

## Chapter 5

# Lock-free parallel garbage collection by mark&sweep

This chapter concerns our technical report [22].

## 5.1 Introduction

On shared-memory multiprocessors, processes coordinate with each other via shared data structures. To ensure the consistency of these concurrent objects, processes need a mechanism for synchronizing their access. In such a system the programmer typically has to explicitly synchronize access to shared data by different processes to ensure correct behavior of the overall system, using synchronization primitives such as semaphores, monitors, guarded statements, mutex locks, etc. In fact, the operations of different processes on a shared data structure should appear to be serialized so that the object state is kept coherent after each operation.

Due to blocking, the classical synchronization paradigms using locks can incur many problems such as convoying, priority inversion and deadlock. A *lock-free* (also called *non-blocking*) implementation of a shared object guarantees that within a finite number of steps always some process trying to perform an operation on the object will complete its task, independently of the activity and speed of other processes [28]. As lock-free synchronizations are built without locks, they are immune from the aforementioned problems. In addition, lock-free synchronizations can offer progress guarantees. A number of researchers [6, 9, 28, 29, 51, 54] have proposed techniques for designing lock-free implementations. Essential for such implementations are advanced machine instructions such as *load-linked (LL)/store-conditional (SC)*, or *compare-and-swap (CAS)*.

In this chapter we propose a lock-free implementation of mark&sweep *garbage collection (GC)*. Garbage *collectors* are employed to identify at run-time which objects are no longer referenced by the *mutators* (i.e. user programs that use and modify the objects). The heap space occupied by these objects is said to be *garbage* and must be re-cycled for subsequent new objects. The garbage collectors reclaim all garbage by adding them to a so called *free-list*, which keeps track of free memory. Some programming languages (e.g. C, C++) force or allow the programs to do their own GC, which means that programs are required to delete objects that they allocate in memory. However, this task is so difficult that non-trivial applications often exhibit incorrect behavior as the result of memory leaks or dangling pointers. To relieve programmers of virtually all memory-management problems, it is preferable to offer GC that is automatically triggered during memory allocation when the amount of free memory falls below some threshold or after a certain number of allocations.

There are several basic strategies for GC: reference counting [15, 44, 50, 59], mark&sweep

[4, 8, 16, 17, 18] and copying [30, 38, 39, 40, 61]. Reference counting algorithms can do their job incrementally (the entire heap need not be collected at once, resulting in shorter collection pauses), but impose overhead on the mutators and fail to reclaim circular garbage. Mark&sweep algorithms can reclaim circular structure, and don't place any burden on the mutators like reference counting algorithms do, but tend to leave the heap fragmented. Copying algorithms can reduce fragmentation, but add the cost of copying data from one space to another and require twice as much memory as a mark&sweep collector. Moreover, copying also requires that the programming language restricts address manipulation operations, which isn't true for C or C++. For a more detailed introduction to garbage collection and memory management the reader is referred to [43].

One often encounters GC algorithms (e.g. [10, 18, 19, 60, 65]) that employ "stop-the-world" mechanisms, which suspend all normal running threads and then perform GC. Such an algorithm introduces a global synchronization point between all threads and tends to become a scaling bottleneck that limits program performance and processor utilization. In particular, a "stop-the-world" mechanism violates non-blockingness. This is unacceptable when the system must guarantee response time of interactive applications. Therefore, to achieve parallel speed-ups on shared-memory multiprocessors, lock-free algorithms are of interest [46, 67, 69].

There are several lock-free GC algorithms in the literature. The first one is due to Herlihy and Moss [30]. They present a lock-free copying GC algorithm, which uses copying for moving objects to avoid blocking synchronization. In their algorithm, the failure of a participating thread can indefinitely prevent the freeing of unbounded memory. In [38], Hesselink and Groote give a wait-free (wait-freedom is stronger than lock-freedom) GC algorithm using reference counting. However, this collector applies only to a restricted programming model, in which objects are not allowed to be modified between creation and deletion, and is therefore generally limited. Detlefs et al. [15] provide a lock-free GC algorithm using reference counting. The approach relies on a strong hardware primitive, namely *double-compare-and-swap* (*DCAS*) for atomic update of two completely distinct words in memory. Michael [56] presents an efficient lock-free memory management algorithm that does not require special operating system or hardware support. However, his algorithm only guarantees an upper bound on the number of removed nodes not yet freed at any time. This is undesirable because a single garbage node might induce a large amount of occupied resources and might never be reclaimed.

Mark&sweep algorithms do not move objects. They can thus coexist well with C/C++ code, where one never dares to move an object because of possible address computations, and are gaining popularity. Our lock-free mark&sweep algorithm is non-intrusive and features high-performance and reliability. Moreover, unlike most previously published Mark&sweep algorithms [4, 8, 16, 17, 18, 60], we make no assumption on the maximum numbers of mutators and collectors that can operate concurrently at any time. The performance of GC is improved when more processors are involved in it. As far as we could find, no similar algorithm exist.

The correctness properties of any concurrent implementation are seldom easy to verify. This is in general even harder for lock-free algorithms. Our previous work (see chapter 2) shows that providing correctness proofs for such algorithms require huge amounts of effort, time, and skill. In chapters 3 and 4, we have developed two reduction theorems that enable us to reason about a lock-free program to be designed on a higher level than the synchronization primitives *LL/SC* and *CAS*. The reduction theorems are based on refinement mappings as described by Lamport [49], which are used to prove that a lower-level specification correctly implements a higher-level one. Using the reduction theorems, fewer invariants are required and some invariants are easier to discover and formulate without considering the internal structure of the final implementation. In particular, nested loops in the lower-level algorithm may be treated as one loop at a time.

Even so, 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 use the higher-order interactive theorem prover PVS [63] for mechanical support. It is worth noting that there are only a few computer checked correctness proofs of concurrent GC algorithms. In [26], Havelund and Shankar use PVS to verify a safety property of a Mark&sweep GC algorithm, originally suggested by Ben-Ari [8]. In [59], Moreau and Duprat model a distributed reference counting algorithm and prove the safety and liveness property with Coq [14].

## Overview of this chapter

Section 5.2 contains the specification of the garbage collector and the interface offered to the users. A higher-level implementation are presented in Section 5.3. In Section 5.4, the correctness properties are proven. The proof is based on a list of invariants and lemmas, presented in Appendix B.1, while the relationships between the invariants are given by a

dependency graph in Appendix B.2. Section 5.5 gives a low-level lock-free transformation by means of *LL* and *SC* using the reduction theorem (Theorem 3.4.2) developed in chapter 3. The result is given in Appendix B.3. Section 5.6 presents the results of some practical experiments. Conclusions are drawn in Section 5.7.

## 5.2 Specification

We assume a fixed set `Node` of nodes, each of which is identified with a unique label between 1 and  $N$  for some  $N \in \mathbb{N}$ . To specify garbage collection, we introduce a specification variable `free` of type set of nodes to hold the nodes that are available for allocation of new objects by the application processes. The set `free` is filled by the garbage collectors.

The nodes outside `free` form a finite directed graph of varying structure, called the heap, see Fig. 5.1. Each node in the graph points to zero or more children (nodes), and the descendent relation may be circular. In the following context, we regard the attributes of nodes as arrays indexed by  $1 \dots N$ . The number of children of a node  $x$  is indicated by its `arity`, which is denoted by `arity[x]` (instead of `Node[x].arity` as it would be if  $x$  is an index of array `Node` of some record type with a field `arity`). We let  $C$  be the upper bound of the arities of the nodes. The expression `child[x, j]` stands for the pointer to the  $j$ th child of node  $x$ , where  $1 \leq j \leq \text{arity}[x]$ .

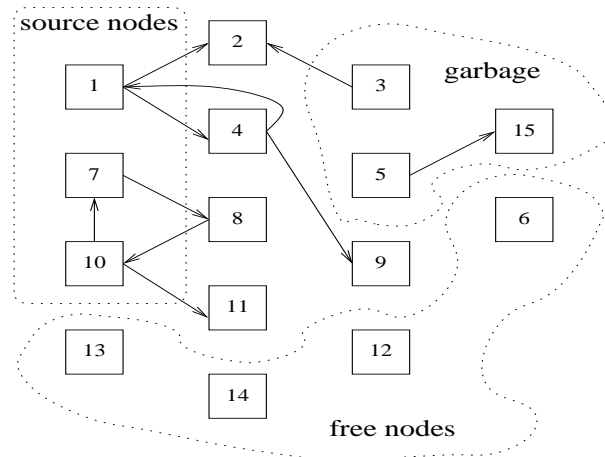


Figure 5.1: A graph representation of the memory

A node is called a *root* when some process has direct read access to it (such as global/static

variables, stack locations and registers of the system). Each application process  $p$  maintains a private set  $roots_p$  that holds its root nodes. The set  $Roots$  is the union of all  $roots_p$  for all processes  $p$ .

Access to nodes can be transferred between processes. We assume that there is a two-dimensional array  $\mathbf{Mbox}$  indexed with a pair of processes that serves as mailboxes. If process  $p$  allows process  $q$  to access some node  $x$ , it writes  $x$  at  $\mathbf{Mbox}[p, q]$  using *Send*. Then, process  $q$  can claim access to node  $x$  by calling *Receive*.

We call a node a *source node* if the node is either in  $Roots$  or in some mailbox. A node is called *accessible* iff it is reachable by following a chain of pointers from a source node. Free nodes must not be accessible. Only nodes in the **free** set are allowed to be allocated by the mutators. A node is said to be a *garbage node* if it is neither accessible nor in the **free** set. Behind “user system” garbage collectors compute the set of nodes reachable from a set of source nodes and reclaim all garbage nodes by placing them into the **free** set. More formally, we define

$$\begin{aligned} R(p, x) &\equiv (\exists z \in roots_p: z \xrightarrow{*} x), \\ R(x) &\equiv (\exists z \in Roots: z \xrightarrow{*} x) \vee (\exists p, q \in \mathbf{Process}: \mathbf{Mbox}[p, q] \xrightarrow{*} x), \end{aligned}$$

where the reachability relation  $\xrightarrow{*}$  is the reflexive transitive closure of relation  $\rightarrow$  on nodes defined by

$$z \rightarrow x \equiv (\exists k: 1 \dots \mathbf{arity}[z]: \mathbf{child}[z, k] = x).$$

According to the definitions, a node  $x$  is accessible iff  $R(x)$  holds. Process  $p$  is said to have access to node  $x$  if  $R(p, x)$  holds. Obviously,  $R(p, x)$  implies  $R(x)$ . The fact that a node  $x$  is a garbage node is formalized by:

$$\neg R(x) \wedge x \notin \mathbf{free}.$$

GC does not modify the memory graph (children or arities of nodes) but only repeatedly adds garbage nodes to the **free** set by executing:

```
proc GCollect()
  ⟨ choose  $x \in \mathbf{Node}$  such that  $\neg R(x) \wedge x \notin \mathbf{free}$ ; free :=  $\mathbf{free} \cup x$ ; ⟩
```

Here, and henceforth, we use angular brackets  $\langle \rangle$  to indicate that embraced statements are (thought to be) executed atomically.

To specify that GC does happen and is eventually exhaustive, we give the progress property of the collectors specified as follows:

$$\neg R(x) \wedge x \notin \mathbf{free} \Rightarrow \diamond(\neg R(x) \wedge x \in \mathbf{free})$$

that is, every garbage node will be eventually put into the **free** set by a garbage collector.

The machine architecture that we have in mind is based on modern shared-memory multiprocessors that can access a common shared address space. There can be several processes running on a single processor. Let us assume there are  $P$  concurrently executing sequential processes along with a set of variables. In the text of a procedure, we use *self* to indicate the process identifier of the process that invokes the procedure. The interface consists of a shared data structure of nodes, and a number of procedures that can be called in the application processes.

We provide procedures that can read and modify the reachable part of the memory graph (from source nodes). An application programmer can assume that the behavior of the routines to access the data is as provided here. In this sense these routines are the specification of our algorithm. In the next section we provide implementable routines with the same behavior as specified here. The specification procedures are *Create*, *AddChild*, *GetChild*, *Make*, *Protect*, *UnProtect*, *Send*, *Receive* and *Check*. We use braces { } to indicate a precondition that must hold when invoking a certain procedure.

```

proc Create(): Node
  local  $x$  : Node;
  ⟨ when available extract  $x$  from free;
    arity[ $x$ ] := 0;  $roots_{self} := roots_{self} \cup \{x\}$ ; ⟩
  return  $x$ ;

proc AddChild( $x$ ,  $y$ : Node): Bool
{  $R(self, x) \wedge R(self, y)$  }
  local  $suc$  : Bool;
  ⟨  $suc := (\mathbf{arity}[x] < C)$ ;
    if  $suc$  then  $\mathbf{arity}[x]++$ ;  $\mathbf{child}[x, \mathbf{arity}[x]] := y$ ; fi ⟩
  return  $suc$ ;

proc GetChild( $x$ : Node,  $rth$ :  $\mathbb{N}$ ): Node  $\cup$  {0}
{  $R(self, x)$  }
  local  $y$  : Node  $\cup$  {0};

```

```

  < if  $1 \leq rth \leq \text{arity}[x]$  then  $y := \text{child}[x, rth]$ ; else  $y := 0$ ; fi >
  return  $y$ ;

```

```

proc Make( $c$ : array [ ] of Node,  $n$ :  $1 \dots C$ ): Node
{  $\forall j: 1 \leq j \leq n: R(\text{self}, c[j])$  }
  local  $x$ : Node;  $j$ :  $\mathbb{N}$ ;
  < when available extract  $x$  from free;
  for  $j := 1$  to  $n$  do  $\text{child}[x, j] := c[j]$  od;
   $\text{arity}[x] := n$ ;  $\text{roots}_{\text{self}} := \text{roots}_{\text{self}} \cup \{x\}$ ; >
  return  $x$ ;

```

```

proc Protect( $x$ : Node)
{  $R(\text{self}, x) \wedge x \notin \text{roots}_{\text{self}}$  }
  <  $\text{roots}_{\text{self}} := \text{roots}_{\text{self}} \cup \{x\}$ ; >
  return;

```

```

proc UnProtect( $z$ : Node)
{  $z \in \text{roots}_{\text{self}}$  }
  <  $\text{roots}_{\text{self}} := \text{roots}_{\text{self}} \setminus \{z\}$ ; >
  return;

```

```

proc Send( $x$ : Node,  $r$ : Process)
{  $R(\text{self}, x) \wedge \text{Mbox}[\text{self}, r] = 0$  }
  <  $\text{Mbox}[\text{self}, r] := x$ ; >
  return;

```

```

proc Receive( $r$ : Process): Node
{  $\text{Mbox}[r, \text{self}] \neq 0$  }
  local  $x$ : Node;
  <  $x := \text{Mbox}[r, \text{self}]$ ;
   $\text{Mbox}[r, \text{self}] := 0$ ;  $\text{roots}_{\text{self}} := \text{roots}_{\text{self}} \cup \{x\}$ ; >
  return  $x$ ;

```

```

proc Check(r, q: Process): Bool
  local suc : Bool;
  ⟨ suc := (Mbox[r, q] = 0); ⟩
  return suc;

```

The application programmers are responsible for ensuring that an offered procedure is called only when its precondition (enclosed by braces if there is any) holds. It is a proof obligation for us that all preconditions of any interface procedure are stable from the perspective of the calling process.

A mutator may continuously allocate a node, add some pointers in the memory, and remove a node from its *roots* set or mailbox. When an allocation request is made, the mutator tries to find a free node (see procedures *Create* and *Make*). The condition “available” in *Create* and *Make* is implementation dependent. When an allocation request cannot be met from the free memory, the mutator either waits, or invokes a new round of GC to free more garbage, or expands the current heap by requesting more memory from the operating system. The threshold value that determines whether or not to invoke a new round of GC can be customized by the user.

The interface is designed in such a way that, when  $R(p, x)$  holds, no other process can falsify  $R(p, x)$ . This means that every process can justify the accessibility of node  $x$  by checking  $R(p, x)$  (via repeatedly reading arities and children of nodes) without worrying about possible interference from other processes. Indeed, no process is able to decrease  $\text{arity}[x]$  or modify  $\text{child}[x, j]$  for  $1 \leq j \leq \text{arity}[x]$  when node  $x$  is accessible (see procedures *AddChild* and *Make*). Instead, the interface only allows to extend the graph by addition of already accessible children. This restriction is stronger than elsewhere, e.g. [8, 45].

The intention of *UnProtect* is that it makes the node and its descendants eligible for garbage collection unless some other process wants to keep them. Via *Send*, *Receive* and *Check*, our algorithm can be used in a distributed system, in which all processors cooperatively traverse the entire data graph by exchanging “messages” to access remote nodes.

### 5.3 A higher-level implementation

The idea behind most GC algorithms in use is to first recursively trace all reachable nodes starting from root nodes, then nodes not reached are considered garbage and can be collected. We present a lock-free implementation that comes close to the classical mark&sweep algorithms. Since we allow to transfer access to nodes between processes via mailboxes, we have strengthened the definition of garbage to non-reachability from source nodes instead of from root nodes.

Atomicity is a semantic correctness condition for concurrent systems. Each process in the implementation is a sequential program comprised of labeled groups of statements. Each group is thought to be executed atomically. Actions on private variables can be freely merged to one of the nearest atomic groups without violating the atomicity restriction.

For simplicity, we first extend the specification to a high-level implementation, where all actions on shared variables are separated into distinct atomic accesses except for some special commands enclosed by angular brackets  $\langle \dots \rangle$ . In order to be able to finally transform the higher-level algorithm into the low-level algorithm using the reduction theorem developed in chapter 3, we require that every labeled atomic group of statements in the higher-level algorithm refer to at most one shared node.

**Notational Conventions.** Recall that there are  $P$  processes with process identifiers ranging from 1 up to  $P$  and  $N$  nodes labeled from 1 up to  $N$ . Unless otherwise specified, we assume that the free variables  $p$ ,  $q$  and  $r$  are universally quantified over process identifiers and the free variables  $w$ ,  $x$ ,  $y$  and  $z$  universally quantified over node labels. Since the same sequential program can be executed by all processes, we adopt the convention that every private variable name can be subscripted by the process identifier. In particular,  $pc_p$  is the program location of process  $p$ . Recall that we regard the attributes of nodes as arrays indexed by  $1 \dots N$ . E.g. we do not write  $\text{Node}[x].f$  but  $f[x]$ . In order to avoid using too many parentheses, we sometimes use indentation to eliminate brackets and define a binding order for some symbols that we frequently use. The following is a list of these symbols, grouped by binding order; the groups are ordered from the highest binding order to the lowest:

all subscripts and superscripts       $()$ ,  $[]$        $\neg$  (not)       $\wedge$  (and)       $\vee$  (or)  
 $\Rightarrow$  (implication),  $\equiv$  (equivalent)       $\forall$ ,  $\exists$

```

Constant
  P = number of processes;
  N = number of nodes;
  C = upper bound of number of children;
Type
  colorType: {white, black, grey};
  nodeType: record =
    arity: N; % number of children
    child: array [1...C] of 1...N; % pointers to children
    color: colorType; % holds the color of the node
    srcnt: N; % reference counter for a source node
    freecnt: N; % dereference counter for a source node
    ari: N; % number of children at the beginning of GC
    father: N ∪ {-1}; % records the parent node GC traverses
    round: N; % the latest round of GC involved in
  end
Shared variables
  Node: array [1...N] of nodeType; % N shared nodes
  Mbox: array [1...P, 1...P] of 0...N; % mailboxes
  shRnd: N; % the version of the current round of GC
Private variables
  roots: a subset of 1...N; % a set of root nodes
  rnd: N; % private copy of "shRnd", initially 0!
  toBeC: a subset of 1...N; % a set of nodes to be checked
Initialization:
  shRnd = 1 ∧ ∀x: 1...N: round[x] = 1;
  all other variables are equal to be the minimal values in their respective domains.

```

Figure 5.2: Data Structure

### 5.3.1 Data Structure

The data structure we use in the higher-level implementation is shown in Fig. 5.2. We define a shared array `Node` with  $N$  elements to represent the memory graph, and a shared two-dimensional array `Mbox` with  $P \times P$  elements to represent mailboxes. Besides fields `arity` and `child`, each node has one of three colors: *white*, *black* and *grey*, which is stored in the field `color`. The `free` set is implemented as a virtual set that contains all *white* nodes. All *black* nodes reachable from a source node are interpreted as accessible nodes, and all other *black* nodes are garbage. *Grey* is a transient color that only occurs during GC.

Since any accessible node must not be freed as garbage, the system needs to keep track of

source nodes that are created by a process and may still be referred to by other processes. For safety, a process is not allowed to inspect another process's private variable such as *roots*. Instead, we introduce a field `srcnt` for each node to count all references (processes and mailboxes) to the node as a source node. Intentionally, we would like to have something like<sup>1</sup>:

$$\text{srcnt}[x] = \#\{\{p \mid x \in \text{roots}_p\}\} + \#\{\{(p, q) \mid \text{Mbox}(p, q) = x\}\}.$$

Therefore, each collector can recognize a source node immediately by checking if its `srcnt` field is positive. We define:

$$R1(x) \equiv (\exists z: \text{srcnt}[z] > 0: z \xrightarrow{*} x),$$

and we have  $R(x) \Rightarrow R1(x)$ . We do not apply other reference counting to the nodes, since manipulating reference counters is slow and may incur expensive overhead with every duplication and deletion of the pointers.

The main difficulty with tracing the memory graph is that the memory structure can dynamically change during GC. In order to solve this problem, we need some coordination between mutators and collectors to take the view of the memory graph, on which all collectors work. To avoid possible interference between mutators and collectors (we will explain this later), the updates of the field `srcnt` of the node in *UnProtect*, upon deletion from the *roots* set, is postponed until the end of GC. We use the field `freecnt` to count the postponed decrementings of `srcnt`. The fields `ari` and `father` contain the number of children a node has at the beginning of GC and the parent node of a node in a tree traversed from a source node by collectors, respectively.

Moreover, since more than one process may participate in GC and they may operate concurrently with mutators, we need to avoid interference from delayed processes. We use a shared variable `shRnd` to hold the round number of the current GC, together with an additional field `round` in the record of a node. The private variable `rnd` is a private copy of the shared variable `shRnd`. A process *p* participates in the current round of GC if and only if  $\text{rnd}_p = \text{shRnd}$ . We introduce the global private variable *toBeC* to transfer information about checked nodes between internal calls. There is also a local private variable *toBeD*.

---

<sup>1</sup>The precise formula is invariant *I5* in Appendix B.1.

### 5.3.2 Algorithm

In this section, we give a higher-level implementation for the collectors and the mutators. Since procedure calls only modify private control data, procedure headers are not always labeled themselves, but their bodies usually have numbered atomic statements. The labels are chosen identical to the labels in the PVS code, and are therefore not completely consecutive.

Brackets `[ ]` and the actions between braces `{ }` and parenthesis `( )` can be ignored in the implementation. They only serve in the proof of correctness. We will explain this in section 5.4.

#### Collectors

Our garbage collectors are encoded in the procedure *GCollect* as shown in Fig. 5.3. It is comprised of three phases: (1) paint all *black* nodes *grey* while recording the current memory structure, (2) paint all *grey* nodes reachable from the source nodes back to *black* after traversing the memory graph, and (3) reclaim all garbage by painting all remaining *grey* nodes *white*. The transitions between the colors are shown in Fig. 5.4.

Processes first let *rnd* get the current value of `shRnd` (this is the only action that updates the private variable *rnd*) to prepare for participating in this round of GC. A new round of GC is started when the fastest process reaches line 101 with  $rnd_{self} = shRnd$  holding in the precondition. It is proved by means of invariants that before a new round of GC is started, all earlier rounds of GC have completed:

$$\forall x: 1 \dots N: \text{round}[x] = \text{shRnd} \wedge \text{color}[x] \neq \text{grey}.$$

In order to prevent some process from doing useless or even harmful work, every modification on a node in each phase is protected by a guard, which can force a process (in particular, a process with  $rnd_{self} \neq shRnd$ ) to abandon its delayed operation.

In the first phase, from label 101 to label 108, processes try to update field `round`, paint *black* nodes *grey* and record the present memory structure using fields `ari` and `father`. The processes only need to paint the *black* nodes *grey* since the *white* nodes can not be garbage.

As the algorithm allows parallel use of mutators, being a source node is not stable during GC. A new source node can be allocated from the `free` set by *Create* or *Make*, or generated by *Protect* or *Send* during GC.

```

proc GCollect() =
  local x: 1...N; toBeD: a subset of 1...N;
% first phase
100: rnd := shRnd; toBeC := {1,...,N};
101: while shRnd = rnd ∧ toBeC ≠ ∅ do
  choose x ∈ toBeC;
108:  ⟨ if round[x] = rnd then
    round[x] := rnd + 1; ari[x] := arity[x]; { outGC[x] := false; }
    if color[x] = black then color[x] := grey; fi;
    if srcnt[x] > 0 then father[x] := 0; else father[x] := -1; fi; fi ⟩
    toBeC := toBeC \ {x};
  od;
% second phase
121: toBeC := {1,...,N}; toBeD := {1,...,N};
122: while shRnd = rnd ∧ toBeD ≠ ∅ do
  choose x ∈ toBeD;
126:  toBeD := toBeD \ {x};
  ⟨ if father[x] = 0 then ⟩
    Mark_stack(x); fi;
  od;
% third phase
129: while shRnd = rnd ∧ toBeC ≠ ∅ do
  choose x ∈ toBeC;
134:  ⟨ if round[x] = rnd + 1 ∧ color[x] = grey then
    color[x] := white;
    ( assert ¬R(x) ∧ x ∉ free; free := free ∪ x; ) fi; ⟩
    toBeC := toBeC \ {x};
  od;
135: ⟨ if rnd = shRnd then shRnd := rnd + 1; { outGC := λ(i:1...N):true; } fi; ⟩
137: return
end GCollect.

```

Figure 5.3: Procedure *GCollect*

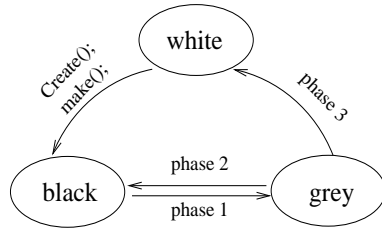


Figure 5.4: Transitions between these colors.

There may be some delay in decrementing field `srcnt` when the number of references decreases (see *UnProtect*). Therefore, we can not say a node  $x$  is a source node if its field `srcnt` is positive. Instead, a node is ever a source node since the latest calibration (this is carried out at the end of the second phase) if its field `srcnt` is positive.

We let the field `father` of each node with positive `srcnt` be 0, and that of other nodes be  $-1$  in the first phase. A new source node  $x$  can then be distinguished from others by checking if  $\text{srcnt}[x] > 0 \wedge \text{father}[x] \neq 0$  holds. For simplicity, we say that a node  $x$  with  $\text{father}[x] = 0$  is an *old source node*. When the fastest process participating in the current GC is at the end of its first phase, all non-free nodes are *grey* except that new source nodes are *black*.

A delayed initialization on node  $x$  will be skipped because of the guard in line 108 since `round[x]` is never decreased. As usual with version numbers, here we need to assume that sufficient bits are allocated for the version numbers to ensure that they cannot “wrap around” during the interval of a process’s GC cycle.

In the second phase, from label 121 to label 126, the processes build a forest in the set of all reachable nodes starting from the old source nodes. Trees in the forest are mutually disjoint. Each of them is rooted by a chosen old source node, and is created via calling a procedure *Mark\_stack* (see Fig. 5.5) in a *while* loop. During *Mark\_stack*, all the *grey* nodes on the tree are painted back to *black* in the order from the leaf to the root.

The procedure *Mark\_stack* is mainly a form of graph search, and it was initially designed as a recursive procedure. Since we want to prove the correctness of our algorithm with PVS, we eliminated the recursion in favor of an explicit stack. The private variable *toBeC* serves to ensure that the search of a collector traverses every node at most once. This is important since the memory graph may have cycles and nodes may be reachable from different old source nodes.

```

proc Mark_stack(x: 1...N) =
  local w, y: 1...N; suc: Bool; j, k: ℕ;
  stack: Stack; head: ℕ; set: a subset of 1...N;
  ch: [1...C] of 1...N;
150: toBeC := toBeC \ {x}; set := {x}; head := 0;
151: while shRnd = rnd ∧ set ≠ ∅ do
  choose w ∈ set;
157: set := set \ {w};
  ⟨ if color[w] = grey ∧ round[w] = rnd + 1 then
    k := ari[w];
    for j := 1 to k do ch[j] := child[w, j] od; ⟩
    head++; stack[head] := w; j := 1;
158: while shRnd = rnd ∧ j ≤ k do
  y := ch[j];
  if y ∉ toBeC then j++;
  else
163:   ⟨ if (father[y] = -1 ∨ father[y] = w)
     ∧ color[y] = grey ∧ round[y] = rnd + 1 then
       father[y] := w; ⟩
       toBeC := toBeC \ {y}; set := set ∪ {y}; fi;
       j++; fi;
  od; fi;
  od;
168: while shRnd = rnd ∧ head ≠ 0 do
  y := stack[head];
175: head--;
  ⟨ if color[y] = grey ∧ round[y] = rnd + 1 then
    color[y] := black;
    srcnt[x] := srcnt[x] - freecnt[x]; freecnt[x] := 0; fi; ⟩
  od;
180: return
end Mark_stack.

```

Figure 5.5: Procedure *Mark\_stack*



stack after their old children have been temporarily stored. The order of the elements pushed on the stack is essential for correctness.

After the tree has been established, the process paints all *grey* nodes *black* in the order in which they are popped from the *stack* (from label 168 to label 175) if the action is not too late. When a node in the tree is painted *black*, its descendants (with respect to the *father* relation) in the tree have been painted *black* already (see Fig. 5.6). So the other processes need not trace or paint the subtree starting from that node. In particular, processes need not trace or paint the tree starting from a new source node. The proof of all this requires interesting and rather complicated graph theoretic invariants. At the end of *Mark\_stack*, the process returns to the procedure *GCollect* to search the tree from another old source node.

Note that it is sufficient to explore all accessible *grey* nodes in the second phase without the help of new source nodes. Using the view of the memory structure taken in the first phase may cause to miss collecting some new garbage that is generated by *UnProtecting* a source node after the first phase, but this does not matter since the new garbage will be recycled within two rounds of GC according to the liveness property (we will come to this later).

All old source nodes appear in the different trees of the forest. The tasks of tracing reachable nodes starting from the different old source nodes can be distributed among several processes. When the fastest process that participates in the current GC is at the end of the second phase, all accessible *grey* nodes have been detected and painted *black*.

In the third phase, from label 129 to the end of the procedure, processes try to re-cycle all remaining *grey* nodes by coloring them *white* (i.e. adding them to the **free** set). The main proof obligation for the algorithm is that all nodes being freed are not accessible. When the fastest process that participates in the current GC is at the end of third phase, the shared variable **shRnd** is incremented to notify all other collectors to quit garbage collecting. We define a round of GC to be completed as soon as the fastest process involved in that round of GC finishes the third phase by incrementing **shRnd**.

It is advantageous that the processes may exchange information. The processes involved in the same round of GC should not use the same strategy for choosing  $x$  in the same phase. For the interested reader, more details can be found in the algorithms for the *write-all-problem* [24, 46]. The main idea is to partition the task statically into many roughly equivalent subtasks (more subtasks than the number of available threads), and then let each

thread dynamically claim one subtask at a time and remove the subtask after completion.

## Mutators

The implementations of the procedures for the mutators are relatively easy. We provide the code in Fig. 5.7 and Fig. 5.8 for the interface procedures in the mutators, which match directly with the procedures in the specification. Note that the mutators do not modify fields `ari`, `father` and `round` of nodes.

Procedure *Create* and *Make* serve to extend the memory graph with a new node. In *Create* and *Make*, “time to do GC” indicates that some variable, like time or the amount of free memory, reaches a threshold value. Allocation in the mutator (see *Create* and *Make*) is potentially expensive. It requires a linear search over the whole memory. This problem can be solved by implementing the free set as a lock-free list (see [55, 69]) with adding a new element to the list in a new numbered line just before the last `fi` in line 134, and deletions of elements from the list in lines 200 and 300.

In procedure *UnProtect*, at line 450, the decrementing of the field `srcnt` of the node is postponed when the process removes the node from its *roots* set. Instead, we use the field `freecnt` to count every delayed *UnProtect*. The immediate incrementation of `srcnt` is incorrect because of the following counterexample. Assume there are three nodes: node 1 is a free node, node 2 is a source node, with one child, node 3. Now, process *p* starts the first phase of GC. Just after process *p* initializes node 1 (a white node), it goes to sleep. Then process *q* is scheduled and *Makes* node 1 a new root node, of which the `color` becomes *black* (instead of *grey*), and sets node 3 as a new child of node 1. Then process *q* *UnProtects* node 2, and node 2 happens to become a non-source node afterwards. Then process *p* wakes up, resumes to initialize node 2 and node 3. Since in the second phase, processes only explore all *grey* nodes reachable from old source nodes, processes will regard node 3 as an inaccessible node and collect it mistakenly as garbage in the third phase.

One may wonder why the decrementing of the field `srcnt` is postponed from *UnProtect* to line 175 of *Mark\_stack*. We tried to update fields `srcnt` in the first phase of GC. However, we found that this is not correct while we proceeded the mechanical proof with PVS. The counterexample is the same as the previous one. After inspecting some invariants, we found that all accessible *grey* nodes can be traced without the help of either the *black* nodes or the upper *grey* nodes resided in the local stack. This means that it is safe to update the field `srcnt` at that moment. Moreover, fields `srcnt` of all remaining *grey* nodes appearing

```

proc Create(): 1...N =
  local x: 1...N;
  while true do
200:   choose x ∈ 1...N;
206:   ⟨ if color[x] = white then
        color[x] := black; srcnt[x] := 1;
        ⟨ assert x ∈ free; free := free \ x; ⟩
        [ [ arity[x] := 0; roots := roots ∪ {x}; ] ]
        break;
208:   elseif time to do GC then
        GCcollect(); fi;
      od;
210: [ return x ]
end Create.

proc AddChild(x, y: 1...N): Bool =
{ R(self, x) ∧ R(self, y) }
  local suc: Bool;
258: ⟨ [ [ suc := (arity[x] < C);
        if suc then arity[x]++; child[x, arity[x]] := y; fi ] ] ⟩
262: [ return suc ]
end AddChild.

proc GetChild(x: 1...N, rth: 1...N): 0...N =
{ R(self, x) }
  local y: 0...N;
280: ⟨ [ [ if 1 ≤ rth ≤ arity[x] then y := child[x, rth]; else y := 0; fi ] ] ⟩
284: [ return y ]
end GetChild.

proc Make(c: array [ ] of 1...N, n: 1...C): 1...N =
{ ∀ j: 1...n: R(self, c[j]) }
  local x: 1...N; j: ℕ;
  while true do
300:   choose x ∈ [1...N];
306:   ⟨ if color[x] = white then
        color[x] := black; srcnt[x] := 1;
        ⟨ assert x ∈ free; free := free \ x; ⟩
        [ for j := 1 to n do child[x, j] := c[j]; od
          arity[x] := n; roots := roots ∪ {x}; ] ]
        break;
308:   elseif time to do GC then
        GCcollect(); fi;
      od;
310: [ return x ]
end Make.

```

Figure 5.7: Procedure *Create*, *AddChild*, *GetChild* and *Make*

```

proc Protect( $x: 1 \dots N$ ) =
{  $R(\text{self}, x) \wedge x \notin \text{roots}$  }
400:  $\langle \text{srcnt}[x]++; \rangle$ 
     $\llbracket \text{roots} := \text{roots} \cup \{x\}; \rrbracket$ 
408:  $\llbracket \text{return} \rrbracket$ 
end Protect.

proc UnProtect( $z: 1 \dots N$ ) =
{  $z \in \text{roots}$  }
450:  $\langle \text{freecnt}[z]++; \rangle$ 
     $\llbracket \text{roots} := \text{roots} \setminus \{z\}; \rrbracket$ 
460:  $\llbracket \text{return} \rrbracket$ 
end UnProtect.

proc Send( $x: 1 \dots N, r: 1 \dots P$ ) =
{  $R(\text{self}, x) \wedge \text{Mbox}[\text{self}, r] = 0$  }
500:  $\langle \text{srcnt}[x]++; \rangle$ 
508:  $\llbracket \text{Mbox}[\text{self}, r] := x; \rrbracket$ 
510:  $\llbracket \text{return} \rrbracket$ 
end Send.

proc Receive( $r: 1 \dots P$ ):  $1 \dots N$  =
{  $\text{Mbox}[r, \text{self}] \neq 0$  }
    local  $x: 1 \dots N$ ;
550:  $\llbracket x := \text{Mbox}[r, \text{self}]; \rrbracket$ 
552: if  $x \notin \text{roots}$  then
     $\llbracket \text{Mbox}[r, \text{self}] := 0; \text{roots} := \text{roots} \cup \{x\}; \rrbracket$ 
    else
558:  $\langle \text{srcnt}[x]--; \rangle$ 
559:  $\llbracket \text{Mbox}[r, \text{self}] := 0; \rrbracket$  ( $\text{assert } x \in \text{roots};$ ) fi;
560:  $\llbracket \text{return} \rrbracket$ 
end Receive.

proc Check( $r, q: 1 \dots P$ ): Bool
    local  $\text{suc} : \text{Bool}$ ;
600:  $\llbracket \text{suc} := (\text{Mbox}[r, q] = 0); \rrbracket$ 
602:  $\llbracket \text{return} \text{suc} \rrbracket$ 
end Check.

```

Figure 5.8: Procedure *Protect*, *UnProtect*, *Send*, *Receive* and *Check*

in the third phase are all zero and therefore need not be decremented.

In procedure *Send* and *Receive*, the weaker requirement on the reference counter (i.e. field `srcnt` of a node) is based on the fact that the reference counter does not always need to be accurate.

## 5.4 Correctness

The main issue of the algorithm is how to ensure the correct execution of collectors and mutators when they concurrently compete with each other for the same data structure. The standard notion of correctness for asynchronous parallel algorithms is to assume that the atomic instructions of the threads are interleaved in an arbitrary linear order. The algorithm is correct if it behaves properly for all such interleavings. Any property can be considered as the conjunction of safety properties and liveness properties. In this section we describe the proofs of safety properties and a liveness property of the algorithm by means of invariants.

### 5.4.1 The main loop

In order to verify our memory management system, we model the clients as a very non-deterministic environment in the following loop that may call the interface procedures in any arbitrary order and with arbitrary arguments provided the preconditions are met. This is not part of the memory management system itself, and therefore not to be implemented. It is provided since it is used in the PVS proof to verify the correctness of the system under all possible applications, in the same way as, e.g. in [37] section 4.2.

#### loop

- 1:     **choose** *call*; **case** *call* **of**
  - (*C*)  $\rightarrow$  *Create*();
  - (*A*, *x*, *y*) **with**  $R(\text{self}, x) \wedge R(\text{self}, y) \rightarrow$  *AddChild*(*x*, *y*);
  - (*G*, *x*, *rth*) **with**  $R(\text{self}, x) \rightarrow$  *GetChild*(*x*, *rth*);
  - (*M*, *c*, *n*) **with**  $\forall j: 1 \dots n: R(\text{self}, c[j]) \rightarrow$  *Make*(*c*, *n*);
  - (*P*, *x*) **with**  $R(\text{self}, x) \wedge x \notin \text{roots}_{\text{self}} \rightarrow$  *Protect*(*x*);
  - (*U*, *z*) **with**  $z \in \text{roots}_{\text{self}} \rightarrow$  *UnProtect*(*z*);
  - (*S*, *x*, *r*) **with**  $R(\text{self}, x) \wedge \text{Mbox}(\text{self}, r) = 0 \rightarrow$  *Send*(*x*, *r*);
  - (*R*, *r*) **with**  $\text{Mbox}(r, \text{self}) \neq 0 \rightarrow$  *Receive*(*r*);

```

    (C, r, q) → Check(r, q);
  end
end

```

Normally, after some operation is finished, the process will return to the main loop. In the implementation, there are several places where the same procedure (e.g. *GCollect*) is called. We introduce an auxiliary private variable *return* to hold the return location. Since they are private, they can be assumed to be touched instantaneously without violation of the atomicity restriction.

### 5.4.2 Safety properties

The main aspect of safety is functional correctness and atomicity, say in the sense of [52]. We prove partial correctness of the implementation by showing that each procedure of the implementation executes its specification command always exactly once and that the resulting value of the implementation equals the resulting value in the specification. As shown in Fig. 5.3 to Fig. 5.8, we therefore extend the implementations with auxiliary variables and commands used in the specification. For simplicity, we use brackets  $\llbracket \ \rrbracket$  to enclose the specification commands that perform the same actions as the implementation, and parenthesis  $( \ )$  to enclose the specification commands that can be deleted in the implementation.

GC is an internal affair not relevant for the users of the routines. *GCollect* cannot be invoked explicitly, but will only be invoked implicitly in for instance *Make* and *Create*. This means we only need to prove the match of the specifications and implementations for all user programs, but not for *GCollect*. Instead, the main safety property we have proved for *GCollect* is that the system only collects garbage, i.e. that an accessible node is never freed. This is expressed in the invariant:

*I1:*  $\text{color}[x] = \text{white} \Rightarrow \neg R(x)$ .

To facilitate the proof of correctness, in *GCollect* we introduce an auxiliary variable *outGC* to indicate whether a node is not involved in the current round of GC. All operations on *outGC* are enclosed in braces  $\{ \}$ , and can be assumed to be executed instantaneously without violation of the atomicity restriction.

We note that the implementation is an extension of the specification except that the set **free** in the specification is implemented as a virtual set of all *white* nodes in the implementation. Apart from the common actions enclosed in  $\llbracket \ \rrbracket$ , all implementation commands

do not modify the specification variables and all specification commands do not modify the implementation variables. Therefore for simplicity, we do not distinguish the identical variables and commands used in the specification and the implementation, and enclose them in  $\llbracket \ \rrbracket$ . Functional correctness of the mutator now becomes obvious since we have proved the following invariants:

- I2:  $\text{color}[x] = \text{white} \equiv x \in \text{free}$   
 I3:  $554 \leq pc_p \leq 559 \Rightarrow x_p \in \text{roots}_p$

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 [49] or abstraction from  $\tau$  steps in process algebras.

Furthermore, we also need to prove that all preconditions of the interface procedures are stable under the actions of the other processes. For the interface procedures *AddChild*, *GetChild*, *Make*, *Protect*, *Send* and *Receive*, the stability of the precondition is expressed respectively by the following invariants:

- I6:  $250 \leq pc_p \leq 258 \Rightarrow R(p, x_p) \wedge R(p, y_p)$   
 I7:  $pc_p = 280 \Rightarrow R(p, x_p)$   
 I8:  $300 \leq pc_p \leq 308 \vee (100 \leq pc_p \leq 180 \wedge \text{return}_p = 300) \Rightarrow \forall k: 1 \dots n_p: R(p, c_p[k])$   
 I9:  $pc_p = 400 \vee (500 \leq pc_p \leq 508) \Rightarrow R(p, x_p)$   
 I10:  $500 \leq pc_p \leq 508 \Rightarrow \text{Mbox}[p, r_p] = 0$   
 I11:  $550 \leq pc_p \leq 559 \Rightarrow \text{Mbox}[r_p, p] \neq 0$

Process  $p$  can ensure its rights to have access to node  $x$  by checking the predicate  $R(p, x)$ , independently, because of the following lemma that asserts  $R(p, x)$  can only be invalidated by process  $p$  itself:

- V1:  $p \neq q \wedge R(p, x) \wedge I18 \wedge I25 \triangleright_q R(p, x)$ ,

where we write  $P \triangleright_q Q$  to express that, if precondition  $P$  holds and process  $q$  performs an atomic action, this action has postcondition  $Q$ .

As we announced earlier, no node is grey when the current round of GC is finished. This is formalized in the following invariant:

- I4:  $\neg(\exists p: \text{rnd}_p = \text{shRnd}) \Rightarrow \text{color}[x] \neq \text{grey}$

where the predicate  $rnd_p = \text{shRnd}$  indicates that process  $p$  is involved in the current round of GC.

The difference  $\text{srcnt}[x] - \text{freecnt}[x]$  of node  $x$  counts the number of references to the source node. Since an atomic group of statements in the higher-level implementation must not refer different shared variables (this is an important requirement for the final lock-free transformation), the fields of a node and a mailbox can not be simultaneously modified in the same atomic group. The counter is precisely described by the following invariant:

$$I5: \quad \text{srcnt}[x] - \text{freecnt}[x] = \#\{\{p \mid x \in \text{roots}_p\}\} + \#\{\{(p, q) \mid (\text{Mbox}(p, q) = x \\ \wedge \neg(\text{pc}_q = 559 \wedge p = r_q)) \vee (\text{pc}_p = 508 \wedge x_p = x \wedge q = r_p)\}\}$$

All the safety properties (invariants) have been proved with the interactive proof checker 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 complete the proof, we actually found some mistakes and had to correct previous versions of this algorithm. To prove these invariants, we need many other invariants. All proved invariants and lemmas are listed in Appendix B.1. Appendix B.2 gives the dependencies between the invariants. For the complete mechanical proof, we refer the interested reader to [32].

### 5.4.3 Liveness

A liveness property asserts that program execution eventually reaches some desirable state. In our case, we want to ensure that every garbage node is eventually collected. We shall express this by means of the “leads-to” (denoted as  $o \rightarrow$ ) relation that was developed for UNITY in [12]. For assertions  $P$  and  $Q$ , we use the formal definition:

$$(P \ o \rightarrow \ Q) \equiv \Box(P \Rightarrow \Diamond Q).$$

We write  $P \triangleright Q$  to express that, if  $P$  holds in the precondition of an atomic action,  $Q$  holds in the postcondition. Moreover, we define:

$$(P \ \mathcal{U} \ Q) \equiv P \triangleright (P \vee Q).$$

All these new symbols are defined to have the same binding order as “ $\Rightarrow$ ” (implication).

The liveness property of the algorithm we need to verify is that, it is always the case that every garbage node is eventually collected. That is,

$$\neg R(x) \ o \rightarrow \ \text{color}[x] = \text{white}.$$

### General Lemmas

In order to prove the liveness property of the algorithm, we establish the needed techniques. First, we introduce fairness into our formalism. This can be done with a single rule: *atomic actions always terminate*. This means that if some process is at the label of some atomic action, this process will eventually execute the action and arrive at the label of the next atomic action. GC is infinitely often triggered during memory allocation when the amount of free memory falls below some threshold or after a certain number of allocations. We therefore need one more fairness assumption, namely GC is eventually called:  $\Box(\Diamond(\exists p : pc_p = 100))$ .

Except some well-known lemmas extracted from the literatures, all lemmas in this section have been verified mechanically with PVS. The following lemmas are stated in [62].

**Lemma 5.4.1** *For assertions  $P$ ,  $Q$ ,  $R$  and  $I$ ,*

- (a) *Relation  $o \rightarrow$  is reflexive and transitive.*
- (b) *If  $P \Rightarrow Q$  then  $P o \rightarrow Q$ .*
- (c) *If  $P o \rightarrow R$  and  $Q o \rightarrow R$  then  $(P \vee Q) o \rightarrow R$ .*
- (d) *If  $(P \wedge \Box Q) o \rightarrow R$  then  $(P \wedge \Box Q) o \rightarrow (R \wedge \Box Q)$ .*
- (e)  *$P o \rightarrow (Q \vee (P \wedge \Box \neg Q))$ .*

Lemma 5.4.1 (a), (b) and (c) are used to prove a general *proof lattice* for a program, which is addressed in [62]. Lemma 5.4.1 (d) shows that in every behavior every state where an invariant holds is followed by states where the invariant always hold. Intuitively, Lemma 5.4.1 (e) is true because starting from a time where  $P$  is true, either  $Q$  will be true at some subsequent time, or  $\neg Q$  will be always true from then on. Thus, the general pattern of these proofs by contradiction is to assume that the desired predicate never becomes true, and then show that this assumption leads to a contradiction. For more details, refer to [62].

We found the “steps-to” ( $\triangleright$ ) relation and the “unless” ( $\mathcal{U}$ ) relation are quite useful to prove the “leads-to” ( $o \rightarrow$ ) relation. Since these two relations only involve a single step, they can be checked directly by PVS with the help of invariants. It is not hard to prove the following general lemmas, which are postulated during the proof of the liveness property.

**Lemma 5.4.2** *For assertions  $P$ ,  $Q$ ,  $R$  and  $S$ ,*

- (a) *If  $P$  and  $(P o \rightarrow Q)$  then  $\Diamond Q$ .*

- (a)  $A_0(m) \wedge \neg R1(x) \wedge \text{black}(x) \wedge \text{round}[x] = m$   
 $\mathcal{U} A_1(m) \wedge \neg R1(x) \wedge \text{black}(x) \wedge \text{round}[x] = m$   
 $\mathcal{U} A_1(m) \wedge \neg R1(x) \wedge \text{grey}(x) \wedge \text{round}[x] = m + 1$   
 $\mathcal{U} A_2(m) \wedge \neg R1(x) \wedge \text{grey}(x) \wedge \text{round}[x] = m + 1$   
 $\mathcal{U} A_3(m) \wedge \neg R1(x) \wedge \text{grey}(x) \wedge \text{round}[x] = m + 1$   
 $\mathcal{U} A_3(m) \wedge \text{white}(x)$
- (b)  $\text{shRnd} = m = \text{round}[x] - 1 \wedge \neg R1(x) \wedge \neg \text{white}(x)$   
 $\mathcal{U} (\text{shRnd} = m \wedge \text{white}(x)) \vee (\text{shRnd} = m + 1 = \text{round}[x] \wedge \neg R1(x) \wedge \neg \text{white}(x))$
- (c)  $A_0(m) \wedge \neg R(x) \wedge \neg \text{white}(x) \wedge \neg R(w) \wedge \text{black}(w) \wedge \text{srcnt}(w) > 0 \wedge \text{round}[w] = m$   
 $\mathcal{U} A_1(m) \wedge \neg R(x) \wedge \neg \text{white}(x) \wedge \neg R(w) \wedge \text{black}(w) \wedge \text{srcnt}(w) > 0 \wedge \text{round}[w] = m$   
 $\mathcal{U} A_1(m) \wedge \neg R(x) \wedge \neg \text{white}(x) \wedge \neg R(w) \wedge \text{grey}(w) \wedge \text{srcnt}(w) > 0 \wedge \text{round}[w] = m + 1$   
 $\mathcal{U} A_2(m) \wedge \neg R(x) \wedge \neg \text{white}(x) \wedge \neg R(w) \wedge \text{grey}(w) \wedge \text{srcnt}(w) > 0 \wedge \text{round}[w] = m + 1$   
 $\mathcal{U} A_2(m) \wedge \neg R(x) \wedge \neg \text{white}(x) \wedge \text{srcnt}(w) = 0$
- (d)  $\text{shRnd} = m = \text{round}[w] - 1 \wedge \neg R(x) \wedge \neg \text{white}(x) \wedge \neg R(w) \wedge \text{srcnt}(w) > 0$   
 $\mathcal{U} (\text{shRnd} = m \wedge \text{white}(x)) \vee (\text{shRnd} = m \wedge \neg R(x) \wedge \neg \text{white}(x) \wedge \text{srcnt}(w) = 0)$   
 $\vee (\text{shRnd} = m + 1 = \text{round}[w] \wedge \neg R(x) \wedge \neg \text{white}(x) \wedge \neg R(w) \wedge \text{srcnt}(w) > 0)$
- (e)  $\text{shRnd} \leq m \wedge \neg R(x) \wedge \neg \text{white}(x) \wedge \neg (\text{srcnt}(w) > 0 \wedge (w \xrightarrow{*} x))$   
 $\mathcal{U} (\text{shRnd} \leq m \wedge \text{white}(x)) \vee \text{shRnd} > m$
- (f)  $\text{shRnd} > m \triangleright \text{shRnd} > m$

Using Lemmas 5.4.1, 5.4.2, 5.4.3 and 5.4.4, we prove the following corollaries.

**Corollary 5.4.1** *For any integer  $m$ ,*

$$\text{shRnd} = m \quad o \rightarrow \quad A_4(m)$$

**Proof:** Inspired by Lemma 5.4.1(e), we assume  $\square \neg A_4(m)$ . Since the shared variable  $\text{shRnd}$  can be modified only by some process executing line 135 with precondition  $\text{rnd}_{\text{self}} = \text{shRnd}$ , we then obtain that  $\text{shRnd}$  is constant, i.e.  $m$ . GC is infinitely often triggered during memory allocation when the amount of free memory falls below some threshold or after a certain number of allocations. We therefore assume that there will eventually exist some process  $p$  such that  $pc_p = 100$ . Because of the fairness of atomic actions, we then get  $\diamond(\text{rnd}_p = \text{shRnd} = m \wedge pc_p = 101)$ . Since  $\text{toBeC}$  and  $\text{toBeD}$  are both private variables, all loops in GC are finite (see procedures  $G\text{Collect}$  and  $\text{Mark\_stack}$ ). We therefore obtain  $\diamond(\text{rnd}_p = \text{shRnd} = m \wedge pc_p = 135)$  according to the fairness. This leads to a contradiction.  $\square$

**Corollary 5.4.2** For any integer  $m$ ,

$$shRnd = m \quad o \rightarrow \quad shRnd = m + 1$$

**Proof:** By Lemma 5.4.2(a) and Corollary 5.4.1, we obtain  $\diamond A_4(m)$ . Since the shared variable  $shRnd$  can be modified only by some process executing line 135 with precondition  $rnd_{self} = shRnd$ , we then obtain that  $shRnd$  will be eventually incremented by 1 according to the fairness.  $\square$

**Corollary 5.4.3** In Lemma 5.4.4, all “unless”( $\mathcal{U}$ ) relations can be replaced by “leads-to”( $o \rightarrow$ ) relations.

**Proof:** By Lemma 5.4.2(a) and Corollary 5.4.2, we obtain  $\diamond(shRnd \neq m)$ . Therefore, this corollary is an obvious consequence of using Lemma 5.4.2(b).  $\square$

**Corollary 5.4.4** For any integer  $m$ ,

$$shRnd = m = \mathit{round}[x] \wedge \neg R1(x) \wedge \neg \mathit{white}(x) \quad o \rightarrow \quad shRnd = m \wedge \mathit{white}(x).$$

**Proof:** By invariant *I16*, we obtain  $\mathit{black}(x)$ . By invariant *I34*, we have  $A_0(m) \vee A1(m)$ . Using transitivity of “leads-to” relation, this is an obvious consequence of Lemma 5.4.4(a) and Corollary 5.4.3.  $\square$

**Corollary 5.4.5** For any integer  $m$ ,

$$shRnd = m \wedge \neg R(x) \wedge \neg R1(x) \wedge \neg \mathit{white}(x) \quad o \rightarrow \quad shRnd \leq m + 1 \wedge \mathit{white}(x).$$

**Proof:** By invariant *I13*, we know  $\mathit{round}[x] = m \vee \mathit{round}[x] = m + 1$ . Therefore, this corollary follows from Corollary 5.4.4, Lemma 5.4.4(b) and Corollary 5.4.3.  $\square$

**Corollary 5.4.6** For any integer  $m$ ,

$$\begin{aligned} shRnd = m = \mathit{round}[w] \wedge \neg R(x) \wedge \neg \mathit{white}(x) \wedge \neg R(w) \wedge \mathit{srcnt}(w) > 0 \quad o \rightarrow \\ shRnd = m \wedge \neg R(x) \wedge \neg \mathit{white}(x) \wedge \mathit{srcnt}(w) = 0 \end{aligned}$$

**Proof:** Obviously, we have  $R1(w)$ . By invariants *I16* and *I18*, we then obtain  $\mathit{black}(w)$ . By invariant *I34*, we have  $A_0(m) \vee A1(m)$ . Using transitivity of “leads-to” relation, this is an obvious consequence of Lemma 5.4.4(c) and Corollary 5.4.3.  $\square$

**Corollary 5.4.7** For any integer  $m$ ,

$$\begin{aligned} shRnd = m \wedge \neg R(x) \wedge \neg white(x) \quad o \rightarrow \quad & (shRnd = m \wedge white(x)) \\ \vee (shRnd \leq m + 1 \wedge \neg R(x) \wedge \neg white(x) \wedge \neg(\text{srcnt}(w) > 0 \wedge (w \xrightarrow{*} x))) \end{aligned}$$

**Proof:** It holds obviously if  $\neg(\text{srcnt}(w) > 0 \wedge (w \xrightarrow{*} x))$  is true. Otherwise, by definition of relation  $R$  and transitivity of “ $\xrightarrow{*}$ ”, we obtain  $\neg R(w)$ . By invariant *I13*, we obtain  $\text{round}[w] = m \vee \text{round}[w] = m + 1$ . Then, it follows from Corollary 5.4.6, Lemma 5.4.4(d) and Corollary 5.4.3.  $\square$

**Corollary 5.4.8** *For any integer  $m$ ,*

$$\begin{aligned} shRnd = m \wedge \neg R(x) \wedge \neg white(x) \quad o \rightarrow \quad & (shRnd \leq m + 1 \wedge white(x)) \\ \vee (shRnd \leq m + 1 \wedge \neg R(x) \wedge \neg white(x) \wedge \neg R1(x)). \end{aligned}$$

**Proof:** Intending to use Lemma 5.4.3 with substitutions:  $1 \dots N$  for  $I$ ;  $\neg(\text{srcnt}(w) > 0 \wedge (w \xrightarrow{*} x))$  for  $Q(w)$ ;  $shRnd = m \wedge \neg R(x) \wedge \neg white(x)$  for  $P$ ;  $shRnd \leq m + 1$  for  $S$ ;  $\neg R(x) \wedge \neg white(x)$  for  $R$ ;  $shRnd \leq m + 1 \wedge white(x)$  for  $T$ , it remains to check the premises. The first premise is true because of Corollary 5.4.7 and Lemma 5.4.4(e). The second premise is true because of Lemma 5.4.4(f).  $\square$

**Corollary 5.4.9** *For any integer  $m$ ,*

$$shRnd = m \wedge \neg R(x) \wedge \neg white(x) \quad o \rightarrow \quad shRnd \leq m + 2 \wedge white(x).$$

**Proof:** This is an obvious consequence of Lemma 5.4.1, Corollaries 5.4.5 and 5.4.8.  $\square$

Theorem 5.4.1 follows immediately from Lemma 5.4.1 and Corollary 5.4.9.

## 5.5 The low-level lock-free implementation

Refinement mappings enable us to reduce an implementation by reducing its components in relative isolation, and then gluing the reductions together with the same structure as the implementation. Atomicity guarantees that a parallel execution of a program gives the same results as a sequential and non-deterministic execution. This allows us to use the refinement calculus for stepwise refinement of transition systems [5].

In chapter 3, we formalize Herlihy’s methodology [28] for transferring a sequential implementation of any data structure into a lock-free synchronization using synchronization primitives *LL/SC*, and develop a reduction theorem (Theorem 3.4.2) that enables us to

reason about a general lock-free algorithm to be designed on a higher level than the synchronization primitives *LL/SC*. Theorem 3.4.2 can be universally employed for a lock-free construction to synchronize access to shared nodes of `nodeType`, and be sure that we end up with the reduction of the implementation. This allows us to design and verify a lock-free program on a higher level than the synchronization primitives. The big advantage is that substantial pieces of the concrete program can be dealt with as atomic statements on the higher level and thus the correctness can be more easily verified.

In the higher-level implementation (from Fig. 5.3 to Fig. 5.8), instruction 135 is simply a *CAS* instruction offered by machine architectures or a Read/Write cycle that can easily be implemented by an *LL/SC*. All other special commands enclosed by angular brackets `<...>` only refer to one shared node and some private variables, and therefore can be transformed into low-level lock-free implementations using Theorem 3.4.2. E.g. line 108 of Fig. 5.3 is implemented in lines 104... 107 of Appendix B.3.2, where `round[mp]` is an alias of `Node[mp].round`, and similarly for `color[mp]`, `srcnt[mp]` and `father[mp]`. At lines 126 and 157 (and possibly other cases), since these commands do not modify the node, swapping of pointers is unnecessary. We therefore use a simplified version where *SC* can be replaced by *VL*. After the transformation, all statements in the algorithm are atomic<sup>2</sup>. The transformation is straightforward, and we present our final lock-free algorithm in Appendix B.3.

Apart from that, the higher-level algorithm can also be transformed into a lock-free implementation by means of *CAS* using the reduction theorem (Theorem 4.4.1) developed in chapter 4. This final transformation is a bit more complicated. Because of the similarity, we don't provide the final transformation here.

## 5.6 Practical experiment

We carried out a number of experiments with our algorithm in order to obtain some insight in its practical performance. The first major conclusion is that the performance of the algorithm is strongly influenced by its parameter setting such as the total number of nodes, the condition for joining the garbage collection process, the percentage of occupied nodes and the division of work between garbage collecting and manipulating the data structure. A

---

<sup>2</sup>Note that accesses to "private" nodes do not violate the atomicity restriction and can be freely added to an atomic statement.

second conclusion is that if we set these parameters well, we see no degrading in performance when increasing the number of processes. A third conclusion is that due to the fact that garbage collection is a relatively elaborate affair, the performance in terms of the number of nodes that are created and collected per unit of time is relatively low.

#Processes	1 processor		2 processors		4 processors	
	0	100	0	100	0	100
1	833k	125k	800k	87k	714k	77k
2	800k	118k	784k	200k	750k	136k
3	789k	115k	741k	207k	794k	207k
4	800k	114k	727k	205k	800k	216k
5	794k	114k	758k	204k	800k	233k
10	833k	112k	870k	222k	851k	263k
15	789k	115k	833k	214k	845k	261k
20	769k	118k	769k	211k	800k	258k
25	781k	114k	735k	208k	840k	256k
31	756k	115k	729k	214k	844k	253k

Table 5.1: Some experimental results

In our experimental setup, we let a number of processes repeatedly create a node, read it a number of times and release it again. One process is the garbage collector process, and the settings of parameters are such that no other process will join in to assist this process. However, if a process fails too often to obtain a free node, the process yields its processor, putting itself in the processor waiting queue. This provides an effective load balancing policy. Note that in order to maintain the lock-free nature of the algorithm this process must eventually join garbage collection if obtaining free nodes fails continuously.

More concretely, in the experiments reported in table 5.6, we use a small array of 2000 nodes and let each process create a large ( $> 10^6$ ) number of nodes. Each processor that must create a node tries 15 times to find a free node before yielding its processor.

The experiments have been carried out on a one, two and four processor machine. All machines were Intel Linux machines of the following types:

- The single processor uses a Pentium III (Coppermine) processors of 1Ghz with 256kb cache each.
- The two processor machine contained two Xeon CPUs of 2.8Ghz with 512kb of cache each.

- The four processor machine has four Intel Xeon CPUs of 2.4 Ghz with 512 KB cache each.

The load linked, store conditional and verify link statements have been implemented using the 64 bit compare and swap (`cmpxchg8B`) instruction available on Intel Pentium processors (see [58] for the implementation). This limits parallelism to 32 processes, but does not have problems with wrap around as the implementation in [57].

The table provides the number of nodes that could be created and read per second. The letter ‘k’ indicates that the figures refer to thousands of nodes. In the columns marked with 0 these nodes are read 0 times, and in the columns marked with 100 these nodes are read a 100 times. The column headed with `#Processes` indicates the number of processes that were used to create new nodes. As stated above there is one additional process doing garbage collecting.

Note that the table shows an almost perfect linear scaling. The variations in the table can fully be contributed to statistical noise. Only when there are few processes on a multiprocessor machine performance is bad. This is due to the fact that the garbage collector has plenty of time compared to the heavily loaded processes – which must read often – and therefore wastes its time. Note also that on multiprocessor machines the performance on generating nodes is low compared to the relatively slow single processor machine. We believe that this is due to interprocess communication induced by compare and swap.

## 5.7 Conclusions

We present a lock-free parallel algorithm for mark&sweep GC in a realistic model by means of synchronization primitives *load-linked (LL)/store-conditional (SC)* or *CAS* offered by machine architectures. Our algorithm allows to collect a circular data structure and makes no assumption on the maximum number of mutators and collectors that can operate concurrently during GC. The efficiency of GC can be enhanced when more processors are involved in it. Providing *Send* and *Receive*, our algorithm can be adapted to a distributed system, in which all processors cooperatively traverse the entire data graph by exchanging “messages” to access remote nodes.

Formal verification is desirable because there could be subtle bugs as the complexity of algorithms increases. To ensure our correctness proof presented in this chapter is not flawed, we use the higher-order interactive theorem prover PVS for mechanical support.

(b) If  $P \mathcal{U} Q$  and  $\diamond \neg P$  then  $P \circ \rightarrow Q$ .

**Lemma 5.4.3** *Let  $Q(w)$  be assertions for all  $w \in I$ , where  $I$  is a finite set. Let  $P, R, S$  and  $T$  are assertions with  $\forall w: I: (P \circ \rightarrow T \vee (S \wedge R \wedge Q(w))) \wedge (S \wedge R \wedge Q(w) \mathcal{U} T \vee \neg S)$  and  $\neg S \triangleright \neg S$ . Then  $P \circ \rightarrow T \vee (S \wedge R \wedge \forall w: I: Q(w))$ .*

### Main Theorems

Actually, we prove something stronger, viz., that, every inaccessible node is painted *white* within two rounds of GC.

**Theorem 5.4.1** *For any integer  $m$ ,  $shRnd = m \wedge \neg R(x) \circ \rightarrow shRnd \leq m + 2 \wedge white(x)$ .*

An invariant has the form of  $Q \Rightarrow \square Q$ , where  $Q$  is an immediate assertion. This means that if the program starts with  $Q$  true, then  $Q$  is always true throughout its execution. While we proceed the proof of the liveness property, we only need to concern the reachable states starting from initial states where all invariants hold. Therefore, according to Lemma 5.4.1 (d), we are allowed to add to an assertion any conjunction of invariants freely at any time. To save some space, we denote  $color[x] = white$  by  $white(x)$ , and similarly for the other two colors. Moreover, we define the fastest process that arrives at the first phase, the second phase, the third phase and label 135, and the fastest process that finishes the current GC, respectively by:

$$\begin{aligned}
A_1(m) &\equiv (\exists p : pc_p \in [101, 110] \wedge rnd_p = shRnd = m) \\
&\quad \wedge \neg(\exists p : pc_p \notin [101, 110] \wedge rnd_p = shRnd = m), \\
A_2(m) &\equiv (\exists p : pc_p \notin [101, 110] \wedge rnd_p = shRnd = m) \\
&\quad \wedge \neg(\exists p : pc_p \in [129, 135] \wedge rnd_p = shRnd = m), \\
A_3(m) &\equiv (\exists p : pc_p \in [129, 135] \wedge rnd_p = shRnd = m) \\
&\quad \wedge \neg(\exists p : pc_p = 135 \wedge rnd_p = shRnd = m), \\
A_4(m) &\equiv (\exists p : pc_p = 135 \wedge rnd_p = shRnd = m), \\
A_0(m) &\equiv (\forall p : rnd_p \neq shRnd) \wedge shRnd = m.
\end{aligned}$$

To prove Theorem 5.4.1, we first prove the following lemmas with PVS, which are related to the “steps-to” ( $\triangleright$ ) relation and the “unless” ( $\mathcal{U}$ ) relation.

**Lemma 5.4.4** *For any integer  $m$ ,*

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. At several occasions where PVS refused to let a proof be finished, we actually found a mistake and had to correct previous versions of the algorithm. For the complete mechanical proof, we refer the reader to [32].

The entrenched problem inherited from classical mark&sweep algorithms is that our algorithm may also result in severe memory fragmentation, with lots of small blocks. It is possible that there will be no block of memory on the free list large enough to hold a large object, such as an array. Thus, it is important to move free blocks that happen to be adjacent in memory. We plan in the future to incorporate some appropriate copying technique in our algorithm.