k]))\}]) :$
f `is bijective`,
where $h = \mathtt{H}(\mathtt{next}(\mathtt{currInd}))$

Ne26: $\mathtt{next}(\mathtt{currInd}) \neq 0$
$\Rightarrow \exists(f : [\{v : v \neq \mathbf{null} \wedge (\exists m < h.\mathtt{size} : v = val(h.\mathtt{table}[m]))\} \to$
$\{v : v \neq \mathbf{null} \wedge (\exists k :< curSize : v = val(\mathtt{Y}[k]))\}]) :$
f `is injective`,
where $h = \mathtt{H}(\mathtt{next}(\mathtt{currInd}))$

Ne27: $\mathtt{next}(\mathtt{currInd}) \neq 0\ \wedge\ (\exists n < h.\mathtt{size} : val(h.\mathtt{table}[n]) \neq \mathbf{null})$
$\Rightarrow \exists(f : [\{m : 0 \leq m < h.\mathtt{size} \wedge val(h.\mathtt{table}[m]) \neq \mathbf{null}\} \to$
$\{k : 0 \leq k < curSize \wedge val(\mathtt{Y}[k]) \neq \mathbf{null}\}])$
f `is injective`,
where $h = \mathtt{H}(\mathtt{next}(\mathtt{currInd}))$

Invariants concerning procedure *find* (5...14)

fi1: $a_{fi} \neq \mathbf{0}$

fi2: $pc \in \{6, 11\} \Rightarrow n_{fi} = 0$

fi3: $pc \in \{7, 8, 13\} \Rightarrow l_{fi} = h_{fi}.\mathtt{size}$

fi4: $pc \in [6, 13]\ \wedge\ pc \neq 10 \Rightarrow h_{fi} = \mathtt{H}(index)$

fi5: $pc = 7\ \wedge\ h_{fi} = \mathtt{H}(\mathtt{currInd}) \Rightarrow n_{fi} < curSize$

fi6: $pc = 8\ \wedge\ h_{fi} = \mathtt{H}(\mathtt{currInd})\ \wedge\ \neg Find(r_{fi}, a_{fi})\ \wedge\ r_{fi} \neq \mathbf{done}$
$\Rightarrow \neg\ Find(\mathtt{Y}[key(a_{fi}, curSize, n_{fi})], a_{fi})$

fi7: $pc = 13\ \wedge\ h_{fi} = \mathtt{H}(\mathtt{currInd})\ \wedge\ \neg Find(r_{fi}, a_{fi})\ \wedge\ m < n_{fi}$

$\Rightarrow \quad \neg Find(\mathtt{Y}[key(a_{fi}, curSize, m)], a_{fi})$

fi8: $pc \in \{7, 8\} \ \wedge \ h_{fi} = \mathtt{H}(\mathtt{currInd}) \ \wedge \ m < n_{fi}$
$\Rightarrow \quad \neg Find(\mathtt{Y}[key(a_{fi}, curSize, m)], a_{fi})$

fi9: $pc = 7 \ \wedge \ Find(t, a_{fi}) \Rightarrow \ \mathtt{X}(a_{fi}) = val(t)$,
where $t = h_{fi}.\mathtt{table}[key(a_{fi}, l_{fi}, n_{fi})]$

fi10: $pc \notin (1, 7] \ \wedge \ Find(r_{fi}, a_{fi}) \Rightarrow val(r_{fi}) = rS_{fi}$

fi11: $pc = 8 \ \wedge \ oldp(r_{fi}) \ \wedge \ index = \mathtt{currInd}$
$\Rightarrow \mathtt{next}(\mathtt{currInd}) \neq 0$

Invariants concerning procedure *delete* (15. . . 26)

de1: $a_{del} \neq \mathbf{0}$

de2: $pc \in \{17, 18\} \Rightarrow l_{del} = h_{del}.\mathtt{size}$

de3: $pc \in [16, 25] \ \wedge \ pc \neq 20 \Rightarrow h_{del} = \mathtt{H}(index)$

de4: $pc = 18 \quad \Rightarrow \quad k_{del} = key(a_{del}, l_{del}, n_{del})$

de5: $pc \in \{16, 17\} \vee Deleting \quad \Rightarrow \quad \neg suc_{del}$

de6: $Deleting \ \wedge \ sucS_{del} \quad \Rightarrow \quad r_{del} \neq \mathbf{null}$

de7: $pc = 18 \ \wedge \ \neg \ oldp(h_{del}.\mathtt{table}[k_{del}]) \quad \Rightarrow \quad h_{del} = \mathtt{H}(\mathtt{currInd})$

de8: $pc \in \{17, 18\} \ \wedge \ h_{del} = \mathtt{H}(\mathtt{currInd}) \quad \Rightarrow \quad n_{del} < curSize$

de9: $pc = 18 \ \wedge \ h_{del} = \mathtt{H}(\mathtt{currInd})$
$\wedge \ (val(r_{del}) \neq \mathbf{null} \vee r_{del} = \mathbf{del})$
$\Rightarrow r \neq \mathbf{null} \ \wedge \ (r = \mathbf{del} \vee ADR(r) = ADR(r_{del}))$,
where $r = \mathtt{Y}[key(a_{del}, h_{del}.size, n_{del})]$

de10: $pc \in \{17, 18\} \ \wedge \ h_{del} = \mathtt{H}(\mathtt{currInd}) \ \wedge \ m < n_{del})$
$\Rightarrow \quad \neg Find(\mathtt{Y}[key(a_{del}, curSize, m)], a_{del})$

de11: $pc \in \{17, 18\} \ \wedge \ Find(t, a_{del}) \Rightarrow \mathtt{X}(a_{del}) = val(t)$,
where $t = h_{del}.\mathtt{table}[key(a_{del}, l_{del}, n_{del})]$

de12: $pc = 18 \ \wedge \ oldp(r_{del}) \ \wedge \ index = \mathtt{currInd}$
$\Rightarrow \mathtt{next}(\mathtt{currInd}) \neq 0$

de13: $pc = 18 \quad \Rightarrow \quad k_{del} < \mathtt{H}(index).\mathtt{size}$

where *Deleting* is characterized by

$$
\begin{aligned}
Deleting \quad \equiv \quad & \\
& pc \in [18, 21] \vee pc \in [59, 65] \ \wedge \ return_{gA} = 20 \\
& \vee \ pc \in [67, 72] \ \wedge \ (return_{rA} = 59 \wedge return_{gA} = 20 \\
& \qquad\qquad\qquad\qquad \vee \ return_{rA} = 90 \wedge return_{ref} = 20) \\
& \vee \ pc \geq 90 \ \wedge \ return_{ref} = 20
\end{aligned}
$$

Invariants concerning procedure *insert* (27. . . 52)

in1: $a_{ins} = ADR(v_{ins}) \ \wedge \ v_{ins} \neq \mathbf{null}$

in2: $pc \in [32, 35] \Rightarrow l_{ins} = h_{ins}.\mathtt{size}$

in3: $pc \in [28, 41] \ \wedge \ pc \notin \{30, 36\} \Rightarrow h_{ins} = \mathtt{H}(index)$

in4: $pc \in \{33, 35\} \Rightarrow k_{ins} = key(a_{ins}, l_{ins}, n_{ins})$

in5: $pc \in [32, 33] \vee Inserting \quad \Rightarrow \quad \neg suc_{ins}$

in6: $Inserting \ \wedge \ sucS_{ins} \quad \Rightarrow \quad ADR(r_{ins}) \neq a_{ins}$

in7: $pc = 35 \ \wedge \ \neg \ oldp(h_{ins}.\mathtt{table}[k_{ins}]) \Rightarrow h_{ins} = \mathtt{H}(\mathtt{currInd})$

in8: $pc \in \{33, 35\} \ \wedge \ h_{ins} = \mathtt{H}(\mathtt{currInd}) \quad \Rightarrow \quad n_{ins} < curSize$

in9: $pc = 35 \ \wedge \ h_{ins} = \mathtt{H}(\mathtt{currInd})$
$\wedge \ (val(r_{ins}) \neq \mathbf{null} \vee r_{ins} = \mathbf{del})$
$\Rightarrow r \neq \mathbf{null} \ \wedge \ (r = \mathbf{del} \vee ADR(r) = ADR(r_{ins}))$,
where $r = \mathtt{Y}[key(a_{ins}, h_{ins}.size, n_{ins})]$

in10:	$pc \in \{32, 33, 35\} \;\wedge\; h_{ins} = \mathtt{H}(\mathtt{currInd}) \;\wedge\; m < n_{ins}$
	$\Rightarrow \quad \neg Find(\mathtt{Y}[key(a_{ins}, curSize, m)], a_{ins})$
in11:	$pc \in \{33, 35\} \;\wedge\; Find(t, a_{ins}) \Rightarrow \mathtt{X}(a_{ins}) = val(t)$,
	where $t = h_{ins}.\mathtt{table}[key(a_{ins}, l_{ins}, n_{ins})]$
in12:	$pc = 35 \;\wedge\; oldp(r_{ins}) \;\wedge\; index = \mathtt{currInd}$
	$\Rightarrow \mathtt{next}(\mathtt{currInd}) \neq 0$
in13:	$pc = 35 \Rightarrow k_{ins} < \mathtt{H}(index).\mathtt{size}$

where *Inserting* is characterized by

$$\begin{aligned} Inserting \;\equiv\; & \\ & pc \in [35, 37] \vee pc \in [59, 65] \;\wedge\; return_{gA} = 36 \\ & \vee\; pc \in [67, 72] \;\wedge\; (return_{rA} = 59 \wedge return_{gA} = 36 \\ & \qquad\qquad\qquad\quad \vee\; return_{rA} = 90 \wedge return_{ref} = 36) \\ & \vee\; pc \geq 90 \;\wedge\; return_{ref} = 36 \end{aligned}$$

Invariants concerning procedure *assign* (43. . . 57)

as1:	$a_{ass} = ADR(v_{ass}) \;\wedge\; v_{ass} \neq \mathbf{null}$
as2:	$pc \in [48, 50] \Rightarrow l_{ass} = h_{ass}.\mathtt{size}$
as3:	$pc \in [44, 57] \;\wedge\; pc \notin \{46, 51\} \Rightarrow h_{ass} = \mathtt{H}(index)$
as4:	$pc \in \{49, 50\} \Rightarrow k_{ass} = key(a_{ass}, l_{ass}, n_{ass})$
as5:	$pc = 50 \;\wedge\; \neg\; oldp(h_{ass}.\mathtt{table}[k_{ass}]) \Rightarrow h_{ass} = \mathtt{H}(\mathtt{currInd})$
as6:	$pc = 50 \;\wedge\; h_{ass} = \mathtt{H}(\mathtt{currInd}) \Rightarrow n_{ass} < curSize$
as7:	$pc = 50 \;\wedge\; h_{ass} = \mathtt{H}(\mathtt{currInd})$
	$\wedge\; (val(r_{ass}) \neq \mathbf{null} \vee r_{ass} = \mathbf{del})$
	$\Rightarrow r \neq \mathbf{null} \;\wedge\; (r = \mathbf{del} \vee ADR(r) = ADR(r_{ass}))$,
	where $r = \mathtt{Y}[key(a_{ass}, h_{ass}.size, n_{ass})]$
as8:	$pc \in \{48, 49, 50\} \;\wedge\; h_{ass} = \mathtt{H}(\mathtt{currInd}) \;\wedge\; m < n_{ass}$
	$\Rightarrow \quad \neg Find(\mathtt{Y}[key(a_{ass}, curSize, m)], a_{ass})$
as9:	$pc = 50 \;\wedge\; Find(t, a_{ass}) \Rightarrow \mathtt{X}(a_{ass}) = val(t)$,
	where $t = h_{ass}.\mathtt{table}[key(a_{ass}, l_{ass}, n_{ass})]$
as10:	$pc = 50 \;\wedge\; oldp(r_{ass}sign) \;\wedge\; index = \mathtt{currInd}$
	$\Rightarrow \mathtt{next}(\mathtt{currInd}) \neq 0$
as11:	$pc = 50 \Rightarrow k_{ass} < \mathtt{H}(index).\mathtt{size}$

Invariants concerning procedure *releaseAccess* (67. . . 72)

rA1:	$h_{rA} < \mathtt{H\_index}$
rA2:	$pc \in [70, 71] \Rightarrow h_{rA} \neq 0$
rA3:	$pc = 71 \Rightarrow \mathtt{Heap}(h_{rA}) \neq \perp$
rA4:	$pc = 71 \Rightarrow \mathtt{H}(i_{rA}) = 0$
rA5:	$pc = 71 \Rightarrow h_{rA} \neq \mathtt{H}(i)$
rA6:	$pc = 70 \Rightarrow \mathtt{H}(i_{rA}) \neq \mathtt{H}(\mathtt{currInd})$
rA7:	$pc = 70$
	$\wedge\; (pc.r \in [1, 58] \vee pc.r > 65 \wedge \neg(pc.r \in [67, 72] \wedge i_{rA}.r = index.r))$
	$\Rightarrow \mathtt{H}(i_{rA}) \neq \mathtt{H}(index.r)$
rA8:	$pc = 70 \Rightarrow i_{rA} \neq \mathtt{next}(\mathtt{currInd})$
rA9:	$pc \in [68, 72] \wedge\; (h_{rA} = 0 \vee h_{rA} \neq \mathtt{H}(i_{rA}))$
	$\Rightarrow \mathtt{H}(i_{rA}) = 0$
rA10:	$pc \in [67, 72] \;\wedge\; return_{rA} \in \{0, 59\} \Rightarrow i_{rA} = index$
rA11:	$pc \in [67, 72] \;\wedge\; return_{rA} \in \{77, 90\} \Rightarrow i_{rA} \neq index$
rA12:	$pc \in [67, 72] \;\wedge\; return_{rA} = 77 \Rightarrow \mathtt{next}(index) \neq 0$

rA13: $pc = 71 \ \wedge \ pc.r = 71 \ \wedge \ p \neq r \Rightarrow h_{rA} \neq h_{rA}.r$
rA14: $pc = 71 \ \wedge \ pc.r = 71 \ \wedge \ p \neq r \Rightarrow i_{rA} \neq i_{rA}.r$

Invariants concerning procedure *newTable* (77. . . 84)

nT1: $pc \in [81, 82] \Rightarrow \texttt{Heap}(\texttt{H}(i_{nT})) = \perp$
nT2: $pc \in [83, 84] \Rightarrow \texttt{Heap}(\texttt{H}(i_{nT})) \neq \perp$
nT3: $pc = 84 \Rightarrow \texttt{next}(i_{nT}) = 0$
nT4: $pc \in [83, 84] \Rightarrow \texttt{H}(i_{nT}).\texttt{dels} = 0$
nT5: $pc \in [83, 84] \Rightarrow \texttt{H}(i_{nT}).\texttt{occ} = 0$
nT6: $pc \in [83, 84] \Rightarrow \texttt{H}(i_{nT}).\texttt{bound} + 2 * P < \texttt{H}(i_{nT}).\texttt{size}$
nT7: $pc \in [83, 84] \ \wedge \ index = \texttt{currInd}$
$\Rightarrow \texttt{H}(\texttt{currInd}).\texttt{bound} - \texttt{H}(\texttt{currInd}).\texttt{dels} + 2 * P < \texttt{H}(i_{nT}).\texttt{bound}$
nT8: $pc \in [83, 84] \ \wedge \ k < \texttt{H}(i_{nT}).\texttt{size} \ \Rightarrow \ \texttt{H}(i_{nT}).\texttt{table}[k] = \textbf{null}$
nT9: $pc \in [81, 84] \ \Rightarrow \ i_{nT} \neq \texttt{currInd}$
nT10: $pc \in [81, 84] \ \wedge \ (pc.r \in [1, 58] \vee pc.r \geq 62 \ \wedge \ pc.r \neq 65)$
$\Rightarrow \ i_{nT} \neq index.r$
nT11: $pc \in [81, 84] \ \Rightarrow \ i_{nT} \neq \texttt{next}(\texttt{currInd})$
nT12: $pc \in [81, 84] \Rightarrow \texttt{H}(i_{nT}) \neq \texttt{H}(\texttt{currInd})$
nT13: $pc \in [81, 84]$
$\wedge \ (pc.r \in [1, 58] \vee pc.r > 65 \wedge \neg(pc.r \in [67, 72] \wedge i_{rA}.r = index.r))$
$\Rightarrow \ \texttt{H}(i_{nT}) \neq \texttt{H}(index.r)$
nT14: $pc \in [81, 84] \ \wedge \ pc.r \in [67, 72] \ \Rightarrow \ i_{nT} \neq i_{rA}.r$
nT15: $pc \in [83, 84] \ \wedge \ pc.r \in [67, 72] \ \Rightarrow \ \texttt{H}(i_{nT}) \neq \texttt{H}(i_{rA}.r)$
nT16: $pc \in [81, 84] \ \wedge \ pc.r \in [81, 84] \ \wedge \ p \neq r \ \Rightarrow \ i_{nT} \neq i_{nT}.r$
nT17: $pc \in [81, 84] \ \wedge \ pc.r \in [95, 99] \ \wedge \ index.r = \texttt{currInd}$
$\Rightarrow \ i_{nT} \neq i_{mig}.r$
nT18: $pc \in [81, 84] \ \wedge \ pc.r \geq 99 \ \Rightarrow \ i_{nT} \neq i_{mig}.r$

Invariants concerning procedure *migrate* (94. . . 105)

mi1: $pc = 98 \vee pc \in \{104, 105\} \Rightarrow index \neq \texttt{currInd}$
mi2: $pc \geq 95 \Rightarrow i_{mig} \neq index$
mi3: $pc = 94 \Rightarrow \texttt{next}(index) > 0$
mi4: $pc \geq 95 \Rightarrow i_{mig} \neq 0$
mi5: $pc \geq 95 \Rightarrow i_{mig} = \texttt{next}(index)$
mi6: $pc.r = 70$
$\wedge \ (pc \in [95, 102) \ \wedge \ index = \texttt{currInd} \vee pc \in [102, 103] \vee pc \geq 110)$
$\Rightarrow i_{rA}.r \neq i_{mig}$
mi7: $pc \in [95, 97] \ \wedge \ index = \texttt{currInd} \vee pc \geq 99$
$\Rightarrow i_{mig} \neq \texttt{next}(i_{mig})$
mi8: $(pc \in [95, 97] \vee pc \in [99, 103] \vee pc \geq 110) \ \wedge \ index = \texttt{currInd}$
$\Rightarrow \texttt{next}(i_{mig}) = 0$
mi9: $(pc \in [95, 103] \vee pc \geq 110) \ \wedge \ index = \texttt{currInd}$
$\Rightarrow \texttt{H}(i_{mig}) \neq \texttt{H}(\texttt{currInd})$
mi10: $(pc \in [95, 103] \vee pc \geq 110) \ \wedge \ index = \texttt{currInd}$
$\wedge \ (pc.r \in [1, 58] \vee pc.r \geq 62 \ \wedge \ pc.r \neq 65)$
$\Rightarrow \ \texttt{H}(i_{mig}) \neq \texttt{H}(index.r)$
mi11: $pc = 101 \ \wedge \ index = \texttt{currInd} \vee pc = 102$
$\Rightarrow h_{mig} = \texttt{H}(i_{mig})$
mi12: $pc \geq 95 \ \wedge \ index = \texttt{currInd} \vee pc \in \{102, 103\} \vee pc \geq 110$
$\Rightarrow \ \texttt{Heap}(\texttt{H}(i_{mig})) \neq \perp$
mi13: $pc = 103 \ \wedge \ index = \texttt{currInd} \ \wedge \ k < curSize \ \Rightarrow \ \texttt{H}(index).\texttt{table}[k] = \textbf{done}$

mi14: $pc = 103 \ \wedge\ index = \mathtt{currInd} \ \wedge\ n < \mathtt{H}(i_{mig}).\mathtt{size}$
$\wedge\ \ \mathit{LeastFind}(\mathtt{H}(i_{mig}), a, n)$
$\Rightarrow\ \ \mathtt{X}(a) = \mathit{val}(\mathtt{H}(i_{mig})[\mathit{key}(a, \mathtt{H}(i_{mig}).\mathtt{size}, n)])$

mi15: $pc = 103 \ \wedge\ index = \mathtt{currInd} \ \wedge\ n < \mathtt{H}(i_{mig}).\mathtt{size}$
$\wedge\ \ \mathtt{X}(a) = \mathit{val}(\mathtt{H}(i_{mig}).\mathtt{table}[\mathit{key}(a, \mathtt{H}(i_{mig}).\mathtt{size}, n)] \neq \mathbf{null}$
$\Rightarrow\ \ \mathit{LeastFind}(\mathtt{H}(i_{mig}), a, n)$

mi16: $pc = 103 \ \wedge\ index = \mathtt{currInd} \ \wedge\ k < \mathtt{H}(i_{mig}).\mathtt{size}$
$\Rightarrow\ \ \neg oldp(\mathtt{H}(i_{mig}).\mathtt{table}[k])$

mi17: $pc = 103 \ \wedge\ index = \mathtt{currInd} \ \wedge\ \mathtt{X}(a) \neq \mathbf{null} \ \wedge\ k < h.\mathtt{size}$
$\wedge\, \mathtt{X}(a) = \mathit{val}(h.\mathtt{table}[\mathit{key}(a, h.\mathtt{size}, k)]) \ \wedge\ k \neq n < h.\mathtt{size}$
$\Rightarrow\ \ \mathit{ADR}(h.\mathtt{table}.[\mathit{key}(a, h.\mathtt{size}, n)]) \neq a,$
where $h = \mathtt{H}(i_{mig})$

mi18: $pc = 103 \ \wedge\ index = \mathtt{currInd} \ \wedge\ \mathtt{X}(a) = \mathbf{null} \ \wedge\ k < h.\mathtt{size}$
$\Rightarrow\ \ \mathit{val}(h.\mathtt{table}[\mathit{key}(a, h.\mathtt{size}, k)]) = \mathbf{null}$
$\vee\ \ \mathit{ADR}(h.\mathtt{table}[\mathit{key}(a, h.\mathtt{size}, k)]) \neq a,$
where $h = \mathtt{H}(i_{mig})$

mi19: $pc = 103 \ \wedge\ index = \mathtt{currInd} \ \wedge\ \mathtt{X}(a) \neq \mathbf{null}$
$\Rightarrow\ \ \exists m < h.\mathtt{size} : \mathtt{X}(a) = \mathit{val}(h.\mathtt{table}[\mathit{key}(a, h.\mathtt{size}, m)],$
where $h = \mathtt{H}(i_{mig})$

mi20: $pc = 117 \ \wedge\ \mathtt{X}(a) \neq \mathbf{null} \ \wedge\ \mathit{val}(\mathtt{H}(index).\mathtt{table}[i_{mC}]) \neq \mathbf{null}$
$\vee\ pc \geq 126 \ \wedge\ \mathtt{X}(a) \neq \mathbf{null} \ \wedge\ index = \mathtt{currInd}$
$\vee\ pc = 125 \ \wedge\ \mathtt{X}(a) \neq \mathbf{null} \ \wedge\ index = \mathtt{currInd}$
$\wedge\ \ (b_{mE} \vee \mathrm{val}(w_{mE}) \neq \mathbf{null}$
$\wedge\ \ a_{mE} = \mathit{ADR}(w_{mE}))$
$\Rightarrow \exists m < h.\mathtt{size} : \mathtt{X}(a) = \mathit{val}(h.\mathtt{table}[\mathit{key}(a, h.\mathtt{size}, m)]),$
where $a = \mathit{ADR}(\mathtt{Y}[i_{mC}])$ and $h = \mathtt{H}(\mathtt{next}(\mathtt{currInd}))$

Invariants concerning procedure *moveContents* (110…118):

mC1: $pc = 103 \vee pc \geq 110 \Rightarrow to = \mathtt{H}(i_{mig})$
mC2: $pc \geq 110 \Rightarrow from = \mathtt{H}(index)$
mC3: $pc > 102 \ \wedge\ m \in toBeMoved \Rightarrow m < \mathtt{H}(index).\mathtt{size}$
mC4: $pc = 111 \Rightarrow \exists m < from.\mathtt{size} : m \in toBeMoved$
mC5: $pc \geq 114 \ \wedge\ pc \neq 118 \Rightarrow v_{mC} \neq \mathbf{done}$
mC6: $pc \geq 114 \Rightarrow i_{mC} < \mathtt{H}(index).\mathtt{size}$
mC7: $pc = 118 \Rightarrow \mathtt{H}(index).\mathtt{table}[i_{mC}] = \mathbf{done}$
mC8: $pc \geq 110 \ \wedge\ k < \mathtt{H}(index).\mathtt{size} \ \wedge\ k \notin toBeMoved$
$\Rightarrow \mathtt{H}(index).\mathtt{table}[k] = \mathbf{done}$
mC9: $pc \geq 110 \ \wedge\ index = \mathtt{currInd} \ \wedge\ toBeMoved = \emptyset \ \wedge\ k < \mathtt{H}(index).\mathtt{size}$
$\Rightarrow \mathtt{H}(index).\mathtt{table}[k] = \mathbf{done}$
mC10: $pc \geq 116 \ \wedge\ \mathit{val}(v_{mC}) \neq \mathbf{null}$
$\wedge\, \mathtt{H}(index).\mathtt{table}[i_{mC}] = \mathbf{done}$
$\Rightarrow \mathtt{H}(i_{mig}).\mathtt{table}[\mathit{key}(a, \mathtt{H}(i_{mig}).\mathtt{size}, 0)] \neq \mathbf{null},$
where $a = \mathit{ADR}(v_{mC})$
mC11: $pc \geq 116 \ \wedge\ \mathtt{H}(index).\mathtt{table}[i_{mC}] \neq \mathbf{done}$
$\Rightarrow\ \ \mathit{val}(v_{mC}) = \mathit{val}(\mathtt{H}(index).\mathtt{table}[i_{mC}])$
$\wedge\ \ oldp(\mathtt{H}(index).\mathtt{table}[i_{mC}])$
mC12: $pc \geq 116 \ \wedge\ index = \mathtt{currInd} \ \wedge\ \mathit{val}(v_{mC}) \neq \mathbf{null}$
$\Rightarrow \mathit{val}(v_{mC}) = \mathit{val}(\mathtt{Y}[i_{mC}])$

Invariants concerning procedure *moveElement* (120…126):

mE1: $pc \geq 120 \Rightarrow \mathit{val}(v_{mC}) = v_{mE}$

mE2:	$pc \geq 120 \Rightarrow v_{mE} \neq \mathbf{null}$
mE3:	$pc \geq 120 \Rightarrow to = \mathtt{H}(i_{mig})$
mE4:	$pc \geq 121 \Rightarrow a_{mE} = ADR(v_{mC})$
mE5:	$pc \geq 121 \Rightarrow m_{mE} = to.\mathtt{size}$
mE6:	$pc \in \{121, 123\} \Rightarrow \neg b_{mE}$
mE7:	$pc = 123 \Rightarrow k_{mE} = key(a_{mE}, to.\mathtt{size}, n_{mE})$
mE8:	$pc \geq 123 \Rightarrow k_{mE} < \mathtt{H}(i_{mig}).\mathtt{size}$
mE9:	$pc = 120$ $\wedge\ to.\mathtt{table}[key(ADR(v_{mE}), to.\mathtt{size}, 0)] = \mathbf{null}$ $\Rightarrow index = \mathtt{currInd}$
mE10:	$pc \in \{121, 123\}$ $\wedge\ to.\mathtt{table}[key(a_{mE}, to.\mathtt{size}, n_{mE})] = \mathbf{null}$ $\Rightarrow index = \mathtt{currInd}$
mE11:	$pc \in \{121, 123\}\ \wedge\ pc.r = 103$ $\wedge\ to.\mathtt{table}[key(a_{mE}, to.\mathtt{size}, n_{mE})] = \mathbf{null}$ $\Rightarrow\ \ index.r \neq \mathtt{currInd}$
mE12:	$pc \in \{121, 123\}\ \wedge\ \mathtt{next}(\mathtt{currInd}) \neq 0\ \wedge\ to = \mathtt{H}(\mathtt{next}(\mathtt{currInd}))$ $\Rightarrow n_{mE} < \mathtt{H}(\mathtt{next}(\mathtt{currInd})).\mathtt{size}$
mE13:	$pc \in \{123, 125\}\ \wedge\ w_{mE} \neq \mathbf{null}$ $\Rightarrow\ \ ADR(w_{mE}) = ADR(to.\mathtt{table}[k_{mE}])$ $\vee\ to.\mathtt{table}[k_{mE}] \in \{\mathbf{del}, \mathbf{done}\}$
mE14:	$pc \geq 123\ \wedge\ w_{mE} \neq \mathbf{null}$ $\Rightarrow \mathtt{H}(i_{mig}).\mathtt{table}[k_{mE}] \neq \mathbf{null}$
mE15:	$pc = 117\ \wedge\ val(v_{mC}) \neq \mathbf{null}$ $\vee\ pc \in \{121, 123\}\ \wedge\ n_{mE} > 0$ $\vee\ pc = 125$ $\Rightarrow h.\mathtt{table}[key(ADR(v_{mC}), h.\mathtt{size}, 0)] \neq \mathbf{null}$, where $h = \mathtt{H}(i_{mig})$
mE16:	$pc \in \{121, 123\}$ $\vee\ (pc = 125\ \wedge\ \neg b_{mE}$ $\wedge\ (val(w_{mE}) = \mathbf{null} \vee a_{mE} \neq ADR(w_{mE})))$ $\Rightarrow \forall m < n_{mE}:$ $\neg Find(to.\mathtt{table}[key(a_{mE}, to.\mathtt{size}, m)], a_{mE})$

Invariants about the integer array `prot`.

pr1:	$\mathtt{prot}[i] = \sharp(prSet1(i))\ +\ \sharp(prSet2(i)) + \sharp(\mathtt{currInd} = i) + \sharp(\mathtt{next}(\mathtt{currInd}) = i)$
pr2:	$\mathtt{prot}[\mathtt{currInd}] > 0$
pr3:	$pc \in [1, 58] \vee pc \geq 62\ \wedge\ pc \neq 65 \Rightarrow \mathtt{prot}[index] > 0$
pr4:	$\mathtt{next}(\mathtt{currInd}) \neq 0 \Rightarrow \mathtt{prot}[\mathtt{next}(\mathtt{currInd})] > 0$
pr5:	$\mathtt{prot}[i] = 0 \Rightarrow \mathtt{Heap}(\mathtt{H}[i]) = \bot$
pr6:	$\mathtt{prot}[i] \leq \sharp(prSet3(i))\ \wedge\ \mathtt{busy}[i] = 0\ \ \Rightarrow\ \ \mathtt{Heap}(\mathtt{H}[i]) = \bot$
pr7:	$pc \in [67, 72] \Rightarrow \mathtt{prot}[i_{rA}] > 0$
pr8:	$pc \in [81, 84]\ \ \Rightarrow\ \ \mathtt{prot}[i_{nT}] > 0$
pr9:	$pc \geq 97 \Rightarrow \mathtt{prot}[i_{mig}] > 0$
pr10:	$pc \in [81, 82]\ \ \Rightarrow\ \ \mathtt{prot}[i_{nT}] = \sharp(prSet4(i_{nT})) + 1$

Invariants about the integer array `busy`.

bu1:	$\mathtt{busy}[i] = \sharp(buSet1(i))\ +\ \sharp(buSet2(i)) + \sharp(\mathtt{currInd} = i) + \sharp(\mathtt{next}(\mathtt{currInd}) = i)$
bu2:	$\mathtt{busy}[\mathtt{currInd}] > 0$
bu3:	$pc \in [1, 58]$ $\vee\ pc > 65\ \wedge\ \neg(i_{rA} = index\ \wedge\ pc \in [67, 72])$

$\Rightarrow \texttt{busy}[index] > 0$

bu4: $\texttt{next}(\texttt{currInd}) \neq 0 \Rightarrow \texttt{busy}[\texttt{next}(\texttt{currInd})] > 0$

bu5: $pc = 81 \Rightarrow \texttt{busy}[i_{nT}] = 0$

bu6: $pc \geq 100 \Rightarrow \texttt{busy}[i_{mig}] > 0$

Some other invariants we have postulated:

Ot1: $\texttt{X}(\mathbf{0}) = \mathbf{null}$

Ot2: $\texttt{X}(a) \neq \mathbf{null} \Rightarrow ADR(\texttt{X}(a)) = a$

The motivation of invariant (Ot1) is we never store a value for the address 0. The motivation of invariant (Ot2) is that the address in the hashtable is unique.

Ot3: $return_{gA} = \{1, 10, 20, 30, 36, 46, 51\} \;\wedge\; return_{rA} = \{0, 59, 77, 90\}$
$\wedge\; return_{ref} = \{10, 20, 30, 36, 46, 51\} \;\wedge\; return_{nT} = \{30, 46\}$

Ot4: $pc \in \{0,\ 1,\ 5,\ 6,\ 7,\ 8,\ 10,\ 11,\ 13,\ 14,\ 15,\ 16,\ 17,\ 18,\ 20,$
$21,\ 25,\ 26,\ 27,\ 28,\ 30,\ 31,\ 32,\ 33,\ 35,\ 36,\ 37,\ 41,$
$42,\ 43,\ 44,\ 46,\ 47,\ 48,\ 49,\ 50,\ 51,\ 52,\ 57,\ 59,\ 60,$
$61,\ 62,\ 63,\ 65,\ 67,\ 68,\ 69,\ 70,\ 71,\ 72,\ 77,\ 78,\ 81,$
$82,\ 83,\ 84,\ 90,\ 94,\ 95,\ 97,\ 98,\ 99,\ 100,\ 101,\ 102,$
$103,\ 104,\ 105,\ 110,\ 111,\ 114,\ 116,\ 117,\ 118,\ 120,$
$121,\ 123,\ 125,\ 126\}$

B Dependencies between invariants

Let us write "φ **from** $\psi_1, \cdots, \psi_n$" to denote that φ can be proved to be an invariant using $\psi_1, \cdots, \psi_n$ hold. We write "$\varphi \Leftarrow \psi_1, \cdots, \psi_n$" to denote that φ can be directly derived from $\psi_1, \cdots, \psi_n$. We have verified the following "**from**" and "$\Leftarrow$" relations mechanically:

Co1 **from** fi10, Ot3, fi1
Co2 **from** de5, Ot3, de6, del, de11
Co3 **from** in5, Ot3, in6, in1, in11
Cn1 **from** Cn6, Ot3
Cn2 **from** Cn8, Ot3, del
Cn3 **from** Cn10, Ot3, in1, in5
Cn4 **from** Cn11, Ot3
No1 $\Leftarrow$ No2
No2 **from** nT1, He2, rA2, Ot3, Ha2, Ha1, rA1, rA14, rA3, nT14, rA4
He1 **from** Ha1
He2 **from** Ha3, rA5, Ha1, He1, rA2
He3, He4 **from** Ot3, rA6, rA7, mi12, rA11, rA5
He5 **from** He1
He6 **from** rA8, Ha3, mi8, nT2, rA5
Ha1 **from** true
Ha2 **from** Ha1
Ha3 **from** Ha2, Ha1, He2, He1
Ha4 $\Leftarrow$ Ha3, He3, He4
Cn5 **from** Cn6, Ot3
Cn6 **from** Cn5, Ot3
Cn7 **from** Cn8, Ot3, del
Cn8 **from** Cn7, Ot3
Cn9 **from** Cn10, Ot3, in1, in5

Cn10 **from** Cn9, Ot3, in5
Cn11 **from** Cn11, Ot3
Cu1 **from** Ot3, Ha4, rA6, rA7, nT13, nT12, Ha2, He3, He4, rA11, nT9, nT10, mi13, rA5
Cu2 ⇐ Cu6, cu7, Cu3, He3, He4
Cu3 **from** rA6, rA7, nT13, nT12, mi5, mi4, Ne8, rA5
Cu4 **from** del, in1, as1, rA6, rA7, Ha2, nT13, nT12, Ne9, Cu9, Cu10, de7, in7, as5, He3, He4, mi5, mi4, Ot3, Ha4, de3, mi9,mi10, de5, rA5
Cu6 **from** Ot3, rA6, rA7, Ha2, nT13, nT12, Ha3, in3, as3, Ne23, mi5, mE6, mE7, mE10, mE3, Ne3, mi1, mi4, rA5
Cu7 **from** Ot3, rA6, rA7, Ha2, nT13, nT12, Ha3, in3, as3, in5, mi5, mE6, mE7, mE10, mE3, Ne3, mi4, de7, in7, as5, Ne22, mi9,mi10, rA5, He3, mi12, mi1, Cu9, de1 in1, as1
Cu8 **from** Cu8, FT, Ha2, nT9, nT10, rA6, rA7, mi5, mi4 , mC2, mC5, He3, He4, Cu1, Ha4, mC6, mi16, rA5
Cu9,Cu10 **from** rA6, rA7, nT13, nT12, Ha2, He3, He4, Cu1, Ha4, de3, in3, as3, mE3, mi9, mi10, mE10, mE7, rA5
Cu11, Cu12 **from** Cu9, Cu10, Cu13, Cu14, del, in1, as1, rA6, rA7, Ha2, nT13, nT12, He3, He4, Cu1, Ha4, in3, as3, mi14, mi15, de3, in10, as8, mi12, Ot2, fi5, de8, in8, as6, Cu15, de11, in11, rA5
Cu13, Cu14 **from** He3, He4, Ot2, del, in1, as1, Ot1, rA6, rA7, nT13, nT12, Ha2, Cu9, Cu10, Cu1, Ha4, de3, in3, as3, Cu11, Cu12, in10, as8, fi5, de8, in8, as6, Cu15, mi17, mi18, mi12, mi4, de11, rA5
Cu15 **from** He3, He4, rA6, rA7, nT13, nT12, Ha2, Cu1, Ha4, del, in1, as1, de3, in3, as3, fi5, de8, in8, as6, mi12, mi19, mi4, Ot2, Cu9, Cu10, Cu11, Cu12, Cu13, Cu14, rA5
Cu16 ⇐ Cu13, Cu14, Cu15, He3, He4, Ot1
Ne1 **from** nT9, nT10, mi7
Ne2 **from** Ne5, nT3, mi8, nT9, nT10
Ne3 **from** Ne1, nT9, nT10, mi8
Ne4 **from** Ne1, nT9, nT10
Ne5 **from** Ot3, nT9, nT10, mi5
Ne6 ⇐ Ne10, Ne24, He6, He3, He4, Cu4
Ne7 **from** Ha3, rA6, rA7, rA8, nT13, nT12, nT11, He3, He4, mi8, nT7, Ne5, Ha2, He6, rA5
Ne8 **from** Ha3, rA8, nT11, T, mi8, nT6, Ne5, rA5
Ne9 **from** Ha3, Ha2, Ne3, Ne5, de3, as3, rA8, rA6, rA7, nT8, nT11, mC2, nT4, mi8, rA5
Ne9a **from** Ha3, Ne3, rA5, de3, rA8, nT4, mi8
Ne10 **from** Ha3, Ha2, de3, rA8, nT11, Ne3, He6, mi8, nT8, mC2, nT2, Ne5, rA5
Ne11 **from** Ha3, Ha2, He6, T, nT2, nT8, rA8, nT11, mi8, Ne3, mC2, rA5
Ne12, Ne13 **from** Ha3, Ha2, Cu8, He6, He3, He4, Cu1, de3, in3, as3, rA8, rA6, rA7, nT11, nT13, nT12, mi12, mi16, mi5, mi4, de7, in7, as5, Ot2, del,in1, as1, Cu9, Cu10, Cu13, Cu14, Cu15, as9, fi5, de8, in8, as6, mC2, Ne3, Ot1, Ne14, Ne20, mE16, mE7, mE4, mE1, mE12, mE2, Ne15, Ne16, Ne17, Ne18, mi20, de11, in11, rA5
Ne14 **from** Ha3, Ha2, He6, He3, He4, T, nT2, nT8, de3, in3, as3, rA8, nT11, Ot2, del, in1, as1, Cu9, Cu10, mi8, Ne3, mC2, mE7, mE16, mE1, mE4, mE12, Ne17, Ne18, Cu1, rA5
Ne15, Ne16 **from** Ha3, Ha2, Cu8, He6, He3, He4, Cu1, de3, in3, as3, rA8, rA6, rA7, nT11, nT13, nT12, mi12, mi16, mi5, mi4, de7, in7, as5, Ot2, del, in1, as1, Cu9, Cu10, Cu13, Cu14, Cu15, as9, fi5, de8, in8, as6, mC2, Ne3, Ot1, Ne19, Ne20, Ne12, Ne13, mE16, mE7, mE4, mE1, mE12, mE10, mE2, in11, de11, rA5
Ne17, Ne18 **from** Ha3, Ha2, mi8, He6, He3, He4, Cu1, nT2, de3, in3, as3, rA8, rA6, rA7, nT11, nT13, nT12, de7, in7, as5, Ot2, del, in1, as1, Cu9, Cu10, T, nT8, mE2, fi5, de8, in8, as6, mC2, Ne3, mC11, mC6, mC12, mE7, mE10, mE1, Cu8, Cu15, Cu13, Cu14, Cu11, Cu12, as8, de11, rA5
Ne19 **from** Ha3, Ha2, He6, nT2, nT8, de3, in3, as3, rA8, nT11, mi8, Ne3, mE7, Ne14, mE16, Ot1, mE1, mE4, mE12, Ne17, Ne18, rA5
Ne20 **from** Ha3, Ha2, Cu8, He6, He3, He4, Cu1, Ha4, de3, in3, as3, rA8, rA6, rA7, nT11, nT13, nT12, mi12, mi16, mi5, mi4, Ne1, de7, in7, as5, del, in1, as1, Cu9, Cu10, Cu13,

Cu14, Cu15, as9, fi5, de8, in8, as6, mC2, Ne3, Ot1, mi20, in11, rA5

Ne22 **from** Ot3, rA8, Ha2, nT11, Ha3, de3, in3, as3, mi5, mi4, Ne3, nT18, mE3, mi8, mE10, mE7, mE6, Ne5, nT5, nT2, rA5, nT8, nT12, mC2, mE2

Ne23 ⇐ Cu6, cu7, Ne6, Ne7, He3, He4, Ne22, He6

Ne24 ⇐ Ne27, He6

Ne25 ⇐ Ne19, Ne17, Ne18, He6

Ne26 ⇐ Ne17, Ne18, He6

Ne27 ⇐ Cu16, Ne25, Ne26, Ne17, Ne18, He6

fi1, de1, in1, as1 **from**

fi2 **from** fi2, Ot3

fi3 **from** fi4, Ot3, rA6, rA7, Ha2, rA5

fi4 **from** Ot3, rA6, rA7, nT13, nT12

fi5, de8, in8, as6 ⇐ Cu2, de10, in10, as8, fi8, He3, He4

fi6 **from** Ot3, fi1, de1, in1, as1, rA6, rA7, Ha2, nT13, nT12, mi9,mi10, Cu9, Cu10, He3, He4, Cu1, Ha4, fi4, in3, as3, rA5

fi7 **from** fi8, fi6, fi2, Ot3, fi1, de1, in1, as1, rA6, rA7, Ha2, nT13, nT12, mi9,mi10, Cu9, Cu10, He3, He4, Cu1, Ha4, fi4, in3, as3, rA5

fi8 **from** fi4, fi7, fi2, Ot3, fi1, de1, in1, as1, rA6, rA7, Ha2, nT13, nT12, mi9,mi10, Cu9, Cu10, He3, He4, Cu1, Ha4, in3, as3, rA5

fi9 ⇐ Cu1, Ha4, Cu9, Cu10, Cu11, Cu12, fi8, fi3, fi4, fi5, de8, in8, as6, He3, He4

fi10 **from** fi9, Ot3

fi11, de12, in12, as10 **from** Ot3, nT9, nT10, mi9,mi10, Cu8, fi4, de3, in3, as3, fi3, de2, in2, as2

de2 **from** de3, Ot3, rA6, rA7, Ha2, rA5

de3 **from** Ot3, rA6, rA7, nT13, nT12

de4, in4, as4 **from** Ot3

de5 **from** Ot3

de6 **from** Ot3, de1, de11

de7, in7, as5 ⇐ de3, in3, as3, Cu1, Ha4, de13, in13, as11

de9 **from** Ot3, de1, in1, as1, rA6, rA7, Ha2, nT13, nT12, mi9,mi10, Cu9, Cu10, de3, de7, in7, as5, rA5

de10 **from** de3, de9, Ot3, de1, in1, as1, rA6, rA7, Ha2, nT13, nT12, mi9,mi10, Cu9, Cu10, de7, in7, as5, He3, He4, rA5

de11 ⇐ de10, de2, de3, He3, He4, Cu1, Ha4, Cu9, Cu10, Cu11, Cu12, fi5, de8, in8, as6

de13, in13, as11 ⇐ Ax2, de2, de3, de4, in2, in3, in4, as2, as3, as4

in2 **from** in3, Ot3, rA6, rA7, Ha2, rA5

in3 **from** Ot3, rA6, rA7, nT13, nT12

in5 **from** Ot3

in6 **from** Ot3, in1, in11

in9 **from** Ot3, de1, in1, as1, rA6, rA7, Ha2, nT13, nT12, mi9,mi10, Cu9, Cu10, He3, He4, in3, de7, in7, as5, rA5

in10 **from** in9, fi2, Ot3, de1, in1, as1, rA6, rA7, Ha2, nT13, nT12, mi9,mi10, Cu9, Cu10, He3, He4, in3, de7, in7, as5, rA5

in11 ⇐ in10, in2, in3, Cu1, Ha4, Cu9, Cu10, Cu11, Cu12, fi5, de8, in8, as6

as2 **from** as3, He3, He4, Ot3, rA6, rA7, Ha2, rA5

as3 **from** Ot3, rA6, rA7, nT13, nT12

as7 **from** Ot3, de1, in1, as1, rA6, rA7, Ha2, nT13, nT12, mi9,mi10, Cu9, Cu10, as3, de7, in7, as5, rA5

as8 **from** as7, Ot3, de1, in1, as1, rA6, rA7, Ha2, nT13, nT12, mi9,mi10, Cu9, Cu10, He3, He4, as3, de7, in7, as5, rA5

as9 ⇐ as8, as2, as3, He3, He4, Cu1, Ha4, Cu9, Cu10, Cu11, Cu12, fi5, de8, in8, as6

rA1 **from** Ha2

rA2 **from** Ot3

rA3 **from** Ot3, rA9, He2, He1, rA2, rA13
rA4 **from** Ot3, nT14
rA5 **from** Ot3, rA1, rA2, Ha3, He2
rA6, rA7 **from** Ot3, nT13, nT12, nT14, rA11, mi4, bu2, bu3, Ha3, mi6, Ha2, He3, He4, He2, rA2
rA8 **from** Ot3, bu4, nT14, mi6, Ne2, mi5
rA9 **from** Ot3, Ha2, nT14, He1, He2
rA10 **from** Ot3
rA11 **from** Ot3, nT13, nT12, mi2
rA12 **from** Ot3, nT9, nT10
rA13 **from** Ot3, rA5
rA14 **from** Ot3, rA4, He1, rA2
nT1 **from** Ot3, pr5, Ha3, nT14, nT16, Ha2
nT2 **from** Ot3, nT14, Ha3, rA5
nT3 **from** Ot3, nT9, nT10
nT4 **from** Ot3, Ha3, de3, nT13, nT12, nT15, rA5
nT5 **from** Ot3, Ha3, in3, as3, nT13, nT12, nT15, nT18, mE3, mi4, rA5
nT6 **from** Ot3, nT13, nT12, nT14, Ha3, rA5
nT7 **from** Ot3, nT13, nT12, nT15, rA6, rA7, Ha2, mi9,mi10, nT14, Ha3, nT16, rA5
nT8 **from** Ot3, de3, in3, as3, nT13, nT12, nT15, nT18, mE3, mi4, Ha3, mC2, nT16, nT2, Ha2, rA5
nT9, nT10 **from** Ot3, pr2, pr3, nT18
nT11 **from** Ot3, pr4, nT16, mi8
nT13, nT12 $\Leftarrow$ nT9, nT10, Ha3, He3, He4
nT14 **from** Ot3, nT9, nT10, nT18, nT16, pr7
nT15 $\Leftarrow$ nT14, Ha3, nT2
nT16 **from** Ot3, pr8
nT17 **from** Ot3, mi5, pr4, nT11, mi10
nT18 **from** Ot3, pr9, mi5, nT11
mi1 **from** Ot3, mi9,mi10, mi10
mi2 **from** Ot3, Ne4
mi3 **from** Ot3, fi11, de12, in12, as10, nT9, nT10, Ne5
mi4 **from** Ot3, mi9,mi10, mi3
mi5 **from** Ot3, nT9, nT10, Ne5, mi10, mi4
mi6 **from** Ot3, mi5, bu6, rA8, mi9, mi10, bu4, mi4
mi7 **from** Ot3, mi2, mi7, mi4, nT18, Ne2, mi10, nT17, mi3
mi8 **from** Ot3, mi10, Ne2, mi3
mi9, mi10 **from** Ot3, He3, He4, nT9, nT10, nT18, Ne3, Ha3, mi3, nT17, mi10, He2, mi4, mi12, mi6, He6
mi11 **from** Ot3, nT18, mi9, mi6, mi6
mi12 **from** Ot3, rA8, nT2, He6, mi9, mi5, mi3, Ha3, mi4, rA5
mi12 **from** Ot3, mi12, nT18, mi6, Ha3, mi4, rA5
mi13 **from** Ot3, rA6, rA7, Ha2, nT13, nT12, He3, He4, mi9,mi10, mC9, rA5
mi14, mi15 $\Leftarrow$ Ne12, Ne13, mi5, Cu15, mi13, Ot2, He3, He4, Ne17, Ne18, Cu8, He6, He5, mi4, Ot1
mi16 $\Leftarrow$ Ne11, mi5, mi4
mi17, mi18 $\Leftarrow$ Ne15, Ne16, mi5, Cu15, mi13, Ot2, He3, He4, Ne17, Ne18, Cu8, He6, He5, mi4
mi19 $\Leftarrow$ Ne20, mi5, Cu15, mi13, Ot2, He3, He4
mi20 **from** Ha3, Ha2, Cu8, He6, He3, He4, Cu1, Ha4, de3, in3, as3, rA8, rA6, rA7, nT11, nT13, nT12, mi5, mi4, de7, in7, as5, Ot2, del, in1, as1, Cu9, Cu10, Cu13, Cu14, Cu15, as9, fi5, de8, in8, as6, mC6, Ne3, Ot3, mC11, mi13, mi9,mi10, mC2, mE3, mE10, mE7, mC12, mE1, mE13, Ne17, Ne18, mE2, mE4, Ot1, mE6, Ne10, in11, rA5
mC1 **from** Ot3, mi6, mi11, nT18
mC2 **from** Ot3, rA6, rA7, nT13, nT12, mC2

mC3 **from** Ot3, mC3, nT13, nT12, rA6, rA7, Ha2, rA5
mC4 **from** Ot3, mC4, mC2, mC3, He3, He4, rA6, rA7, Ha2, rA5
mC5 **from** Ot3
mC6 **from** Ot3, rA6, rA7, Ha2, nT13, nT12, mC2, rA5
mC7 **from** Ot3, rA6, rA7, Ha2, nT13, nT12, mC2, rA5
mC8 **from** Ot3, rA6, rA7, Ha2, nT13, nT12, He3, He4, mC7, rA5
mC9 **from** Ot3, rA6, rA7, Ha2, nT13, nT12, He3, He4, mi9,mi10, He5, mC7, mC8, rA5
mC10 **from** Ot3, rA6, rA7, Ha2, nT13, nT12, mC2, del, in1, as1, mi6, Ha3, mi4, nT18, mE15, mC11, mi5, rA5
mC11 **from** Ot3, rA6, rA7, Ha2, nT13, nT12, mC2, rA5
mC12 **from** Ot3, rA6, rA7, mC2, mC11, Cu9, Cu10, de7, in7, as5, mi9, mC6
mE1 **from** Ot3
mE2 **from** Ot3
mE3 **from** mC1, Ot3, mi6, nT18
mE4 **from** Ot3, mE1
mE5 **from** Ot3, mE3, Ha3, mi6, mi4, nT18, Ha2, rA5
mE6 **from** Ot3
mE7 **from** Ot3, Ha2, Ha3, mi6, mi4, mE3, rA5
mE8 **from** Ot3, Ha3, mi6, mi4, nT18, Ha2, mE3, rA5
mE9 **from** Cu1, Ha4, Ot3, Ha2, Ha3, mi6, mi4, mE3, mC2, mC10, mE1, mC1, del, in1, as1, mi13, mi12, mC6, mE2, rA5
mE10 **from** del, in1, as1, mE3, mi6, Ot3, Ha2, Ha3, mi4, mE11, mE9, mE7, rA5
mE11 $\Leftarrow$ mE10, mi13, mE16, mi16, mi5, mE3 , Ne12, Ne13, mC12, mE2, mE1, mE4, mC6, mE12, mi12, Cu13, Cu14, He3, He4, mi4
mE12 $\Leftarrow$ Ne23, Ne22, mE16, He6, Ne8
mE13 **from** Ot3, Ha2, mE14, del, in1, as1, Ha3, mi6, mi4, mE3, rA5
mE14 **from** Ot3, Ha2, del, in1, as1, Ha3, mi6, mi4, nT18, mE3, mE2, rA5
mE15 **from** Ot3, mE1, Ha2, del, in1, as1, Ha3, mi6, mi4, nT18, mE3, mE2, mE7, mE14, mE4, rA5
mE16 **from** Ha3, Ha2, mE3, del, in1, as1, mi6, mE2, mE4, mE1, mE7, mi4, Ot3, mE14, mE13, rA5
pr1 **from** Ot3, rA11, rA10, nT9, nT10, Ne5, mi2, mi4, mi8, mi5
pr2, pr3 **from** pr1, Ot3, rA11, mi1
pr4 $\Leftarrow$ pr1
pr5 $\Leftarrow$ pr6, pr1, bu1
pr6 **from** Ot3, Ha2, nT9, nT10, nT14, nT16, He2, rA2, pr1, bu1, pr10, rA9, He1, rA4
pr7, pr8, pr9 $\Leftarrow$ pr1, mi4
pr10 **from** Ot3, pr1, nT9, nT10, nT14, nT17
bu1 **from** Ot3, rA11, rA10, nT9, nT10, Ne5, mi2, mi8, mi5, bu5
bu2, bu3 $\Leftarrow$ bu1, Ot3, rA10
bu4 $\Leftarrow$ bu1
bu5 **from** Ot3, nT9, nT10, nT16, nT18, pr1, bu1
bu6 $\Leftarrow$ bu1, mi4
Ot1 **from** del, in1, as1
Ot2 **from** del, in1, as1
Ot3 **from** true
Ot4 **from** Ot3

References

[1] Attiya, H., Bar-Noy, A., Dolev, D., Peleg, D. Reischuk, R.: Renaming in an asynchronous environment. J. ACM **37** (1990) 524–548

[2] Bar-Noy, A., Dolev, D.: Shared-memory vs. message-passing in an asynchronous distributed environment. In Proc. 8th ACM Symp. on principles of distributed computing, pp. 307–318, 1989

[3] Cassez, F., Jard, C., Rozoy, B., Dermot, M. (Eds.): Modeling and Verification of Parallel Processes. 4th Summer School, MOVEP 2000, Nantes, France, June 19-23, 2000.

[4] Groote, J.F., Hesselink, W.H., Mauw, S., Vermeulen, R.: An algorithm for the asynchronous write-all problem based on process collision. Distributed Computing **14** (2001) 75–81

[5] Harbison, S.P.: Modula-3, Prentice Hall 1992

[6] Herlihy, M.P.: Wait–free synchronization. ACM Trans. on Program. Languages and Systems **13** (1991) 124–149

[7] Herlihy, M.: A methodology for implementing highly concurrent data objects. ACM Trans. on Programming Languages and Systems bf 15 (1993), 5

[8] Herlihy, M.P. and Moss, J.E.B.: Lock-free garbage collection for multiprocessors. IEEE Transactions on Parallel and Distributed Systems **3** 304–311, 1992

[9] Hesselink, W.H.: Wait–free linearization with a mechanical proof. Distrib Comput **9** (1995) 21–36

[10] Hesselink, W.H.: Bounded Delay for a Free Address. Acta Informatica **33** (1996) 233–254

[11] Hesselink, W.H., Groote, J.F.: Wait-free concurrent memory management by Create, and Read until Deletion (CaRuD). Distributed Computing **14** (2001) 31–39

[12] `http://www.cs.rug.nl/~wim/mechver/hashtable`

[13] Kanellakis, P.C. and Shvartsman, A.A.: *Fault-tolerant parallel computation*. Kluwer Academic Publishers, 1997

[14] Lamport, L.: The temporal logic of actions. ACM Trans. on Programming Languages and Systems **16** (1994) 872–923.

[15] Knuth, D.E.: The Art of Computer Programming. Part 3, Sorting and searching. Addison-Wesley, 1973.

[16] Lynch, N.A.: Distributed Algorithms. Morgan Kaufman, San Francisco, 1996.

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

[18] Owre, S., Shankar, N., Rushby, J.M., Stringer-Calvert, D.W.J.: PVS Version 2.4 (2001). System Guide, Prover Guide, PVS Language Reference. `http://pvs.csl.sri.com`

[19] Valois, J.D.: Lock-free linked lists using compare-and-swap. *Proceedings of the 14th Annual Principles of Distributed Computing*, pages 214-222, 1995. See also J.D. Valois. ERRATA. Lock-free linked lists using compare-and-swap. Unpublished manuscript, 1995

[20] Valois, J.D.: Implementing Lock-Free Queues, *Proceedings of the Seventh International Conference on Parallel and Distributed Computing Systems*, pages. 64-69, Las Vegas, October 1994

[21] Wirth, N.: Algorithms + Data Structures = Programs. Prentice Hall, 1976.