

RIJKSUNIVERSITEIT GRONINGEN

# Design and Verification of Lock-free Parallel Algorithms

Proefschrift

ter verkrijging van het doctoraat in de  
Wiskunde en Natuurwetenschappen  
aan de Rijksuniversiteit Groningen  
op gezag van de  
Rector Magnificus, dr. F. Zwarts,  
in het openbaar te verdedigen op  
vrijdag 15 april 2005  
om 16.15 uur

door

Hui Gao

geboren op 8 september 1969  
te Jiangsu, China

Promotores:                   Prof. dr. W.H. Hesselink  
                                  Prof. dr. ir. J.F. Groote

Beoordelingscommissie:   Prof. dr. M. Herlihy  
                                  Prof. dr. W. Fokkink  
                                  Prof. dr. S.D. Swierstra

ISBN 90-367-2231-4



The work in this thesis has been carried out under the auspices of the research school IPA (Institute for Programming research and Algorithmics).

# Acknowledgements

Finally, I wish to express my gratitude to all persons who gave me the possibility to finish this thesis, though there are so many people that I can't mention all of them right here.

First, I would like to express my thanks to my former advisors Prof. Dr. H.G. Dehling and Prof. Dr. A.C. Hoffmann, and Prof. W. Schaafsma for creating the job opportunity for me to start the research as a Ph.D candidate in University of Groningen, in February 2000.

Then I would like to thank my present advisor Prof. Dr. Wim H. Hesselink for taking me over in February 2001, after both of my former advisors left the university of Groningen. In the following four years, he kept an eye on the progress of my work and was always available when I needed his advice. He has always been a great source of wisdom, knowledge and experience. My second advisor, Prof. Dr. Jan Friso Groote came to direct my research one year later. His critical comments, valuable hints and creative suggestions helped me to overcome many obstacles in my research. I am deeply indebted to my two advisors. The combination of their technical and editorial advice substantially contributed to the completion of this thesis. I am really glad and lucky that I have them both as my mentors.

I am very grateful to the members of the reading committee, Prof. Dr. Maurice Herlihy, Prof. Dr. Wan Fokkink, and Prof. Dr. Doaitse Swierstra, for reviewing the manuscript of my thesis and giving me many valuable comments.

Thanks also go to the chairman of Fundamental Computing Science group, Prof. Dr. Gerard R. Renardel de Lavalette, who was always willing to provide a solid financial support for me to attend summer schools and conferences.

I am grateful to all the members in our reading club, Wim H. Hesselink, Jan Eppo Jonker, Jan H. Jongejan, Hendrik Wietze de Haan and Roland Veen, for innumerable discussions, and valuable comments on all the drafts of the papers presented in this thesis.

I also want to extend my thanks to my former and present office mates, Daniel Straumann, Nico Kruithof, Hendrik Wietze de Haan and Marek Burza, for providing a good working atmosphere and sharing of tips, teas, cookies, and etc. Especially, I want to thank Hendrik Wietze de Haan for helping me to translate the summary of this thesis into Dutch. Big thanks to our secretaries, E.D. Elshof, D.J. Hansen and Y.G.M. Vergnes, for their kind assistance with various practical problems.

Many thanks goes to all my friends who've stood by me through all the years.

Last, but not least, I would like to give my special thanks to my family. It was my wife, Lixia Tang, who inspired me to study in the Netherlands five years ago. It was my son, Shenghan Gao, who frequently corrected the pronunciation of my English. My wife and my son's unconditional and endless love as well as their steady and substantial support are the main source of my life. My warm thanks go to my two elder sisters, Bi Gao and Hua Gao, for their kind concern and having taken care of my parents for so many years. My parents, Zhongnan Gao and Juru He, receive my deepest gratitude and love for always being there for me. Indeed, without love, inspiration, encouragement, and optimism from them, I would never have been able to finish this work.

# Contents

<b>Acknowledgements</b>	<b>iv</b>
<b>1 Introduction</b>	<b>1</b>
1.1 Shared memory architectures . . . . .	2
1.1.1 The programming model . . . . .	3
1.1.2 Synchronization . . . . .	4
1.2 Correctness . . . . .	8
1.2.1 The temporal logic . . . . .	8
1.2.2 Safety property . . . . .	9
1.2.3 Liveness property . . . . .	10
1.3 Verification . . . . .	11
1.3.1 Model checking . . . . .	11
1.3.2 Theorem proving . . . . .	12
1.4 Introduction to PVS . . . . .	14
1.4.1 The PVS specification language . . . . .	14
1.4.2 The PVS prover . . . . .	17
1.4.3 Experiences with PVS . . . . .	17
1.5 Overview of the thesis . . . . .	19
<b>2 Lock-free dynamic hash tables with open addressing</b>	<b>21</b>
2.1 Introduction . . . . .	22
2.2 The interface . . . . .	25
2.3 The algorithm . . . . .	28
2.3.1 Hashing . . . . .	29
2.3.2 Tagging of values . . . . .	30

2.3.3	Data structure . . . . .	30
2.3.4	Primary procedures . . . . .	32
2.3.5	Memory management and concurrent migration . . . . .	38
2.4	Correctness (Safety) . . . . .	47
2.4.1	Main properties . . . . .	48
2.4.2	Intuitive proof . . . . .	49
2.4.3	The model in PVS . . . . .	50
2.5	Correctness (Progress) . . . . .	53
2.5.1	The easy part of progress . . . . .	54
2.5.2	Progress of newTable . . . . .	54
2.5.3	The failure of wait-freedom . . . . .	55
2.6	Conclusions . . . . .	56
<b>3</b>	<b>A formal reduction for lock-free parallel algorithms</b>	<b>57</b>
3.1	Introduction . . . . .	58
3.2	Lock-free transformation . . . . .	59
3.3	Reduction . . . . .	60
3.3.1	Observed Specification . . . . .	61
3.3.2	Refinement mappings . . . . .	61
3.3.3	Correctness . . . . .	62
3.4	A lock-free pattern . . . . .	63
3.4.1	Simulation . . . . .	65
3.4.2	Refinement . . . . .	67
3.5	Large object . . . . .	68
3.6	Conclusions . . . . .	70
<b>4</b>	<b>A general lock-free algorithm using compare-and-swap</b>	<b>72</b>
4.1	Introduction . . . . .	73
4.2	Synchronization primitives . . . . .	75
4.3	The lock-free implementation using CAS . . . . .	76
4.4	Correctness . . . . .	78
4.4.1	Invariants . . . . .	79
4.4.2	Refinement . . . . .	80
4.4.3	Progress . . . . .	81

4.5	Conclusions . . . . .	83
<b>5</b>	<b>Lock-free parallel garbage collection by mark&amp;sweep</b>	<b>84</b>
5.1	Introduction . . . . .	85
5.2	Specification . . . . .	88
5.3	A higher-level implementation . . . . .	93
5.3.1	Data Structure . . . . .	94
5.3.2	Algorithm . . . . .	96
5.4	Correctness . . . . .	105
5.4.1	The main loop . . . . .	105
5.4.2	Safety properties . . . . .	106
5.4.3	Liveness . . . . .	108
5.5	The low-level lock-free implementation . . . . .	113
5.6	Practical experiment . . . . .	114
5.7	Conclusions . . . . .	116
<b>A</b>	<b>For lock-free dynamic hash tables</b>	<b>118</b>
A.1	Invariants . . . . .	118
A.2	Dependencies between invariants . . . . .	129
<b>B</b>	<b>For lock-free parallel GC</b>	<b>137</b>
B.1	Invariants . . . . .	137
B.2	Dependencies between invariants . . . . .	141
B.3	The low-level lock-free algorithm . . . . .	143
B.3.1	Data Structure . . . . .	143
B.3.2	Algorithm . . . . .	144
	<b>Bibliography</b>	<b>151</b>
	<b>Summary</b>	<b>157</b>
	<b>Samenvatting</b>	<b>162</b>



# Chapter 1

## Introduction

Computer programs tell the computer what specific operations to perform and in what specific order to carry out specified tasks. Algorithms are the essential parts of computer programs. Computer applications are often based on several algorithms. An algorithm is usually applicable in many applications and is to a large extent independent of the particular programming language used for the applications. An error in the design of an algorithm for solving a problem can lead to failures in the implementing program.

The basic unit of execution in many operating systems is called a *process*. In some cases, what you think of as a single program (e.g., a web or database server) actually consists of multiple processes communicating with each other. On uniprocessor systems, the processes rely on the operating system to frequently switch from one process to another so that it appears as if all the processes are executing at the same time. Many modern computers are multiprocessor systems, which have specialized hardware capable of executing several programs simultaneously. The main goals of the use of multiprocessor systems are to speed up the computations by using multiple processors, to perform large computations that are not possible on a uniprocessor system, and to allow subtasks of a larger job to run concurrently.

Algorithms that allow a computer performing more than one task concurrently are called *parallel algorithms*. To avoid chaos, likely to occur when multiple tasks or processes compete for scarce shared resources, we need some form of synchronization. The classical synchronization paradigms using locks can lead to many problems, including the waste of some time that we have won by parallelization. These are the reasons to strive for lock-free parallel algorithms. In section 1.1, we explain this in more detail.

Hardware and software are widely used in applications where failure is unacceptable. Since parallel algorithms are executed by letting the processors perform their actions concurrently, with steps interleaved in a nondeterministic way, it is generally impossible to understand the algorithms by predicting exactly how they will execute. Therefore, it is much harder to design correct parallel algorithms than their sequential counterparts. Furthermore, since different executions of them may use a different order and the number of combinations is exponential, it is usually impossible to test them well enough to trust them. Indeed, many published parallel algorithms are wrong or have wrong correctness proofs.

Lock-free algorithms are among the most difficult parallel algorithms. When testing is not enough or even impossible, we have to verify the correctness of them by using mathematical proof techniques. Nobody can avoid making mistakes, and it is important to verify the design as early as possible to exclude incorrectness. Ideally, the design and its proof are developed hand in hand. This explains why the words “design and verification” are combined in the title of this thesis.

In section 1.2, the concept of “correctness” is worked out. In section 1.3, we go deeper into the concept of “verification”. Since verification with a complete hand-written proof is too much work for the algorithms we are dealing with, we use the proof checker PVS for this purpose, as explained in section 1.4. We give an overview of the remainder of the thesis in section 1.5.

## 1.1 Shared memory architectures

We are interested in parallelism based on modern shared-memory multiprocessor that can access a common shared address space. These shared-memory multiprocessor systems tightly couple multiple microprocessors (with high speed communications between them), memory and I/O, and provide more computing power in a single machine. On a multiprocessor system, the programmer is expected to enable the efficient sharing of these resources, and take care that the different processors help each other rather than hinder each other.

A shared-memory multiprocessor system shows only a single memory image to the user even though the memory is physically distributed over the processors. Each processor is allowed to read from and write into each memory location. Sometimes the collaboration of different processors requires them to wait for each other. More than one processor may read the same memory location at the same time. However, on this architecture, to ensure the

coherent contents of the various processor caches, simultaneous writing of the same memory location and simultaneous reading and writing of the same memory location cannot take place. Such simultaneous read and write actions are sequentialized, meaning that they take place in an arbitrary order.

### 1.1.1 The programming model

An atomic action is a sequence of one or more statements that appear to be executed as a single, indivisible action without interruption. There are two kinds of atomic actions, namely *fine-grained* and *coarse-grained* atomic actions. A fine-grained atomic action is one that can be implemented directly by a single machine instruction. A coarse-grained atomic action consists of a sequence of fine-grained atomic actions that are (thought to be) executed atomically. Each *sequential process* consists of a series of atomic instructions, and can be viewed as a sequence of events or actions.

The programming model for shared-memory multiprocessor machines is a non-deterministic interleaving model. On the model, a concurrent program can be considered as a nonempty collection of sequential processes. These sequential processes communicate and/or synchronize with each other. The choice of the process from which the next atomic statement is selected is arbitrary, except for the synchronization statements.

A concurrent program consists of a declaration of variables, their initial values, and a set of atomic statements. Some of these variables represent *data* variables (e.g., global or local variables), which are explicitly manipulated by the program text. Some are *control* variables, which represent, for example, the location of control for each process in a concurrent program. Others are auxiliary variables, which are only used for the verification. The *state* of a program at any point in time consists of the values of all its variables, which can be characterized by a predicate called an *assertion*. A concurrent program can be modeled as a transition system  $\mathcal{S} : (\Sigma, \Theta, \mathcal{N})$  where

1.  $\Sigma$  is the state space.
2.  $\Theta$ , a predicate on  $\Sigma$  that determines the initial states.
3.  $\mathcal{N}$ , the next-state relation, is a reflexive relation on  $\Sigma \times \Sigma$ .

The next-state relation  $\mathcal{N}$  describes all the state transitions, and is required to be reflexive in order to allow stutterings (or idlings) [49, 53]. An *execution* sequence of  $\mathcal{S}$  is a nonempty list

over  $\Sigma$ , in which every pair of consecutive elements corresponds to executing one program step and belongs to  $\mathcal{N}$ . Each program corresponds to the set of all possible execution sequences starting from the initial states. A state is called *reachable* iff it occurs in an execution starting from the initial states. The program is terminated when there are no further state changes.

As is well-known, a next-state relation  $\mathcal{N}$  can be described by a *Hoare triple*:  $\{\varphi\}\mathcal{N}\{\psi\}$ , where  $\varphi$  and  $\psi$  are predicates on the state. This Hoare triple expresses that, if an atomic action represented by the next-state relation  $\mathcal{N}$  is executed in a state where  $\varphi$  holds, and if the execution terminates, then it terminates in a state where  $\psi$  holds.

### 1.1.2 Synchronization

On shared-memory architectures, processes coordinate with each other via shared data structures, called *concurrent objects*. The order of execution of multiple processes can alter the meaning of programs. A concurrent program must be correct under all possible interleavings. There is no assumption on relative process speeds, synchronized clocks, etc.

However, to ensure the consistency of these concurrent objects, there are often *critical sections*, sequences of actions on one or more data objects, that must be executed by one process at a time. Data sharing is the main reason that concurrent programming is so difficult. When different processes are working on the same critical section, they may disturb each other's work, leading to a chaotic behavior or an undesired computation result. This problem is often called a *synchronization problem*. Typically, synchronization primitives are used to avoid such interference between the processes.

#### Hardware primitives

Standard hardware instructions, e.g. *load* and *store*, are atomic with respect to each other. Each of them involves only at most one memory access. E.g. an assignment statement appears to execute as an atomic action if it satisfies the *at-most-once property*: an attribute of an assignment statement  $x = e$  in which either (1)  $x$  is not read by another process and  $e$  contains at most one reference to a variable changed by another process, or (2)  $e$  contains no reference to variables changed by other processes.

Many machines provide special hardware primitives that allow the processes to perform several standard instructions atomically. They can be used to build blocks (i.e., software primitives) to solve versatile synchronization problems. The commonly available special

hardware primitives are *test-and-set*(TAS), *compare-and-swap*(CAS), *fetch-and-increment*(FAI), *fetch-and-decrement*(FAD) and *load-linked/store-conditional*(LL/SC).

Hardware primitives do not block. However, using special hardware primitives directly may result in non-portable code.

### Classical software primitives

Rather than relying on assumptions about the granularity of hardware primitives, it is preferable to use software primitives that are built on the hardware primitives to provide higher-level synchronization support. Classical software solutions are spin-locks, mutexes, semaphores, condition variables and monitors, which are so common that they are often embedded into programming languages.

A *spin-lock* (also known as *busy waiting*) is an implementation of synchronization in which a process repeatedly executes a loop waiting for a boolean condition to be true. Though spin-locks are very efficient, one of the most serious drawbacks of spin-locks is that they can consume all the available CPU cycles without performing a useful task.

When several processes want to access a critical section, mutually exclusive locks (or *mutexes*) form a protection mechanism that serializes their accesses to the critical section by assuring a single, exclusive owner at any time. Mutexes have two basic operations for each critical section, namely **lock** and **unlock**. If a process calls **lock** on an unlocked critical section, then the critical section is locked and the process is enabled to continue its execution. If, on the other hand, the critical section is locked by some process, then the other processes that want to access that critical section will be blocked until the process calls **unlock** to release the resource.

Conceptually, a *semaphore* is a simple atomical counter. Semaphores are typically used to coordinate access to resources. A process calls **up** to atomically increment the counter when resources are added or released by the process, and calls **down** to decrement the counter when resources are removed or occupied by the process. When the semaphore becomes zero, which indicates that no more resources are available, processes trying to decrement the semaphore will be blocked until the counter becomes positive. The initial value of a semaphore indicates the number of identical instances of a critical resource. A semaphore initialized to 1 serves as a mutex.

Sometimes we want to check a condition in a critical section and then wait for it to be valid. A *condition variable* allows processes to synchronize on the value of data. It

provides a logical abstraction for suspending a process's execution until the data reaches some particular state or until some particular event occurs. The condition is tested under the protection of a mutex. When the condition is false, a process usually calls **wait** to block on a condition variable and atomically releases the mutex. When another process changes the condition, it can call **signal** to wake up one or more processes waiting on the associated condition variable. If several processes are waiting on a condition variable, a **broadcast** awakens all of them.

A concurrent object is an abstract data type that permits concurrent operations that appear to be atomic [27, 52, 69]. Essentially a *monitor* is a high level primitive with a mutex and several condition variables. This encapsulation of synchronization allows users of the resource to assume it to be properly synchronized (only one process can be active inside the monitor at a time). No extra synchronization code is needed at each entry point of the resource. A monitor provides a program scope, local variables, and multiple entry points. On behalf of its calling process, any operation may suspend itself by starting to wait on a condition, and thereby releasing the control of the monitor.

### Disadvantages of lock-based synchronization

In multiprogrammed systems, synchronization often turns out a performance bottle neck, due to preemptions. Most lock-based synchronization algorithms perform poorly in the face of such delays, because a delayed process holding a lock can impede the progress of other processes waiting for that lock. Furthermore, due to blocking and waiting for a resource, the classical synchronization paradigms using locks can lead to many problems such as convoying, priority inversion, deadlock and livelock. Many algorithms have been developed to limit the effects of these problems.

*Convoying* occurs when a process holding a lock is delayed and blocks all other processes. Sources of these delays include cache misses, remote memory accesses, page faults, scheduling preemptions and interrupts. *Priority inversion* occurs when a high-priority task is blocked and is waiting for a lock, but the lock holder does not make progress due to its low priority. *Deadlock* means that no process can make progress; this can occur when processes hold locks while waiting for locks held by other processes, so that no process can make progress. *Livelock* is the busy-waiting analog of deadlock. It occurs when every process is spinning while waiting for a condition that will never become true.

An important property of a lock is its *granularity*. The granularity is the size of the

object that is locked. Generally, using coarser granularity of the locks simplifies programming, but hampers the performance when many processes are needing concurrent access to the protected object. Conversely, using a finer granularity increases both the overhead of the locks and the risk of deadlock, but reduces lock contentions.

### Lock-free and wait-free objects

The easiest way to implement concurrent objects is by means of classical software solutions, but this leads to blocking when the process that holds exclusive access to the object is delayed or stops functioning.

The object is said to be *lock-free* if any process can be delayed at any point without forcing any other process to block and when, moreover, it is guaranteed that always some process will complete its operation in a finite number of steps, regardless of the execution speeds of the processes and possible adversarial scheduling [6, 28, 48, 64, 69]. However, some process might always lose to some faster process, but this is often unlikely in practice.

We regard “non-blocking” as synonymous to “lock-free”. In several recent papers, e.g. [67], the term “non-blocking” is used for the first conjunct in the above definition of lock-free. Note that this weaker concept does not in itself guarantee progress. Indeed, without real blocking, processes might delay each other arbitrarily without getting closer to completion of their respective operations. The older literature [2, 6, 30] seems to suggest that originally “non-blocking” was used for the stronger concept, and lock-free for the weaker one. Be this as it may, we use lock-free for the stronger concept.

As lock-free synchronizations are built without locks, they are immune from the aforementioned problems. In addition, lock-free synchronizations can offer progress guarantees. Herlihy [27] has shown that the primitive *CAS* and the similar *LL/SC* are universal primitives that solve the consensus problem. A number of researchers [6, 9, 28, 29, 51, 54] have proposed techniques for designing *lock-free* implementations. Essential for such implementations are special hardware instructions such as *LL/SC*, or *CAS*.

The object is said to be *wait-free* when it is guaranteed that any process can complete any operation in a finite number of steps, regardless of the speeds of the other processes [27]. Observe that this gives a stronger fault tolerance than lock-free, since any number of other processes can stop at arbitrary points in their executions without stopping the execution of other processes. However, wait-free objects are much more difficult to construct and usually less efficient.

## 1.2 Correctness

Concurrent programming is more difficult and error prone than sequential programming. It is notoriously difficult to ensure absence of runtime errors such as race conditions and dangling pointers, which may cause unpredictable or irreproducible behavior.

Correctness is essential because of the increasing integration of software in different kinds of applications such as embedded systems, communication protocols, transportation systems, etc. Ensuring the correctness of the design at the earliest possible stage is a major challenge in any responsible system development where the failure could be fatal and very expensive. A dramatic example of such a failure is the Ariane 5 rocket. Due to a software error, which was responsible for calculating the rocket's movement, it exploded on June 4, 1996.

The basic correctness conditions for concurrent systems are functional correctness and atomicity, say in the sense of [52], chapter 13. In order to verify the functional correctness of a program, one needs to specify the programming model of the behavior (derived from the requirements) of the program in a formal language. These specifications are critical since they can serve in many different applications. They must be correct before the applications are built upon them.

It may be easy to prove the correctness of an algorithm under assumption of a coarse grain of atomicity, but this can impose too severe restrictions on the implementation. A fine grain of atomicity is easier to implement, but it may make it harder to prove the correctness of the algorithm.

*Partial correctness* is a property of a program that computes the desired result, assuming the program terminates. *Total correctness* means: partial correctness and termination. Every correctness property a system satisfies can be formulated in terms of two kinds of properties: safety and liveness.

### 1.2.1 The temporal logic

Temporal Logic is a well-developed branch of modal logic with a notion for arguing about the times when assertions are true. It is widely applied to specifications and verifications of programs.

Time in temporal logic is discrete. A formula for asserting aspects of the state at a certain point in time, is called a *state formula* (or simply an assertion). In addition to



making a *temporal formula* out of *state formulas* by applying the boolean operators ( $\neg$ ,  $\wedge$ ,  $\vee$ , and so on), quantifiers  $\forall$  and  $\exists$ , in this thesis we use the following *temporal operators* to express assertions temporal:

- $\Box$  : always, meaning “is true now and forever”;
- $\Diamond$  : eventually, meaning “is true now or sometime in the future”;
- $\bigcirc$  : next, meaning “is true at the next point in time”.

Quite often,  $\bigcirc x$  is abbreviated as  $x'$ .  $\varphi \rightarrow \psi$  holds if  $\varphi$  implies  $\psi$  in the current state. We write  $\varphi \Rightarrow \psi$  as an abbreviation for  $\Box(\varphi \rightarrow \psi)$ .  $\mathcal{S} \models \varphi$  denotes that  $\varphi$  is logical consequence of the specification  $\mathcal{S}$ .

### 1.2.2 Safety property

Safety properties assert that nothing bad will ever happen, and are falsified when the program enters a bad state. Partial correctness, mutual exclusion, and absence of deadlock are examples of safety properties.

An assertion is an *invariant* iff it holds in every reachable state. It follows that every reachable state satisfies every invariant, and safety properties can be reduced to invariant properties. For establishing invariance properties, we use the following standard invariant rule as the main working tool:

**Rule INV1.** For assertion  $\varphi$ ,

$$\begin{array}{l} \Theta \rightarrow \varphi \\ \{\varphi\} \mathcal{N} \{\varphi\} \end{array}$$

---


$$\mathcal{S} \models \Box\varphi$$

The consequent of the rule, which is below the line, states that program  $\mathcal{S}$  satisfies  $\Box\varphi$ , i.e. always  $\varphi$ . So,  $\varphi$  is indeed an invariant. The rule allows to infer this from the two antecedents above the line: by the first antecedent,  $\varphi$  holds initially, and by the second antecedent it is propagated from each state to its next state.

Quite often the invariant property  $\varphi$  one wants to verify is not strong enough to be proven by itself (i.e. not *inductive*). To prove assertion  $\varphi$  to be an invariant, we normally need to find an inductive invariant  $\psi$ , which is stronger than assertion  $\varphi$  but weaker than the initial conditions and is preserved by any computational step of the system. It is important to realize that if  $\varphi$  is an invariant, then there always exists an inductive invariant  $\psi$  stronger than  $\varphi$  provided the language is rich enough [53].

In general, in order to prove that  $\varphi$  is an invariant, it is sufficient to use some already proved invariant  $I$  to prove either of the following two rules:

**Rule INV2.** For invariant  $I$ , assertion  $\varphi$ ,

$$I \Rightarrow \varphi$$

---


$$\mathcal{S} \models \Box \varphi$$

**Rule INV3.** For invariant  $I$ , assertion  $\varphi$ ,

$$\Theta \rightarrow \varphi$$

$$\{I \wedge \varphi\} \mathcal{N} \{\varphi\}$$

---


$$\mathcal{S} \models \Box \varphi$$

Since invariant  $I$  is usually denoted by a huge formula, we need to use it in an efficient way.

### 1.2.3 Liveness property

A liveness property asserts that something good will eventually happen—namely, that the program must eventually reach a good state. Termination and eventual entry into a critical section are examples of liveness properties.

In some circumstances, we need to assume that the programming model has some sense of fairness, by which we mean that in its long-term behavior it does not show undue bias in favoring some process when making nondeterministic choices. The purpose for fairness conditions is to rule out executions where the system idles indefinitely with control at some internal point of a procedure and with some transition of that procedure enabled. The proof of liveness relies on the fairness conditions associated with a specification.

For any atomic action  $\mathcal{A}$ , the predicate  $En(\mathcal{A})$  is defined to be the predicate that is true for a state iff it is possible to take an  $\mathcal{A}$  step starting in that state. The weak fairness condition  $WF(\mathcal{A})$  is the condition that in every execution, action  $\mathcal{A}$  must eventually be taken if  $En(\mathcal{A})$  remains true. The strong fairness condition  $SF(\mathcal{A})$  is the condition that in every execution, action  $\mathcal{A}$  must be executed infinitely often if  $En(\mathcal{A})$  is infinitely often true.

Liveness properties are often expressed using the “leads-to” relation (denoted as  $\circ \rightarrow$ ). The leads-to relation is defined by:  $(\varphi \circ \rightarrow \psi) \equiv \Box(\varphi \rightarrow \Diamond \psi)$ , which means, whenever  $\varphi$  is true,  $\psi$  will be true now or in the future. The following rules (stated in [49]) allow to deduce a simple leads-to formula from a weak fairness condition and a strong fairness condition, respectively.

**Rule WF1.**

$$\begin{array}{l}
\{\varphi\} \mathcal{N} \{\varphi \vee \psi\} \\
\{\varphi\} (\mathcal{N} \wedge \mathcal{A}) \{\psi\} \\
\varphi \wedge I \Rightarrow \text{En}(\mathcal{A}) \\
\hline
\mathcal{S} \models \text{WF}(\mathcal{A}) \Rightarrow (\varphi \circ \rightarrow \psi)
\end{array}$$

**Rule SF1.**

$$\begin{array}{l}
\{\varphi\} \mathcal{N} \{\varphi \vee \psi\} \\
\{\varphi\} (\mathcal{N} \wedge \mathcal{A}) \{\psi\} \\
\Box \varphi \wedge \Box \mathcal{N} \wedge \Box I \Rightarrow \Diamond \text{En}(\mathcal{A}) \\
\hline
\mathcal{S} \models \text{SF}(\mathcal{A}) \Rightarrow (\varphi \circ \rightarrow \psi)
\end{array}$$

Where  $\varphi$ ,  $\psi$ , and  $I$  are predicates,  $\mathcal{N}$  is the (reflexive) next relation and  $\mathcal{A}$  is an irreflexive binary relation on  $\Sigma$ .

The rule *WF1* (or *SF1*) asserts that  $\varphi \circ \rightarrow \psi$  holds for a specification with next-state relation  $\mathcal{N}$  and weak fairness condition  $\text{WF}(\mathcal{A})$  (or strong fairness condition  $\text{SF}(\mathcal{A})$ ) for some non-stuttering action  $\mathcal{A}$ , provided we can prove the antecedents for every reachable state characterized by invariant  $I$ .

## 1.3 Verification

When writing a program, we often make mistakes. Syntax errors are easily caught or flagged by a good compiler. Code inspection by an independent team can normally detect most obvious errors, but no guarantee of correctness is made. Testing or debugging can often reveal more errors, but it cannot demonstrate the absence of bad states. Given a program, how can we determine that it always behaves as expected?

Verifying a program is proving, in a formal mathematical way, that the program has some desired properties written in logical formulas. There are two formal verification approaches, namely model checking and theorem proving.

### 1.3.1 Model checking

Model checking is a usual way of verification. It consists of an automatic exhaustive analysis of the reachable state space, which often must be finite. In model checking, the design of

a model as well as its desired properties are first converted into a formalism accepted by a model checker. Ideally, model checking can then be performed automatically by model checkers, which explicitly or implicitly enumerate the reachable state space of a (finite-state) reactive program, to verify the correctness of the system with respect to these logical formulas. When model checking fails, the user is often provided with an error trace. This can be used as a counterexample for the checked property.

Model checking does not aim at being fully general. It can only verify instantiations of systems. E.g., it can only verify the correctness of a network protocol for particular networks and not for general networks. It is fairly hard to capture a complete set of properties for all but the most simplistic designs. Though it is no longer absolutely restricted to finite-state systems, it is still only applicable to systems whose states have short and easily manipulated descriptions. This indicates that model checking cannot be used for the verification of data-intensive applications, where the state space is very large or even infinite.

The main challenge of model checking is how to deal with the so called “state explosion problem”: if the number of states is too large, the model checker requires unreasonable amount of time and memory to complete verification. E.g. in the concurrent program containing  $n$  processes, each with  $m$  atomic actions, the number of different states is  $(n \times m)!/(m!)^n$ . When  $n = 5$  and  $m = 10$ , this is a number of 32 digits.

However, model checking is a demonstrated success in the development of hardware products. Researchers and industrialists have used checkers like SMV, Murphi, COSPAN and SPIN to find bugs in many published circuit designs for multiprocessor. It has been adopted by the hardware community to complement the traditional validation method of hardware simulation.

### 1.3.2 Theorem proving

In theorem proving (also referred to as *deductive verification*), the proof of the correctness is mechanically checked by a theorem prover, such as PVS, Coq, Isabelle, HOL, ACL2 and Nqthm. The theorem proving tools consist of a powerful collection of inference rules that can be applied to repeatedly reduce a proof goal to simpler sub-goals until all the final proof goals can be discharged automatically by the primitive proof steps of the theorem prover. Most theorem provers give the user a lot more flexibility and control in doing the proofs. In case of a negative result, the user can derive a scenario. This scenario can give the user greater insights into the specification. Analyzing the scenario can lead to a modification to

the system and the verification.

Large systems that cannot be verified by model checking, can still be verified by theorem proving. Theorem proving can be used for reasoning about an infinite state space. It avoids state explosion by a compact (or logical) representation of states and state transformations. Since state space explosion is not a problem, no abstraction techniques need to be applied and the verification can directly be done on the parameterized (or general) model. Most theorem provers are highly expressive. Some properties that cannot be easily specified using model checkers can be easily specified in the languages of most theorem provers.

In principle, human-guided theorem proving can verify any correct design, but doing so may require considerable effort, time and skill. It can be performed only by experts with certain logic reasoning and considerable experience. Most theorem provers can be used in a variety of ways with different amounts of automation. Normally, they require a great degree of manual intervention. So far, there is no theorem prover that can fully automatically prove "interesting" theorems. Some experts roughly expect such an intelligent theorem prover in 200 years, but unfortunately, none of us can wait.

The main task in theorem proving is to show that some conjecture is a logical consequence of a set of the axioms, hypotheses and some already proved assertions. In this thesis, we use theorem proving to verify the correctness of the algorithms. Safety properties can be reduced to invariant properties, and to prove progress one usually needs to establish auxiliary invariant properties too.

In order to establish some invariance property, there are two methods used to find out the appropriate inductive invariant (that implies the property). One method is bottom-up approach. Using this method, we only need to analyze the given program alone, independently of the goal assertion whose invariance we wish to prove. This method is guaranteed to produce an inductive but maybe useless assertion. The alternative method is top-down approach, which takes into account both the program and the assertion. Guided by the given goal assertion being verified, this approach is guaranteed to produce a useful assertion, which however need not be inductive, and which may even turn out to be false.

Our proof architecture for verifying some invariance 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 root node, which characterizes the main 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.

The main property will not be proved until the proof tree stops growing and all leaf nodes in the tree have been proved. 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 assumed property turns out to be not valid. Indeed, this may help to uncover a defect of the algorithm.

## 1.4 Introduction to PVS

PVS (Prototype Verification System) is a mechanized framework for writing precise specifications and constructing interactive proofs. PVS uses the text editor *Emacs* to provide an integrated interface to its specification language, type checker and theorem prover. It exploits the synergy between a highly expressive specification language and powerful automated deduction, and is widely considered to be one of the most powerful theorem provers in use today.

Our lock-free algorithms presented in this thesis are so complicated that they cannot be verified by a model checker. Therefore, we have chosen the theorem prover PVS for mechanical support. In this section, we only provide a glimpse into the usage of PVS system. More detailed PVS documentation can be found in [63].

### 1.4.1 The PVS specification language

The PVS specification language is close to “normal” notation. It is based on a strongly typed higher-order logic with a rich type system. Specifications in PVS are structured into hierarchies of parameterized *theories*. A theory is a collection of types, constants, variables, definitions, assumptions, axioms and theorems. Constraints can be attached to the parameters and types, and thus contribute to the clarity of the specifications. PVS has an extensive library of theories of mathematics, called *preludes*, which provide many useful types, definitions, lemmas, etc. A theory may import or instantiate predefined or user-defined theories.

The type system of PVS includes uninterpreted types that may be introduced by the

user, a large number of built-in types (e.g. boolean, natural, integer and real), type-constructors (e.g. enumerations, tuples, records, functions and sets), subtypes and dependent types that can be used to introduce constraints, and abstract data types (e.g. lists and binary trees).

Types can be unspecified. The declaration:  $A : \text{TYPE}$ , defines  $A$  to be an uninterpreted type that is disjoint from other types.

The type-constructors are used extensively in the sequel. Enumeration types are used for defining types whose values can be completely enumerated. A tuple type has the form  $[T_1, \dots, T_n]$  where  $T_i$  is a type. The  $i$ th projection (function) is given by `'i` or `proj_i` with domain  $T_1 \times \dots \times T_n$  and range  $T_i$ . A record type is a finite list of fields of the form  $R : \text{TYPE} = [\# a_1 : T_1, \dots, a_n : T_n \#]$  where  $a_i$  is an accessor function. Given a record  $r : R$ ,  $a_i(r)$  or  $r.a_i$  is used to access the  $i$ -th field of a record  $r$ . Record types are similar to tuple types, except the order of the fields is irrelevant and accessors are used instead of projections. Function types are declared as  $F : \text{TYPE} = [T_1, \dots, T_n \rightarrow T]$ . An element of this type is simply a function whose domain is the sequence of types  $T_1, \dots, T_n$ , and range is  $T$ . A *lambda* expression allows writing a function expression without explicitly introducing the function name. E.g. the function that doubles an integer may be written as `LAMBDA (j : int) : 2*j`. Sets are represented as predicates in PVS in the form of  $\text{pred}[T]$  and  $\text{setof}[T]$ , which are shorthand for  $[T \rightarrow \text{bool}]$ .

Much of the expressive power of the language comes from subtypes and dependent types. The declaration  $T1 : \text{TYPE}$  **from**  $T$  declares  $T1$  to be a subtype of  $T$ . The predicate subtype  $\{x : T \mid P(x)\}$  consists of those elements of type  $T$  that satisfy the predicate  $P$ . The dependent types are constructed using predicate subtypes. An example of a dependent type in PVS, is the declaration of an resizable hash table as a record with two fields:  $\text{Hashtable} : \text{TYPE} = [\# \text{size} : \text{nat}, \text{table} : [\text{below}(\text{size}) \rightarrow \text{Value}] \#]$ , where  $\text{size}$  is a natural number denoting the size of the hash table, and  $\text{table}$  is a function denoting the values at each position in the hash table. The domain of  $\text{table}$  is the predicate subtype of the natural numbers less than  $\text{size}$  and thus depends on the actual size of the hash table.

The constraints introduced in predicate subtypes and dependent types may incur the type-checker to generate proof obligations called type correctness conditions (TCCs). In general, type checking of PVS specifications is undecidable, and thus requires the theorem proving capabilities of PVS.

A distinctive feature in the PVS language is that the type system is augmented with

abstract datatypes. An example of a datatype is the extended domain of values stored in our lock-free hash table:

```
EValue : datatype begin
  del : delp
  old(val1 : Value) : oldp
  nor(val2 : Value) : norp
end EValue
```

The **EValue** datatype has three constructors, namely *del*, *old* and *nor*, that allow an extended value to be constructed. E.g. **del** denotes a deleted value. The term *old*(*v1*) is the result after tagging value *v1* *old*. The recognizers *delp*, *oldp* and *norp* are predicates over **EValue**. They are true when their arguments (of type **EValue**) are constructed using the corresponding constructors. E.g. *oldp*(*old*(*v1*))  $\equiv$  *true*.

The PVS languages offers a close approximation of the standard mathematical notation such as arithmetic and logical operators, function application and etc. In PVS only total functions are allowed. For a recursive function a well-founded *measure* must be provided to show that it decreases for each recursive call. E.g. we define the *j*-th ancestor of a node ranging from 1 to *N* by the recursive function:

```
Ancestor(x : range(N), j : nat) : recursive range(N) =
  if j = 0 or father(x) ≤ 0 then x
  else Ancestor(father(x), j - 1) endif
  measure j
```

However, it is not allowed to define mutual recursion across two or more definitions.

Logical expressions can be used to construct both propositional and predicate calculus formulas. The logical constants are denoted as *true* and *false*. The basic logical constructs are: **not**( $\sim$ ), **and**(&), **or**, **implies**( $\Rightarrow$ ), **iff**( $\Leftrightarrow$ ). The universal and existential quantifiers are **forall** and **exists**, respectively.

Formula declarations introduce *axioms*, *assumptions*, *theorems* and *obligations* using keyword **AXIOM**, **ASSUMPTION**, **THEOREM** and **OBLIGATION**, respectively. The identifier associated with a declaration can be referenced during proofs. The body of the formula is a boolean expression. Axioms are boolean formulas taken as *true* in proofs. Internal to the theory, assumptions are used exactly as axioms. Externally, for each import of a theory, the assumptions have to be proved with the actual parameters. Theorems are boolean



formulas whose validities need to be established. Theorems may be introduced with other keywords such as **CLAIM** and **LEMMA**. Obligations are generated by the system for **TCCs**, and cannot be specified by the user. *Judgements* are lemmas about subtypes that get applied automatically during type checking.

A formula declaration may contain free variables, in which case PVS assumes the universal closure of the formula. E.g.  $p(x) \Rightarrow q(y)$  is equivalent to  $(\text{forall } x, y : p(x) \Rightarrow q(y))$ .

### 1.4.2 The PVS prover

PVS prover combines Gentzen's *sequent calculus* with a collection of powerful primitive inference procedures that are applied interactively under user guidance. The primitive inferences include propositional and quantifier rules, induction, rewriting, simplification and decision procedures.

The system supports top-down proof exploration and construction. The proof structure forms a tree, where the theorem to be verified is the root of the tree. The nodes of the tree are sequents of the form:  $a_1, \dots, a_n \vdash c_1, \dots, c_m$ , where  $a_i$  are called *antecedents* and  $c_i$  are called *consequents*. A sequent is *valid* if the disjunction of the consequents can be inferred from the conjunction of all the antecedents. The intuitive interpretation of the above sequent is that:  $a_1 \wedge \dots \wedge a_n \Rightarrow c_1 \vee \dots \vee c_m$ .

During the proof construction, one of the leaf sequents is the *current* sequent (i.e. the current proof state), to which the proof commands are applied. Each proof step results in *child* sequents (or sub-goals) that are at least as strong as their *parent* sequents. The root sequent (or theorem) is finally proved if the proof tree stops growing and all the leaf sequents have been proved valid.

The proof commands use combinations of the inference rules that are built in the prover. Moreover, PVS allows to combine several proof commands into high-level proof strategies to facilitate the reasoning. We refer to the PVS prover guide [63] for the available proof commands and the other aspects of interactive proving with PVS.

### 1.4.3 Experiences with PVS

In this thesis, we present two lock-free algorithms that are considered very complicated. The only approach capable of formally verifying these algorithms is using theorem proving. PVS's combination of a highly expressive specification language and a powerful interactive

proof checking capability yields a productive verification environment. Our experiences with PVS also show that it is a well-performing theorem-proving tool. It may take much more time to do the same work if we chose other theorem provers such as Nqthm, of which the proof states are much harder to read and analyze.

In general, theorem proving requires considerable technical expertise. The PVS system needs a precise description of the problem written in the PVS language. This forces the user to think carefully about the problem in order to produce an appropriate specification and hence requires a deep understanding of the problem. Moreover, one obtains a clear list of assumptions under which the algorithm is correct. With handwritten proofs, such assumptions are often unknown or hidden.

TCCs arise, for instance, when a term is type checked against an expected predicate subtype. In practice, TCCs can often be discharged automatically by the proof automation tools provided by the system. In general, type checking is a simple and effective way to discover many errors in specifications.

PVS provides a collection of powerful proof commands to carry out propositional, equality, and arithmetic reasoning with the use of definitions and lemmas. Case analysis is surprising useful for introducing assumptions that will eventually be discharged. To make proofs easier to debug, PVS permits proof steps to be undone. Quite often, the `grind` command is a good way to complete a proof that does not require induction. If `grind` does not work or takes too much time (say more than half minute) to complete the proof, we always first try to read the output of the sequent to get some new insights into the problem. If even this does not help, we then try to do case analysis to make some assumptions or classify the proof into sub-goals. Hiding irrelevant formulas in the sequent is rather useful, since it helps to reduce the running time of the proof, and enables the user to focus on the essential formulas.

According to our experiences, it may be easier to detect an error using model checking than theorem proving, but it is much more difficult to locate the source of the error. When the specification is adapted/extended, most parts of the PVS proofs can be re-used without much effort.

## 1.5 Overview of the thesis

Efficient programming of multi-processor systems holds a challenge to the designer because of the conflicting issues of efficient distribution and maximal concurrency, as opposed to reliable communication and predictable behavior. Communication between the processors needs a form of synchronization which in a poor design may hamper speed and even lead to deadlock. Since concurrent systems are very nondeterministic, it is usually almost impossible to perform tests that provide adequate coverage. Model checkers can be used for verification of communication protocols, but cannot verify data intensive algorithms. We therefore aim at the design of efficient and provably correct lock-free algorithms for multiprocessor systems.

Lock-free algorithms 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. In view of the complexity of the algorithms presented in this thesis, we turned to the interactive theorem prover PVS for mechanical support.

The chapters 2 and 3 concern our published papers [20, 21, 23]. Chapter 4 is a slightly modified version of our paper that is under submission. Chapter 5 concerns our technical report [22].

In the second chapter, we present an efficient lock-free algorithm for parallel accessible hash tables with open addressing, which promises more robust performance and reliability than conventional lock-based implementations. For a multiprocessor architecture our solution is as efficient as sequential hash tables. The algorithm allows processors that have widely different speeds or come to a halt. It can easily be implemented using C-like languages and requires on average only constant time for insertion, deletion or accessing of elements. The algorithm allows the hash tables to grow and shrink when needed.

For the correctness of the algorithm, we employ standard deductive verification techniques to prove around 200 invariance properties of our algorithm, and describe how this is achieved with the theorem prover PVS.

In the third chapter, we formalize Herlihy's methodology [28] for transferring a sequential implementation of any data structure into a lock-free synchronization by means of synchronization primitives *LL/SC*. This is done by means of a reduction theorem that enables us to reason about the general lock-free algorithm to be designed on a higher level than the synchronization primitives. The reduction theorem is based on refinement mapping as

described by Lamport [49] and has been verified with the theorem prover PVS. Using the reduction theorem, fewer invariants are required and some invariants are easier to discover and formulate.

The lock-free implementation works quite well for small objects. However, for large objects, the approach is not very attractive as the burden of copying the data can be very heavy. We propose two enhanced lock-free algorithms for large objects in which slower processes don't need to copy the entire object again if their attempts fail. This results in lower copying overhead than in Herlihy's proposal.

*CAS* is a synchronization primitive for lock-free algorithms. Most uses of it, however, suffer from the so-called *ABA* problem. The simplest and most efficient solution to the *ABA* problem is to include a tag with the memory location such that the tag is incremented with each update of the target location. However, applying this solution is not theoretically bug-free and limits the applicability of these algorithms.

In the fourth chapter, we present a general lock-free pattern that is based on the synchronization primitive *CAS* without causing the *ABA* problem or problems with wrap around. It can be used to provide lock-free functionality for any generic data type. Our algorithm is a *CAS* variation of Herlihy's *LL/SC* methodology for lock-free transformation. The basis of our techniques is to poll different locations on reading and writing objects, in such a way that the consistency of an object can be checked by its location instead of its tag. It consists of simple code that can be easily implemented using C-like languages.

In the fifth chapter, we present a lock-free parallel algorithm for mark&sweep garbage collection (GC) in a realistic model using synchronization primitives *LL/SC* or *CAS*. *Mutators* and *collectors* can simultaneously operate on the data structure. In particular no strict alternation between usage and cleaning up is necessary contrary to what is common in most other garbage collection algorithms.

We first design and prove an algorithm with a coarse grain of atomicity and subsequently apply the reduction theorem developed in chapter 3 to implement the higher-level atomic steps by means of the low-level primitives. 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 therefore turned to PVS for mechanical support.

Chapter 6 gives some conclusions and a summary of the contents of this thesis.

## Chapter 2

# Lock-free dynamic hash tables with open addressing

This chapter has been published as [21]. Its extended abstract appears in [20].

## 2.1 Introduction

We are interested in efficient, reliable, parallel algorithms. The classical synchronization paradigm based on mutual exclusion is not most suited for this, since mutual exclusion often turns out to be a performance bottleneck, and failure of a single process can force all other processes to come to a halt. This is the reason to investigate lock-free or wait-free concurrent objects, see e.g. [6, 27, 28, 30, 38, 46, 48, 55, 64, 66, 67, 69].

### Lock-free and wait-free objects

A *concurrent object* is an abstract data type that permits concurrent operations that appear to be atomic [27, 52, 69]. The easiest way to implement concurrent objects is by means of mutual exclusion, but this leads to blocking when the process that holds exclusive access to the object is delayed or stops functioning.

The object is said to be *lock-free* if any process can be delayed at any point without forcing any other process to block and when, moreover, it is guaranteed that always some process will complete its operation in a finite number of steps, regardless of the execution speeds of the processes [6, 28, 48, 64, 69]. The object is said to be *wait-free* when it is guaranteed that any process can complete any operation in a finite number of steps, regardless of the speeds of the other processes [27].

We regard “non-blocking” as synonymous to “lock-free”. In several recent papers, e.g. [67], the term “non-blocking” is used for the first conjunct in the above definition of lock-free. Note that this weaker concept does not in itself guarantee progress. Indeed, without real blocking, processes might delay each other arbitrarily without getting closer to completion of their respective operations. The older literature [2, 6, 30] seems to suggest that originally “non-blocking” was used for the stronger concept, and lock-free for the weaker one. Be this as it may, we use lock-free for the stronger concept.

### Concurrent hash tables

The data type of hash tables is very commonly used to efficiently store huge but sparsely filled tables. Before 2003, as far as we know, no lock-free algorithm for hash tables had been proposed. There were general algorithms for arbitrary wait-free objects [3, 6, 7, 27, 35, 36], but these are not very efficient. Furthermore, there are lock-free algorithms for different domains, such as linked lists [69], queues [68] and memory management [30, 38].

In this chapter we present a lock-free algorithm for hash tables with open addressing that is in several aspects wait-free. The central idea is that every process holds a pointer to a hash table, which is the current one if the process is not delayed. When the current hash table is full, a new hash table is allocated and all active processes join in the activity to transfer the contents of the current table to the new one. The consensus problem of the choice of a new table is solved by means of a test-and-set register. When all processes have left the obsolete table, it is deallocated by the last one leaving. This is done by means of a compare-and-swap register. Measures have been taken to guarantee that actions of delayed processes are never harmful. For this purpose we use counters that can be incremented and decremented atomically.

After the initial design, it took us several years to establish the safety properties of the algorithm. We did this by means of the proof assistant PVS [63]. Upon completion of this proof, we learned that a lock-free resizable hash table based on chaining was proposed in [66]. We come back to this below.

Our algorithm is lock-free and some of the subtasks are wait-free. We allow fully parallel *insertion*, *assignment*, *deletion*, and *finding* of elements. Finding is wait-free, the other three are not. The primary cause is that the process executing it may repeatedly have to execute or join a migration of the hash table. Assignment and deletion are also not wait-free when other processes repeatedly assign to the same address successfully.

Migration is called for when the current hash table is almost filled. This occurs when the table has to grow beyond its current upper bound, but also for maintenance after many insertions and deletions. The migration itself is wait-free, but, in principle, it is possible that a slow process is unable to access and use a current hash table since the current hash table is repeatedly replaced by faster processes.

Migration requires subtle provisions, which can be best understood by considering the following scenario. Suppose that process  $A$  is about to (slowly) insert an element in a hash table  $H_1$ . Before this happens, however, a fast process  $B$  has performed migration by making a new hash table  $H_2$ , and copying 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 hash table, 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 hash table, 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 hash table, and it is clear that processes should carefully keep each other informed about their activities on the tables.

### Performance, comparison, and correctness

Actually, only an extra check is required in the main loop of the main functions, one extra bit needs to be set when writing data in the hashtables and at some places a write operation has been replaced by a compare-and-swap, which is more expensive. For ordinary operations on the hashtable, this is the only overhead and therefore a linear speed up can be expected on multiprocessor systems. The only place where no linear speed up can be achieved is when copying the hashtable. Especially, when processes have widely different speeds, a logarithmic factor may come into play (see algorithms for the write all problem [24, 46]). Indeed, initial experiments indicate that our algorithm is as efficient as sequential hash tables. It seems to require on average only constant time for insertion, deletion or accessing of elements.

Some differences between our algorithm and the algorithm of [66] are clear. No formally verified correctness proof was given for the algorithm in [66]. In our algorithm, the hashed values need not be stored in dynamic nodes if the address-value pairs (plus one additional bit) fit into one word. Our hash table can shrink whereas the table of bucket headers in [66] cannot shrink. A disadvantage of our algorithm, due to its open addressing, is that migration is needed as maintenance after many insertions and deletions.

An apparent weakness of our algorithm is the worst-case space complexity in the order of  $\mathcal{O}(PM)$  where  $P$  is the number of processes and  $M$  is the size of the table. This only occurs when many of the processes fail or fall asleep while using the hash table. Failure while using the hash table can be made less probable by adequate use of the procedure “releaseAccess”. This gives a trade-off between space and time since it introduces the need of a corresponding call of “getAccess”. When all processes make ordinary progress and the hash table is not too small, the actual memory requirement is  $\mathcal{O}(M)$ .

The migration activity requires worst-case  $\mathcal{O}(M^2)$  time for each participating process. This only occurs when the migrating processes tend to choose the same value to migrate and the number of collisions is  $\mathcal{O}(M)$  due to a bad hash function. This is costly, but even this is in agreement with wait-freedom. The expected amount of work for migration for all



processes together is  $\mathcal{O}(M)$  when collisions are sparse, as should be the case when migrating to a hash table that is sufficiently large.

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 verify this using a proof checker or theorem prover.

As a correctness notion, we take that the operations behave the same as for ‘ordinary’ hash tables, under some arbitrary linearization [31] 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.

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 [11, 63] 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.

## Overview of this chapter

Section 2.2 contains the description of the hash table interface offered to the users. The algorithm is presented in Section 2.3. Section 2.4 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.1, while the relationships between the invariants are given by a dependency graph in Appendix A.2. Progress of the algorithm is proved informally in Section 2.5. Conclusions are drawn in Section 2.6.

## 2.2 The interface

The aim is to construct a hash table 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 hash table is an implementation of (partial) functions between two domains, here called *Address* and *Value*. The hash table thus implements a modifiable shared variable  $\mathbf{X} : \text{Address} \rightarrow \text{Value}$ . The domains *Address* and *Value* both contain special default elements  $0 \in \text{Address}$  and  $\mathbf{null} \in \text{Value}$ . An equality  $\mathbf{X}(a) = \mathbf{null}$  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

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

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

$$\text{Ax1: } v = \mathbf{null} \equiv ADR(v) = 0.$$

Indeed, we need  $\mathbf{null}$  as the value corresponding to the undefined addresses and use address 0 as the (only) address associated with the value  $\mathbf{null}$ . We thus require the hash table to satisfy the invariant

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

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  $\mathbf{null}$  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.  $\mathbf{X}(a) = \mathbf{null}$ . 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 hash table. Each process is characterized by the sequence of operations

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

A process that needs to access the table, first calls the procedure *getAccess* to get the current hash table 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 [52], chapter 13. Functional correctness is expressed by prescribing how the procedures *find*, *insert*, *delete*, *assign* affect the value of the abstract mapping **X** in relation to the return value. Atomicity means that the effect on **X** and the return value takes place 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.

```

proc findS(a : Address \ {0}) : Value =
    local rS : Value;
(fS)    $\langle$  rS := X(a)  $\rangle$ ;
return rS.

```

```

proc deleteS(a : Address \ {0}) : Bool =
    local sucS : Bool;
(dS)    $\langle$  sucS := (X(a)  $\neq$  null) ;
        if sucS then X(a) := null end  $\rangle$  ;
return sucS.

```

```

proc insertS(v : Value \ {null}) : Bool =
    local sucS : Bool ; a : Address := ADR(v) ;
(iS)    $\langle$  sucS := (X(a) = null) ;
        if sucS then X(a) := v end  $\rangle$  ;
return sucS.

```

```

proc  $assign_S(v : Value \setminus \{\mathbf{null}\}) =$ 
    local  $a : Address := ADR(v) ;$ 
(aS)     $\langle X(a) := v \rangle ;$ 
end.

```

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**  $X(a)$ .

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  $X(ADR(v)) \neq v$  since another process deletes  $ADR(v)$  between the execution of (iS) and the response of *insert*.

In Section 2.3.4, we provide implementations for the operations *find*, *delete*, *insert*, *assign*. We prove partial correctness of the implementations by extending them with the auxiliary variables and commands used in the specification. So, we regard  $X$  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 each of the four implementations executes its specification command always exactly once and that the resulting value  $r$  or  $suc$  of the implementation equals the resulting value  $rS$  or  $sucS$  in the specification. 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 analogous to removal of stutterings in TLA [49] or abstraction from  $\tau$  steps in process algebras.

## 2.3 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.

### 2.3.1 Hashing

We implement function  $\mathbf{X}$  via hashing with open addressing, cf. [47, 70]. We do not use direct chaining, where colliding entries are stored in a secondary list, as is done in [66]. A disadvantage of open addressing with deletion of elements is that the contents of the hash table must regularly be refreshed by copying the non-deleted elements to a new hash table. As we wanted to be able to resize the hash tables anyhow, we consider this less of a burden.

In principle, hashing is a way to store address-value pairs in an array (hash table) 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. [47].

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

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

where  $l$  is the length of the array (hash table), that satisfies

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

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

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

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

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

### 2.3.2 Tagging of values

As is well known [47], hashing with open addressing needs a special value **del**  $\in$  *Value* to replace deleted values.

When the current hash table becomes full, the processes need to reach consensus to allocate a new hash table of new size to replace the current one. Then all values except **null** and **del** must be migrated to the new hash table. A value that is being migrated cannot be simply removed, 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:

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

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

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

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  $ADR(v) = ADR(val(v))$ .

### 2.3.3 Data structure

A *Hash table* is either  $\perp$ , indicating the absence of a hash table, or it has the following structure:

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

The field **size** indicates the size of the hash table, **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 hash

table, we write *h.size*, *h.occ*, *h.dels* and *h.bound* to access these fields of the hash table. We write *h.table*[*i*] to access the *i*<sup>th</sup> *EValue* in the table.

Apart from the *current* hash table, which is the main representative of the variable **X**, we have to deal with *old* hash tables, which were in use before the current one, and *new* hash tables, 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 hash table and allow to delete hash tables that are not used anymore. The basic idea is to count the number of processes that are using a hash table, by means of a counter **busy**. The hash table can be thrown away when **busy** is set to 0. An important observation is that **busy** cannot be stored as part of the hash table, 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 hash table 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 hash table 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 hash tables and the **busy** counters. There can be  $2P$  hash tables around, because each process can simultaneously be accessing one hash table 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 hash table to which the contents of hash table **H**[*i*] is being copied. If **next**[*i*] equals 0, this means that there is no next hash table. 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 them.

We use a shared variable **currInd** to hold the index of the currently valid hash table:

```

currInd : 1 ..  $2P$  .

```

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

It is assumed that initially  $H[1]$  is pointing to some hash table. The other initial values of the shared variables are given by

$$\begin{aligned} \text{currInd} &= \text{busy}[1] = \text{prot}[1] = 1, \\ H[i] = \text{busy}[i] = \text{prot}[i] &= 0 \text{ for all } i \neq 1, \\ \text{next}[i] &= 0 \text{ for all } i. \end{aligned}$$

### 2.3.4 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 hash table. At entry of each primary procedure, it must be the case that the variable  $H[\text{index}]$  contains valid information. In section 2.3.5, we provide the procedure *getAccess* with the main purpose to guarantee this property. When *getAccess* has been called, the system is obliged to keep the hash table at *index* stored in memory, even if there are no accesses to the hash table 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 hash table for some time.

### Syntax

We use a syntax analogous to Modula-3 [25]. 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 loops.  
**if .. then .. {elsif ..} [else ..] end** is the conditional statement.  
**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.



### The main loop

We model the clients of the hash table in the following loop. 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  $a \neq 0 \rightarrow \text{find}(a)$ 
        (d, a) with  $a \neq 0 \rightarrow \text{delete}(a)$ 
        (i, v) with  $v \neq \text{null} \rightarrow \text{insert}(v)$ 
        (a, v) with  $v \neq \text{null} \rightarrow \text{assign}(v)$ 
        (r)  $\rightarrow \text{releaseAccess}(\text{index});$  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.

### Procedure *find*

Finding an address in a hash table 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  $r := h.\text{table}[\text{key}(a, l, n)] ; \quad n++ ;$ 
until  $r = \text{null} \vee a = \text{ADR}(r) ;$ 

```

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

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

```

In order to prove correctness, we add between braces instructions that only modify auxiliary variables, like the specification variables  $\mathbf{X}$  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  $\text{Nat}$  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 hash table with open addressing, except for the code at the condition  $r = \text{done}$ . This code is needed for the case that the value at address  $a$  has been 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 2.3.5.

In line 7, the accessed hash table should be valid (see invariants *fi4* and *He4* in Appendix A.1). After *refresh* the local variables *n*, *h* and *l* must be reset, to restart the search in the new hash table. 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 hash table. It is not evident but it has been proved that the linear search of the process executing *find* cannot be violated by other processes (see invariants *Cu9*, *Cu10* and *fi8*), 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 every valid hash table contains at least one entry **null** or **done**. Therefore, the local variable *n* in the procedure *find* never goes beyond the size of the hash table (see invariants *Cu1*, *fi4*, *fi5* and axiom *Ax2*). When the **bound** of the new hash table is tuned properly before use (see invariants *Ne7*, *Ne8*), the hash table will not be updated too frequently, and termination of the procedure *find* can be guaranteed.

### Procedure *delete*

To some extent, 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:          $l := h.size$  ;  $n := 0$  ;
        elseif  $a = ADR(r)$  then
18b:          $\langle$  if  $r = h.table[k]$  then
                 $suc := true$  ;  $h.table[k] := del$  ;
                {  $cnt++$  ; (dS) ;  $Y[k] := del$  }
                end  $\rangle$  ;
        else  $n++$  end ;
        until  $suc \vee r = null$  ;
25:         if  $suc$  then  $h.dels++$  end ;
26:         return  $suc$  .

```

The repetition in this procedure has two ways to terminate. Either deletion fails with  $r = \mathbf{null}$  in 17, or deletion succeeds with  $r = h.table[k]$  in 18b. In the latter case, we have in one atomic statement a double access of the shared variable  $h.table[k]$ . 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 18b, and it is executed precisely once (see invariant *Cn2*), since in 18b the guard  $a = ADR(r)$  implies  $r \neq \mathbf{null}$  (see invariant *del* 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 hash table almost everywhere except that the values it contains are not tagged as *old* or rewritten as **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 hash table (see invariant *Cu4*).

### Procedure *insert*

The procedure for insertion in the table is given below. Basically, it is the standard algorithm for insertion in a hash table with open addressing. Notable is line 28 where the current process finds that the current hash table 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 hash table have been

marked *old*, which is a sign that hash table *h* is outdated, and the new hash table 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 ;
    elsif 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 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 hash table (see invariant *Co3* and the specification of *insert*).

**Procedure assign**

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

```

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

```

**2.3.5 Memory management and concurrent migration**

In this section, we provide the public procedures *getAccess* and *releaseAccess* and the auxiliary procedures *refresh* and *newTable* which are responsible for allocation and deallocation.

We begin with the treatment of memory by providing a model of the heap.

### The model of the heap

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

```

Heap : array Nat of Hashtable := ([Nat] $\perp$ ) ;
H_index : Nat := 1 .

```

So, initially,  $\text{Heap}[i] = \perp$  for all indices  $i$ . The indices of array **Heap** are the pointers to hash tables. We thus simply regard **pointer to Hashtable** as a synonym of *Nat*. Therefore, the notation  $h.\text{table}$  used elsewhere in this chapter stands for  $\text{Heap}[h].\text{table}$ . Since we reserve 0 (to be distinguished from the absent hash table  $\perp$  and the absent value **null**) for the null pointer (i.e.  $\text{Heap}[0] = \perp$ , see invariant *HeI*), we initialize **H\_index**, which is the index of the next hash table, to be 2 instead of 0 or 1. Allocation of memory is modeled in

```

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

```

We assume that *allocate* sets all values in the hash table  $\text{Heap}[\text{H\_index}]$  to **null**, and also sets its fields **size** and **bound** as specified. The variables **occ** and **dels** are set to 0 because the hash table is completely filled with the value **null**.

Deallocation of hash tables is modeled by

```

proc deAlloc(h : Nat) =
  ⟨ assert  $\text{Heap}[h] \neq \perp$  ;  $\text{Heap}[h] := \perp$  ⟩
end .

```

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

### Procedure *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 hash table in the procedure *newTable* (see invariants *pr2*, *pr3* and *nT12*).

The hash table 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 hash table 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 hash table is not valid. Once some process gets hold of one hash table after calling *getAccess*, no process can throw it away until the process releases it (see invariant *rA7*).

### Procedure *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.

```

The test  $h \neq 0$  at 69 is necessary since it is possible that  $h = 0$  at the lines 68 and 69. This occurs e.g. in the following scenario. Assume that process  $p$  is at line 62 with  $index \neq currInd$ , while the number  $i = index$  satisfies  $H[i] = 0$  and  $busy[i] = 0$ . Then process  $p$  increments  $busy[i]$ , calls  $releaseAccess(i)$ , and arrives at 68 with  $h = 0$ .

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 hash table 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.

### Procedure *newTable*

When the current hash table has been used for some time, some actions of the processes may require replacement of this hash table. Procedure *newTable* is called when the number of occupied positions in the current hash table exceeds the **bound** (see lines 28, 44). Procedure *newTable* tries to allocate a new hash table as the successor of the current one. If several processes call *newTable* concurrently, they need to reach consensus on the choice of an index for the next hash table (in line 84). A newly allocated hash table 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 hash table (see invariant *nT8*), of which the **bound** is set greater than  $H[index].\mathbf{bound} - H[index].\mathbf{dels} + 2P$  in order to avoid creating a too small hash table (see invariants *nT6*, *nT7*).

We require the **size** of a hash table to be more than  $\mathbf{bound} + 2P$  because of the following scenario:  $P$  processes find “ $h.\mathbf{occ} > h.\mathbf{bound}$ ” 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  $b_{mE} = \mathit{true}$  ( $b_{mE}$  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 hash table reach one more than its **bound**. Just at that moment,  $P - 1$  processes wake up, increase the **occ** of the current hash table to be  $P - 1$  more, and return to line 30. Since  $P - 1$  processes insert different values in the hash table, after  $P - 1$  processes finish their insertions, the **occ** of the current hash table reaches  $2P - 1$  more than its **bound**.

It may be useful to make **size** larger than  $\mathbf{bound} + 2P$  to avoid too many collisions, e.g. with a constraint  $\mathbf{size} \geq \alpha \cdot \mathbf{bound}$  for some  $\alpha > 1$ . If we did not introduce **dels**, every migration would force the sizes to grow, so that our hash table 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.  $H[i]$ ,  $\mathbf{Heap}[H\_index]$ , 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 hash table 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.

### Procedure *migrate*

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

```

proc migrate() =
    local  $i : 0 \dots 2P$ ;  $h$  : pointer to Hashtable ;  $b$  : Bool ;
94:    $i := \mathbf{next}[\mathbf{index}]$ ;
95:   prot[ $i$ ]++ ;
97:   if  $\mathbf{index} \neq \mathbf{currInd}$  then
98:     prot[ $i$ ]-- ;
    else
99:     busy[ $i$ ]++ ;
100:     $h := \mathbf{H}[i]$  ;
101:    if  $\mathbf{index} = \mathbf{currInd}$  then
        moveContents( $\mathbf{H}[\mathbf{index}]$ ,  $h$ ) ;
103:     $\langle b := (\mathbf{currInd} = \mathbf{index})$  ;
        if  $b$  then  $\mathbf{currInd} := i$  ; {  $\mathbf{Y} := \mathbf{H}[i].\mathbf{table}$  }
        end  $\rangle$  ;
    if  $b$  then
104:     busy[ $\mathbf{index}$ ]-- ;
105:     prot[ $\mathbf{index}$ ]-- ;

```

```

    end end ;
    releaseAccess(i) ;
end end .

```

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

Line 103 contains a compare&swap instruction to update the current hash table 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 hash table (see invariant *mi5*). The increments of `prot[i]` and `busy[i]` here are needed to protect the next hash table. The decrements serve to avoid memory loss.

### Procedure *refresh*

In order to avoid that a delayed process starts migration of an old hash table, 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 hash table and *getAccess* to acquire the present pointer to the current hash table. Otherwise, the process can join the migration.

### Procedure *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 abandoned 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 v = done then
112:      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 it is copied (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 hash table, it becomes **done** in the old hash table in line 117. 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 hash table *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 [24, 46].

### Procedure *moveElement*

The procedure *moveElement* moves a value to the new hash table. 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 hash table (see invariant *Ne19*), since otherwise the main property of open addressing would be violated. In total, four situations can occur in the procedure *moveElement*:

- the current location *k* contains a value with a different address. The process increases *n* to inspect the next location.
- the current location *k* contains a value with the same address. This means that the value has already been copied to the new hash table, the process therefore terminates.
- the current location *k* is an empty slot. The process inserts *v* and returns. If insertion fails, since another process filled the empty slot in between, the search is continued.
- when *index* happens to differ from *currInd*, the entire migration has been completed.

While the current hash table pointer is not updated yet, there exists at least one **null** entry in the new hash table (see invariants *Ne8*, *Ne22* and *Ne23*), hence the local variable *n*

in the procedure *moveElement* never goes beyond the size of the hash table (see invariants *mE3* and *mE8*), and the termination is thus guaranteed.

## 2.4 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 2.4.1. To prove these invariants, we need many other invariants. These are listed in Appendix A.1. In section 2.4.2, we sketch the verification of some of the invariants by informal means. In section 2.4.3, we describe how the theorem prover PVS is used in the verification. As exemplified in 2.4.2, Appendix A.2 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,  $\#V$  denotes the number of elements of  $V$ . If  $b$  is a boolean, then  $\#b = 0$  when  $b$  is false, and  $\#b = 1$  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  $H(\text{currInd}).\text{size}$  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.

### 2.4.1 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:

$$\text{Co1: } pc = 14 \Rightarrow \text{val}(r_{fi}) = rS_{fi}$$

$$\text{Co2: } pc \in \{25, 26\} \Rightarrow \text{suc}_{del} = \text{suc}S_{del}$$

$$\text{Co3: } pc \in \{41, 42\} \Rightarrow \text{suc}_{ins} = \text{suc}S_{ins}$$

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 [52], 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:

$$\text{Cn1: } pc = 14 \Rightarrow \text{cnt}_{fi} = 1$$

$$\text{Cn2: } pc \in \{25, 26\} \Rightarrow \text{cnt}_{del} = 1$$

$$\text{Cn3: } pc \in \{41, 42\} \Rightarrow \text{cnt}_{ins} = 1$$

$$\text{Cn4: } pc = 57 \Rightarrow \text{cnt}_{ass} = 1$$

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

$$\text{No1: } \#\{k \mid k < \text{H\_index} \wedge \text{Heap}(k) \neq \perp\} \leq 2P$$

An important safety property is that no process accesses deallocated memory. Since most procedures perform memory accesses, by means of pointers that are local variables, the proof of this is based on a number of different invariants. Although this is not explicit



in the specification, it has been checked because the theorem prover PVS does not allow access to deallocated memory as this would violate type correctness conditions.

### 2.4.2 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 [33]. 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 2.4.1). According to Appendix A.2, the stability of *Co1* follows from the invariants *Ot3*, *fi1*, *fi10*, which are given in Appendix A.1. 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  $r_{fi} = \mathbf{null} \vee a_{fi} = \mathbf{ADR}(r_{fi})$ . This condition is abbreviated as *Find*( $r_{fi}, a_{fi}$ ). Invariant *fi10* then implies that action 13 has the precondition  $\text{val}(r_{fi}) = rS_{fi}$ , 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.  $\square$

**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.  $\square$

**Proof** of invariant *fi10*. According to Appendix A.2, we only need to use *fi9* and *Ot3*. Let us use the abbreviation  $k = \text{key}(a_{fi}, l_{fi}, n_{fi})$ . 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  $\text{Find}(h_{f_i}.\text{table}[k], a_{f_i})$

After execution of statement 7,  $r_{f_i}$  becomes  $h_{f_i}.\text{table}[k]$ , and  $rS_{f_i}$  becomes  $\mathbf{X}(a_{f_i})$ . By  $f_i9$ , we get  $\text{val}(r_{f_i}) = rS_{f_i}$ . Therefore the validity of  $f_i10$  is preserved.

2. otherwise.

After execution of statement 7,  $r_{f_i}$  becomes  $h_{f_i}.\text{table}[k]$ , which then falsifies the antecedent of  $f_i10$ .  $\square$

**Proof** of invariant  $f_i9$ . According to Appendix A.2, we proved that  $f_i9$  follows from  $\text{Ax}2$ ,  $f_i1$ ,  $f_i3$ ,  $f_i4$ ,  $f_i5$ ,  $f_i8$ ,  $\text{Ha}4$ ,  $\text{He}4$ ,  $\text{Cu}1$ ,  $\text{Cu}9$ ,  $\text{Cu}10$ , and  $\text{Cu}11$ . We abbreviate  $\text{key}(a_{f_i}, l_{f_i}, n_{f_i})$  as  $k$ . We deduce  $h_{f_i} = \text{H}(\text{index})$  from  $f_i4$ ,  $\text{H}(\text{index})$  is not  $\perp$  from  $\text{He}4$ , and  $k$  is below  $\text{H}(\text{index}).\text{size}$  from  $\text{Ax}2$ ,  $f_i4$  and  $f_i3$ . We split the proof into two cases:

1.  $\text{index} \neq \text{currInd}$ : By  $\text{Ha}4$ , it follows that  $\text{H}(\text{index}) \neq \text{H}(\text{currInd})$ . Hence from  $\text{Cu}1$ , we obtain  $h_{f_i}.\text{table}[k] = \mathbf{done}$ , which falsifies the antecedent of  $f_i9$ .
2.  $\text{index} = \text{currInd}$ : By premise  $\text{Find}(h_{f_i}.\text{table}[k], a_{f_i})$ , we know that  $h_{f_i}.\text{table}[k] \neq \mathbf{done}$  because of  $f_i1$ . By  $\text{Cu}9$  and  $\text{Cu}10$ , we obtain  $\text{val}(h_{f_i}.\text{table}[k]) = \text{val}(\mathbf{Y}[k])$ . Hence it follows that  $\text{Find}(\mathbf{Y}[k], a_{f_i})$ . Using  $f_i8$ , we obtain

$$\forall m < n_{f_i} : \neg \text{Find}(\mathbf{Y}[\text{key}(a_{f_i}, \text{curSize}, m)], a_{f_i})$$

We get  $n_{f_i}$  is below  $\text{curSize}$  because of  $f_i5$ . By  $\text{Cu}11$ , we conclude

$$\mathbf{X}(a_{f_i}) = \text{val}(h_{f_i}.\text{table}[k])$$

$\square$

### 2.4.3 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 [53], 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 is proved by a PVS *Theorem* 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 eleven preconditions to imply well-definedness: e.g. in *find7*, the hash table must be non-NIL and  $\ell$  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  $(s1, s2) \in \text{step}$  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)
```

## 2.5 Correctness (Progress)

In this section, we prove that our algorithm is lock-free, and that it is wait-free for several subtasks. However, the proof was not checked with PVS.

Recall that an algorithm is called *lock-free* if always at least some 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. We say that an operation is *wait-free* if any process involved in that operation is guaranteed to complete it in a finite number of its own steps, regardless of the (in)activity of other processes.

### 2.5.1 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*. The loop of *moveContents* is clearly 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,  $to.table[k] \neq \mathbf{null}$  is stable, the loop of *moveElement* is also bounded. This concludes the sketch that *migrate* is wait-free.

### 2.5.2 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  $\mathbf{next}(index) \neq 0$  is stable while process  $p$  stays in the region  $[77, 84]$ , once the condition  $\mathbf{next}(index) \neq 0$  holds, process  $p$  will exit *newTable* in a few rounds.

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

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

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

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

and consider the sum  $\sum_{i=1}^{2P} (\#(prSet1(i)) + \#(prSet2(i)))$ . 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  $\#(prSet1(j)) + \#(prSet2(j)) = 0$  and  $j \neq index.p$ . By the invariant

$$\begin{aligned} pr1: \quad \text{prot}[j] &= \#(prSet1(j)) + \#(prSet2(j)) + \#(\text{currInd} = j) \\ &\quad + \#(\text{next}(\text{currInd}) = j) \end{aligned}$$

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

While  $\text{currInd}$  is constant, no process can modify  $\text{prot}[j]$  for  $j \neq \text{currInd}$  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.

### 2.5.3 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. In that case, however, the other processes repeatedly succeed. It follows that *getAccess*, *refresh*, and *newTable* are lock-free.

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 hash tables.

According to the invariants *fi5*, *de8*, *in8* and *as6*, the primary procedures *find*, *delete*, *insert*, *assign* are loops bounded by  $n \leq h.\text{size}$ , and  $n$  is only reset to 0 during migration. If  $n$  is not reset to 0, it is incremented or stays constant. Indeed, the atomic **if** statements in 18b, 35b, and 50b have no **else** parts. In *delete* and *assign*, it is therefore possible that  $n$  stays constant without termination of the loop. Since *assign* can modify non-**null** elements of the table, it follows that *delete* and *assign* are not wait-free. This unbounded fruitless activity is possible only when *assign* actions of other processes repeatedly succeed. It follows that the primary procedures are lock-free. This concludes the argument that the system is lock-free.

## 2.6 Conclusions

Lock-free shared data objects are inherently resilient to halting failures and permit maximum parallelism. We have presented a new practical, lock-free algorithm for concurrently accessible hash tables, 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 hash table 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 hash table. 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 only be taken as a rough indicator, since 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 huge 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 years excluding the effort in determining the algorithm and writing the documentation. The whole proof contains around 200 invariants. It takes a 1 Ghz 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.

It may well be possible to simplify the proof and reduce the number of invariants slightly, but we did not work on this. The complete version of the PVS specifications and the whole proof scripts can be found at [33]. Note that we simplified some definitions in this chapter for the sake of presentation.



## Chapter 3

# A formal reduction for lock-free parallel algorithms

This chapter has been published as [23].

### 3.1 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 behaviors of the overall system, using synchronization primitives such as semaphores, monitors, guarded statements, mutex locks, etc. Consequently the operations of different processes on a shared data structure should appear to be serialized: if two operations execute simultaneously, the system guarantees the same result as if one of them is arbitrarily executed before the other.

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 [9, 6, 54, 28, 29, 51] have proposed techniques for designing lock-free implementations. The basis of these techniques is using some synchronization primitives such as *compare-and-swap* (CAS), or *Load-linked* (LL)/*store-conditional* (SC).

Typically, the implementation of the synchronization operations is left to the designer, who has to decide how much of the functionality to implement in software using system libraries. The high-level specification gives lots of freedom about how a result is obtained. It is constructed in some mechanical way that guarantees its correctness and then the required conditions are automatically satisfied [13]. We reason about a high-level specification of a system, with a large grain of atomicity, and hope to deduce an implementation, a low-level specification, which must be fine grained enough to be translated into a computer program that has all important properties of the high-level specification.

However, the correctness properties of an implementation are seldom easy to verify. Our previous work (see chapter 2) shows that a proof may require huge amounts of effort, time, or skill. We therefore develop a reduction theorem that enables us to reason about a lock-free program to be designed on a higher level than the synchronization primitives. The

reduction theorem is 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 theorem, fewer invariants are required and some invariants are easier to discover and easier to formulate, without considering the internal structure of the final implementation. In particular, nested loops in the algorithm may be treated as one loop at a time.

### 3.2 Lock-free transformation

The machine architecture that we have in mind is based on modern shared-memory multi-processors that can access a common shared address space. There can be several processes running on a single processor. Let us assume there are  $P$  ( $\geq 1$ ) concurrently executing sequential processes.

Synchronization primitives *LL* and *SC*, proposed by Jensen et al. [42], have found widespread acceptance in modern processor architectures (e.g. MIPS II, PowerPC and Alpha architectures). They are a pair of instructions, closely related to the *CAS*, and together implement an atomic Read/Write cycle. Instruction *LL* first reads a memory location, say  $X$ , and marks it as “reserved” (not “locked”). If no other processor changes the contents of  $X$  in between, the subsequent *SC* operation of the same processor succeeds and modifies the value stored; otherwise it fails. There is also a validate instruction *VL*, used to check whether  $X$  was not modified since the corresponding *LL* instruction was executed. Implementing *VL* should be straightforward in an architecture that already supports *SC*. Note that the implementation does not access or manipulate  $X$  other than by means of *LL/SC/VL*. Moir [57] showed that *LL/SC/VL* can be constructed on any system that supports either *LL/SC* or *CAS*. A shared variable  $X$  only accessed by *LL/SC/VL* operations can be regarded as a variable that has an associated shared set of process identifiers  $V.X$ , which is initially empty. The semantics of *LL*, *VL* and *SC* are given by equivalent atomic statements below.

```
proc LL(ref  $X$ : val): val =
  <   $V.X := V.X \cup \{self\}$ ; return  $X$ ; >
```

```
proc VL(ref  $X$ : val): boolean =
  <  return ( $self \in V.X$ ) >
```

```

proc SC(ref X: val; in Y: val): boolean =
  ⟨  if self ∈ V.X then V.X := ∅; X := Y; return true
    else return false; fi ⟩

```

where *self* is the process identifier of the acting process.

At the cost of copying an object's data before an operation, Herlihy [28] introduced a general methodology to transfer a sequential implementation of any data structure into a lock-free synchronization by means of synchronization primitives *LL* and *SC*. A process that needs access to a shared object pointed by *X* performs a loop of the following steps: (1) read *X* using an *LL* operation to gain access to the object's data area; (2) make a private copy of the indicated version of the object (this action need not be atomic); (3) perform the desired operation on the private copy to make a new version; (4) finally, call an *SC* operation on *X* to attempt to swing the pointer from the old version to the new version. The *SC* operation will fail when some other process has modified *X* since the *LL* operation, in which case the process has to repeat these steps until consistency is satisfied. The algorithm is non-blocking because at least one out of every *P* attempts must succeed within finite time. Of course, a process might always lose to some faster process, but this is often unlikely in practice.

### 3.3 Reduction

We assume a universal set  $\mathcal{V}$  of typed variables, which is called the *vocabulary*. A state *s* is a type-consistent interpretation of  $\mathcal{V}$ , mapping variables  $v \in \mathcal{V}$  to values  $s[v]$ . We denote by  $\Sigma$  the set of all states. If  $\mathcal{C}$  is a command, we denote by  $\mathcal{C}_p$  the transition  $\mathcal{C}$  executed by process *p*, and  $s[\mathcal{C}_p]t$  indicates that in state *s* process *p* can do a step  $\mathcal{C}$  that establishes state *t*. When discussing the effect of a transition  $\mathcal{C}_p$  from state *s* to state *t* on a variable *v*, we abbreviate  $s[v]$  to *v* and  $t[v]$  to *v'*. We use the abbreviation  $Pres(V)$  for  $\bigwedge_{v \in V} (v' = v)$  to denote that all variables in the set *V* are preserved by the transition. Every private variable name can be extended with the suffix “.” + “process identifier”. We sometimes use indentation to eliminate parentheses.

### 3.3.1 Observed Specification

In practice, the specification of systems is concerned rather with externally visible behavior than computational feasibility. We assume that all levels of specifications under consideration have the same observable state space  $\Sigma_0$ , and are interpreted by their observation functions  $\Pi: \Sigma \rightarrow \Sigma_0$ . Every specification can be modeled as a five-tuple  $(\Sigma, \Pi, \Theta, \mathcal{N}, \mathcal{L})$  where  $(\Sigma, \Theta, \mathcal{N})$  is the *transition system* [53] and  $\mathcal{L}$  is the supplementary property of the system (i.e. a predicate on  $\Sigma^\omega$ ).

The supplementary constraint  $\mathcal{L}$  is imposed since the transition system only specifies safety requirements and has no kind of fairness conditions or liveness assumptions built into it. Since, in reality, a stuttering step might actually perform modifications to some internal variables in internal states, we do allow stuttering transitions (where the state does not change) and the next-state relation is therefore reflexive. A finite or infinite sequence of states is defined to be an *execution* of system  $(\Sigma, \Pi, \Theta, \mathcal{N}, \mathcal{L})$  if it satisfies initial predicate  $\Theta$  and the next-state relation  $\mathcal{N}$  but not necessarily the requirements of the supplementary property  $\mathcal{L}$ . We define a *behavior* to be an infinite execution that satisfies the supplementary property  $\mathcal{L}$ . A (concrete) specification  $\mathcal{S}_c$  *implements* an (abstract) specification  $\mathcal{S}_a$  iff every externally visible behavior allowed by  $\mathcal{S}_c$  is also allowed by  $\mathcal{S}_a$ . We write  $Beh(\mathcal{S})$  to denote the set of behaviors of system  $\mathcal{S}$ .

### 3.3.2 Refinement mappings

A *refinement mapping* from a lower-level specification  $\mathcal{S}_c = (\Sigma_c, \Pi_c, \Theta_c, \mathcal{N}_c, \mathcal{L}_c)$  to a higher-level specification  $\mathcal{S}_a = (\Sigma_a, \Pi_a, \Theta_a, \mathcal{N}_a, \mathcal{L}_a)$ , written  $\varphi: \mathcal{S}_c \sqsubseteq \mathcal{S}_a$ , is a mapping  $\varphi: \Sigma_c \rightarrow \Sigma_a$  that satisfies:

1.  $\varphi$  preserves the externally visible state component:  $\Pi_a \circ \varphi = \Pi_c$ .
2.  $\varphi$  is a *simulation*, denoted  $\varphi: \mathcal{S}_c \preceq \mathcal{S}_a$ :
  - ①  $\varphi$  takes initial states into initial states:  $\Theta_c \Rightarrow \Theta_a \circ \varphi$ .
  - ②  $\mathcal{N}_c$  is mapped by  $\varphi$  into a transition (possibly stuttering) allowed by  $\mathcal{N}_a$ :  
 $\mathcal{Q} \wedge \mathcal{N}_c \Rightarrow \mathcal{N}_a \circ \varphi$ , where  $\mathcal{Q}$  is an invariant of  $\mathcal{S}_c$ .
3.  $\varphi$  maps behaviors allowed by  $\mathcal{S}_c$  into behaviors that satisfy  $\mathcal{S}_a$ 's supplementary property:  $\forall \sigma \in Beh(\mathcal{S}_c): \mathcal{L}_a(\varphi(\sigma))$ .

Below we need to exploit the fact that the simulation only quantifies over all reachable states of the lower-level system, not all states. We therefore explicitly allow an invariant  $\mathcal{Q}$  in condition 2 ②. The following theorem is stated in [1].

**Theorem 3.3.1** *If there exists a refinement mapping from  $\mathcal{S}_c$  to  $\mathcal{S}_a$ , then  $\mathcal{S}_c$  implements  $\mathcal{S}_a$ .*

Refinement mappings give us the ability 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]. Essentially, the reduction theorem allows us to design and verify the program on a higher level of abstraction. The big advantage is that substantial pieces of the concrete program can be dealt with as atomic statements on the higher level.

The refinement relation is transitive, which means that we don't have to reduce the implementation in one step, but can proceed from the implementation to the specification through a series of smaller steps.

### 3.3.3 Correctness

The safety properties satisfied by the program are completely determined by the initial predicate and the next-state relation. This is described by Theorem 3.3.2, which can be easily verified.

**Theorem 3.3.2** *Let  $\mathcal{P}_c$  and  $\mathcal{P}_a$  be safety properties for  $\mathcal{S}_c$  and  $\mathcal{S}_a$  respectively. The verification of a concrete judgment  $(\Sigma_c, \Theta_c, \mathcal{N}_c) \models \mathcal{P}_c$  can be reduced to the verification of an abstract judgment  $(\Sigma_a, \Theta_a, \mathcal{N}_a) \models \mathcal{P}_a$ , if we can exhibit a simulation  $\varphi$  mapping from  $\Sigma_c$  to  $\Sigma_a$  that satisfies  $\mathcal{P}_a \circ \varphi \Rightarrow \mathcal{P}_c$ .*

We make a distinction between safety and liveness properties (see section 1.2 for the proof schemes). The proof of liveness relies on the fairness conditions associated with a specification. The purpose for fairness conditions is to rule out executions where the system idles indefinitely with control at some internal point of a procedure and with some transition of that procedure enabled. Fairness arguments usually depend on safety properties of the system.

### 3.4 A lock-free pattern

We propose a pattern that can be universally employed for a lock-free construction in order to synchronize access to a shared node of **nodeType**. The interface  $\mathcal{S}_a$  is shown in Fig. 3.1, where the following statements are taken as a schematic representation of segments of code:

1. *noncrit*(**ref** *pub*: **aType**, *priv*: **bType**; **in** *tm*: **cType**; **out** *x*:  $1 \dots N$ ) : representing an atomic non-critical activity on variables *pub* and *priv* according to the value of *tm*, and choosing an index *x* of a shared node to be accessed.
2. *guard*(**in** *X*: **nodeType**, *priv*: **bType**) a non-atomic boolean test on the variable *X* of **nodeType**. It may depend on private variable *priv*.
3. *com*(**ref** *X*: **nodeType**; **in** *priv*: **bType**; **out** *tm*: **cType**) : a non-atomic action on the variable *X* of **nodeType** and private variable *tm*. It is allowed to inspect private variable *priv*.

The action enclosed by angular brackets  $\langle \dots \rangle$  is defined as atomic. The private variable *x* is intended only to determine the node under consideration, the private variable *tm* is intended to hold the result of the critical computation *com*, if executed. By means of Herlihy's methodology, we give a lock-free implementation  $\mathcal{S}_{ll/sc}$  of interface  $\mathcal{S}_a$  in Fig. 3.2. In the implementation, we use some other schematic representations of segments of code, which are described as follows:

4. *read*(**ref** *X*: **nodeType**, **in** *Y*: **nodeType**) : a non-atomic read operation that reads the value from the variable *Y* of **nodeType** to the variable *X* of **nodeType**, and does nothing else. If *Y* is modified during *read*, the resulting value of *X* is unspecified but type correct, and no error occurs.
5. *LL*, *SC* and *VL*: atomic actions as we defined before.

Typically, we are not interested in the internal details of these schematic commands but in their behavior with respect to lock-freedom. In  $\mathcal{S}_{ll/sc}$ , we declare *P* extra shared nodes for private use (one for each process). Array *indir* acts as pointers to shared nodes. *node*[*mp.p*] can always be taken as a “private” node (other processes can read but not modify the content of the node) of process *p* though it is declared publicly. If some other process successfully

<p><b>Constant</b>  <math>P = \text{number of processes}; N = \text{number of nodes};</math></p> <p><b>Shared variable</b>  <math>\text{pub: aType}; \text{Node: array } [1 \dots N] \text{ of nodeType};</math></p> <p><b>Private variable</b>  <math>\text{priv: bType}; \text{pc: } \{a_1, a_2\}; x: 1 \dots N; \text{tm: cType};</math></p> <p><b>Program</b>  <b>loop</b>  <math>a_1: \quad \text{noncrit}(\text{pub}, \text{priv}, \text{tm}, x);</math>  <math>a_2: \quad \langle \text{if guard}(\text{Node}[x], \text{priv}) \text{ then } \text{com}(\text{Node}[x], \text{priv}, \text{tm}); \text{fi}; \rangle</math>  <b>end.</b></p> <p><b>Initial conditions</b>  <math>\Theta_a: \quad \forall p: 1 \dots P: \text{pc}_p = a_1</math></p> <p><b>Liveness</b>  <math>\mathcal{L}_a: \quad \Box(\text{pc}_p = a_2 \longrightarrow \Diamond \text{pc}_p = a_1)</math></p>
--

Figure 3.1: Interface  $\mathcal{S}_a$ 

<p><b>Constant</b>  <math>P = \text{number of processes}; N = \text{number of nodes};</math></p> <p><b>Shared variable</b>  <math>\text{pub: aType}; \text{node: array } [1 \dots N+P] \text{ of nodeType};</math>  <math>\text{indir: array } [1 \dots N] \text{ of } 1 \dots N+P;</math></p> <p><b>Private variable</b>  <math>\text{priv: bType}; \text{pc: } [c_1 \dots c_7];</math>  <math>x: 1 \dots N; \text{mp}, m: 1 \dots N+P; \text{tm}, \text{tm1: cType};</math></p> <p><b>Program</b>  <b>loop</b>  <math>c_1: \quad \text{noncrit}(\text{pub}, \text{priv}, \text{tm}, x);</math>  <b>loop</b>  <math>c_2: \quad m := \text{LL}(\text{indir}[x]);</math>  <math>c_3: \quad \text{read}(\text{node}[\text{mp}], \text{node}[m]);</math>  <math>c_4: \quad \text{if guard}(\text{node}[\text{mp}], \text{priv}) \text{ then}</math>  <math>c_5: \quad \quad \text{com}(\text{node}[\text{mp}], \text{priv}, \text{tm1});</math>  <math>c_6: \quad \quad \text{if } \text{SC}(\text{indir}[x], \text{mp}) \text{ then } \text{mp} := m; \text{tm} := \text{tm1}; \text{break}; \text{fi};</math>  <math>c_7: \quad \quad \text{elseif } \text{VL}(\text{indir}[x]) \text{ then break}; \text{fi}; \text{end};</math>  <b>end.</b></p> <p><b>Initial conditions</b>  <math>\Theta_{ll/sc}: \quad (\forall p: 1 \dots P: \text{pc}_p = c_1 \wedge \text{mp}_p = N+p) \wedge (\forall i: 1 \dots N: \text{indir}[i] = i)</math></p> <p><b>Liveness</b>  <math>\mathcal{L}_{ll/sc}: \quad \Box(\text{pc}_p = c_2 \longrightarrow \Diamond \text{pc}_p = c_1)</math></p>
---

Figure 3.2: Lock-free implementation  $\mathcal{S}_{ll/sc}$  of  $\mathcal{S}_a$



updates a shared node while an active process  $p$  is copying the shared node to its “private” node, process  $p$  will restart the inner loop, since its private view of the node is not consistent anymore. After the assignment  $mp := m$  at line c6, the “private” node becomes shared and the node shared previously (which contains the old version) becomes “private”.

Formally, we introduce  $N_c$  as the relation corresponding to command *noncrit* on  $(\mathbf{aType} \times \mathbf{bType} \times \mathbf{cType}, \mathbf{aType} \times \mathbf{bType} \times 1 \dots N)$ ,  $P_g$  as the predicate computed by *guard* on  $\mathbf{nodeType} \times \mathbf{bType}$ ,  $R_c$  as the relation corresponding to *com* on  $(\mathbf{nodeType} \times \mathbf{bType}, \mathbf{nodeType} \times \mathbf{cType})$ , and define

$$\begin{aligned}\Sigma_a &\triangleq (\mathbf{Node}[1 \dots N], \mathbf{pub}) \times (pc, x, \mathbf{priv}, tm)^P, \\ \Sigma_{ll/sc} &\triangleq (\mathbf{node}[1 \dots N+P], \mathbf{indir}[1 \dots N], \mathbf{pub}) \times (pc, x, mp, m, \mathbf{priv}, tm, tm1)^P, \\ \Pi_a(\Sigma_a) &\triangleq (\mathbf{Node}[1 \dots N], \mathbf{pub}), \quad \Pi_{ll/sc}(\Sigma_{ll/sc}) \triangleq (\mathbf{node}[\mathbf{indir}[1 \dots N]], \mathbf{pub}), \\ \mathcal{N}_a &\triangleq \bigvee_{0 \leq i \leq 2} \mathcal{N}_{ai}, \quad \mathcal{N}_{ll/sc} \triangleq \bigvee_{1 \leq i \leq 7} \mathcal{N}_{ci},\end{aligned}$$

The transitions of the abstract system can be described:  $\forall s, t: \Sigma_a, p: 1 \dots P$ :

$$\begin{aligned}s\llbracket(\mathcal{N}_{a0})_p\rrbracket t &\triangleq s = t \quad (\text{to allow stuttering}) \\ s\llbracket(\mathcal{N}_{a1})_p\rrbracket t &\triangleq pc.p = a1 \wedge pc'.p = a2 \wedge Pres(\mathcal{V} - \{\mathbf{pub}, \mathbf{priv}.p, pc.p, x.p\}) \\ &\quad \wedge ((\mathbf{pub}, \mathbf{priv}.p, tm.p), (\mathbf{pub}, \mathbf{priv}.p, x.p)') \in N_c \\ s\llbracket(\mathcal{N}_{a2})_p\rrbracket t &\triangleq pc.p = a2 \wedge pc'.p = a1 \wedge (P_g(\mathbf{Node}[x], \mathbf{priv}.p) \\ &\quad \wedge ((\mathbf{Node}[x], \mathbf{priv}.p), (\mathbf{Node}[x], tm.p)') \in R_c \\ &\quad \wedge Pres(\mathcal{V} - \{pc.p, \mathbf{Node}[x], tm.p\}) \\ &\quad \vee \neg P_g(\mathbf{Node}[x], \mathbf{priv}.p) \wedge Pres(\mathcal{V} - \{pc.p\})).\end{aligned}$$

The transitions of the concrete system can be described in the same way. Here we only provide the description of the step that starts in c6:  $\forall s, t: \Sigma_{ll/sc}, p: 1 \dots P$ :

$$\begin{aligned}s\llbracket(\mathcal{N}_{c6})_p\rrbracket t &\triangleq pc.p = c6 \wedge (p \in V.\mathbf{indir}[x.p] \\ &\quad \wedge pc'.p = c1 \wedge (\mathbf{indir}[x.p])' = mp.p \wedge mp'.p = m.p \\ &\quad \wedge tm'.p = tm1.p \wedge (V.\mathbf{indir}[x.p])' = \emptyset \\ &\quad \wedge Pres(\mathcal{V} - \{pc.p, \mathbf{indir}[x.p], mp.p, tm.p, V.\mathbf{indir}[x.p]\}) \\ &\quad \vee p \notin V.\mathbf{indir}[x.p] \wedge pc'.p = c2 \wedge Pres(\mathcal{V} - \{pc.p\})).\end{aligned}$$

### 3.4.1 Simulation

According to Theorem 3.3.2, the verification of a safety property of concrete system  $\mathcal{S}_{ll/sc}$  can be reduced to the verification of the corresponding safety property of abstract system  $\mathcal{S}_a$  if we can exhibit the existence of a simulation between them.

**Theorem 3.4.1** *The concrete system  $\mathcal{S}_{ll/sc}$  defined in Fig. 3.2 is simulated by the abstract system  $\mathcal{S}_a$  defined in Fig. 3.1, that is,  $\exists \varphi : \mathcal{S}_{ll/sc} \preceq \mathcal{S}_a$ .*

**Proof:** We prove Theorem 3.4.1 by providing a simulation. The simulation function  $\varphi$  is defined by showing how each component of the abstract state (i.e. state of  $\Sigma_a$ ) is generated from components in the concrete state (i.e. state of  $\Sigma_{ll/sc}$ ). We define  $\varphi$  : the concrete location  $c1$  is mapped to the abstract location  $a1$ , while all other concrete locations are mapped to  $a2$ ; the concrete shared variable `node[indir[x]]` is mapped to the abstract shared variable `Node[x]`, and the remaining variables are all mapped to the identity of the variables occurring in the abstract system.

The assertion that the initial condition  $\Theta_{ll/sc}$  of the concrete system implies the initial condition  $\Theta_a$  of the abstract system follows easily from the definitions of  $\Theta_{ll/sc}$ ,  $\Theta_a$  and  $\varphi$ .

The central step in the proof of simulation is to prove that every atomic step of the concrete system is simulated by an atomic step of the abstract system. We therefore need to associate each transition in the concrete system with the transition in the abstract system.

It is easy to see that the concrete transition  $\mathcal{N}_{c1}$  is simulated by  $\mathcal{N}_{a1}$  and that  $\mathcal{N}_{c2}$ ,  $\mathcal{N}_{c3}$ ,  $\mathcal{N}_{c4}$ ,  $\mathcal{N}_{c5}$ ,  $\mathcal{N}_{c6}$  with precondition “ $self \notin V.indir[x.self]$ ”, and  $\mathcal{N}_{c7}$  with precondition “ $self \notin V.indir[x.self]$ ” is simulated by a stuttering step  $\mathcal{N}_{a0}$  in the abstract system. E.g. we prove that  $\mathcal{N}_{c6}$  executed by any process  $p$  with precondition “ $p \notin V.indir[x.p]$ ” is simulated by a stuttering step in the abstract system. By the mechanism of *SC*, an active process  $p$  will only modify its program counter  $pc.p$  from  $c6$  to  $c2$  when executing  $\mathcal{N}_{c6}$  with precondition “ $p \notin V.indir[x.p]$ ”. According to the mapping of  $\varphi$ , both concrete locations  $c6$  and  $c2$  are mapped to abstract location  $a2$ . Since the mappings of the pre-state and the post-state to the abstract system are identical,  $\mathcal{N}_{c6}$  executed by process  $p$  with precondition “ $p \notin V.indir[x.p]$ ” is simulated by the stuttering step  $\mathcal{N}_{a0}$  in the abstract system.

The proof for the simulations of the remaining concrete transitions is less obvious. Since simulation applies only to transitions taken from a reachable state, we postulate the following invariants in the concrete system  $\mathcal{S}_{ll/sc}$ :

$$Q1: (p \neq q \Rightarrow mp.p \neq mp.q) \wedge (indir[y] \neq mp.p) \\ \wedge (y \neq z \Rightarrow indir[y] \neq indir[z])$$

$$Q2: pc.p = c6 \wedge p \in V.indir[x.p] \\ \Rightarrow ((node[m.p], priv.p), (node[mp.p], tm1.p)) \in R_c$$

$$Q3: pc.p = c7 \wedge p \in V.indir[x.p] \Rightarrow \neg P_g(node[m.p], priv.p)$$

$$\begin{aligned}
Q4: \quad & pc.p \in [c3 \dots c7] \wedge p \in V.\text{indir}[x.p] \Rightarrow m.p = \text{indir}[x.p] \\
Q5: \quad & pc.p \in \{c4, c5\} \wedge p \in V.\text{indir}[x.p] \Rightarrow \text{node}[m.p] = \text{node}[mp.p] \\
Q6: \quad & pc.p = \{c5, c6\} \Rightarrow P_g(\text{node}[mp.p], \text{priv}.p)
\end{aligned}$$

In the invariants, the free variables  $p$  and  $q$  range over  $1 \dots P$ , and the free variables  $y$  and  $z$  range over  $1 \dots N$ . Invariant  $Q1$  implies that, for any process  $q$ ,  $\text{node}[mp.q]$  can be indeed treated as a “private” node of process  $q$  since only process  $q$  can modify that. Invariant  $Q4$  reflects the mechanism of the synchronization primitives  $LL$  and  $SC$ .

With the help of those invariants above, we have proved that  $\mathcal{N}_{c6}$  and  $\mathcal{N}_{c7}$  executed by process  $p$  with precondition “ $p \in V.\text{indir}[x.p]$ ” are simulated by the abstract step  $\mathcal{N}_{a2}$  in the abstract system. For reasons of space we refer the interested reader to [34] for the complete mechanical proof.  $\square$

### 3.4.2 Refinement

Recall that not all simulation relations are refinement mappings. According to the formalism of the reduction, it is easy to verify that  $\varphi$  preserves the externally visible state component. For the refinement relation we also need to prove that the simulation  $\varphi$  maps behaviors allowed by  $\mathcal{S}_{ll/sc}$  into behaviors that satisfy  $\mathcal{S}_a$ ’s liveness property, that is,  $\forall \sigma \in \text{Beh}(\mathcal{S}_{ll/sc}) : \mathcal{L}_a(\varphi(\sigma))$ . Since  $\varphi$  is a simulation, we deduce

$$\begin{aligned}
\sigma \models \mathcal{L}_{ll/sc} &\equiv \sigma \models \Box(pc = c2 \longrightarrow \Diamond pc = c1) \\
&\Rightarrow \sigma \models \Box(pc \in [c2 \dots c7] \longrightarrow \Diamond pc = c1) \\
&\Rightarrow \varphi(\sigma) \models \Box(pc = a2 \longrightarrow \Diamond pc = a1) \\
&\equiv \mathcal{L}_a(\varphi(\sigma))
\end{aligned}$$

Consequently, we have our main reduction theorem:

**Theorem 3.4.2** *The abstract system  $\mathcal{S}_a$  defined in Fig. 3.1 is implemented by the concrete system  $\mathcal{S}_{ll/sc}$  defined in Fig. 3.2, that is,  $\exists \varphi : \mathcal{S}_{ll/sc} \sqsubseteq \mathcal{S}_a$ .*

The liveness property  $\mathcal{L}_{ll/sc}$  of concrete system  $\mathcal{S}_{ll/sc}$  can also be proved under the assumption of the strong fairness conditions and the following assumption:

$$\begin{aligned}
&\Box (\Box pc.p \in [c2 \dots c7] \wedge \Box \Diamond p \in V.\text{indir}[x.p] \\
&\longrightarrow \Diamond (pc.p = c6 \vee pc.p = c7) \wedge p \in V.\text{indir}[x.p]).
\end{aligned}$$

The additional assumption indicates that for every process  $p$ , when process  $p$  remains in the loop from  $c2$  to  $c7$  and executes  $c2$  infinitely often, it will eventually succeed in reaching  $c6$  or  $c7$  with precondition “ $p \in V.\text{indir}[x.p]$ ”.

### 3.5 Large object

To reduce the overhead of failing non-blocking operations, Herlihy [28] proposes an exponential back-off policy to reduce useless parallelism, which is caused by failing attempts. A fundamental problem with Herlihy’s methodology is the overhead that results from making complete copies of the entire object ( $c3$  in Fig. 3.2) even if only a small part of an object has been changed. For a large object this may be excessive.

We therefore propose two alternatives given in Fig. 3.3. For both algorithms the fields of the object are divided into  $W$  disjoint logical groups such that if one field is modified then other fields in the same group may be modified simultaneously. We introduce an additional field **ver** in **nodeType** to attach version numbers to each group to avoid unnecessary copying. We assume all version numbers attached to groups are positive. As usual with version numbers, we assume that they can be sufficiently large. We increment the version number of a group each time we modify at least one member in the group.

All schematic representations of segments of code that appear in Fig. 3.3 are the same as before, except

3. **com(ref X: nodeType; in g: 1...W, priv: bType; out tm: cType)** : performs an action on group  $g$  of the variable  $X$  of **nodeType** instead of on the whole object  $X$ .
4. **read(ref X: nodeType; in Y: nodeType, g: 1...W)** : only reads the value from group  $g$  of node  $Y$  to the same group of node  $X$ .

The relations corresponding to these schematic commands are adapted accordingly.

In the first implementation,  $mp$  becomes an array used to record pointers to private copies of shared nodes. In total we declare  $N \times P$  extra shared nodes for private use (one for each process and each node). Note that **node**[ $mp[x].p$ ] can be taken as a “private” node of process  $p$  though it is declared publicly. Array **indir** continues to act as pointers to shared nodes.

At the moment that process  $p$  reads group  $i.p$  of **node**[ $m.p$ ] (line  $l5$ ), process  $p$  may observe the object in an inconsistent state (i.e. the read value is not the current or historical

```

Constant
  P = number of processes; N = number of nodes;
  W = number of groups;
  K = N + N × P;          (*II:    K = N + P;*)

Type
  nodeType: record =
    val: array [1...W] of valType;
    ver: array [1...W] of posnat;
  end

Shared variables
  pub: aType;
  node: array [1...K] of nodeType;
  indir: array [1...N] of 1...K;

Private variables
  priv: bType; pc: l1...l11;
  x: 1...N; m: 1...K;
  mp: array [1...N] of 1...K;  (*II:    mp: 1...K;*)
  new: array [1...W] of posnat; old: array [1...W] of nat;
  g: 1...W; tm, tm1: cType; i: nat;

Program:
  loop
l1:    noncrit(pub, priv, tm, x);
        choose group g to be modified;
        old := node[mp[x]].ver;          (*II:    old := λ(i: 1...W) : 0;*)
        (*II:    replace all mp[x] below by mp *)
        loop
l2:    m := LL(indir[x]);
l3:    i := 1
l4:    while i ≤ W do
            new[i] := node[m].ver[i];
            if new[i] ≠ old[i] then
l5:    read(node[mp[x]], node[m], i); old[i] := 0;
l6:    if ¬VL(indir[x]) then goto l2; fi;
l7:    node[mp[x]].ver[i] := new[i]; old[i] := new[i]; fi;
            i++; end;
l8:    if guard(node[mp[x]], priv) then
l9:    com(node[mp[x]], g, priv, tm1); old[g] := 0;
            node[mp[x]].ver[g] := new[g] + 1;
l10:    if SC(indir[x], mp[x]) then
                mp[x] := m; tm := tm1; break; fi;
l11:    elseif VL(indir[x]) then break; fi; end;
        end.

```

Figure 3.3: Lock-free implementation I (\* implementation II \*) for large objects

view of the shared object) since pointer  $m.p$  may have been redirected to some private copy of the node by some faster process  $q$ , which has increased the modified group's version number (in lines  $l9$  and  $l10$ ). When process  $p$  restarts the loop, it will get higher version numbers at the array  $new$ , and only needs to reread the modified groups, whose  $new$  version numbers differ from their  $old$  version numbers. Excessive copying can be therefore prevented. Line  $l6$  is used to check if the read value of a group is consistent with the version number.

The first implementation is fast for an application that often changes only a small part of the object. However, the space complexity is substantial because  $P + 1$  copies of each node are maintained and copied back and forth. Sometimes, a trade-off is chosen between space and time complexity. We therefore adapt it to our second lock-free algorithm for large objects (shown in Fig. 3.3 also) by substituting all statements enclosed by  $(* \dots *)$  for the corresponding statements in the first version. As we did for small objects, we use only one extra copy of a node for each process in the second implementation.

In the second implementation, since the private copy of a node may belong to some other node, a process first initializes all elements of  $old$  to be zero (line  $l1$ ) before accessing an object, to force the process to make a complete copy of the entire object for the first attempt. The process then only needs to copy part of the object from the second attempt on. The space complexity for our second version saves  $(N - 1) \times P$  times of size of a node, while the time complexity is more due to making one extra copy of the entire object for the first attempt. To see why these two algorithms are correct, we refer the interested reader to [34] for the complete mechanical proof.

### 3.6 Conclusions

This chapter shows an approach to verification of simulation and refinement between a lower-level specification and a higher-level specification. It is motivated by the algorithm of lock-free garbage collection (which will be presented in chapter 5). Using the reduction theorem, the verification effort for a lock-free algorithm becomes simpler since fewer invariants are required and some invariants are easier to discover and easier to formulate without considering the internal structure of the final implementation. Apart from safety properties, we have also considered the important problem of proving liveness properties using the strong fairness assumption.

A more fundamental problem with Herlihy's methodology is the overhead that results

from having multiple processes that simultaneously attempt to update a shared object. Since copying the entire object can be time-consuming, we present two enhanced algorithms that avoid unnecessary copying for large objects in cases where only small parts of the objects are modified. It is often better to distribute the contents of a large object over several small objects to allow parallel execution of operations on a large object. However, this requires that the contents of those small objects must be independent of each other.

Formal verification is desirable because there could be subtle bugs as the complexity of algorithms increases. To ensure our hand-written proof presented in this chapter is not flawed, we use the higher-order interactive theorem prover PVS 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. For the complete mechanical proof, we refer the reader to [34].

## Chapter 4

# A general lock-free algorithm using compare-and-swap

This chapter is a slightly modified version of our paper that is under submission.



## 4.1 Introduction

We are interested in designing efficient data structures and algorithms on shared-memory multiprocessors. A natural model for these machines is an asynchronous parallel machine, in which the processes may execute instructions at a different rate, and are subject to long delays. On such machines, processes often need to coordinate with each other via shared data structures. In order to prevent the corruption of these concurrent objects, processes need a mechanism for synchronizing their access. The traditional approach is to explicitly synchronize access to shared data by different processes to ensure correct behaviors of the overall system, using synchronization primitives such as semaphores, monitors, guarded statements, mutex locks, etc. Consequently the operations of different processes on a shared data structure should appear to be serialized: if two operations execute simultaneously, the system guarantees the same result as if one of them is arbitrarily executed before the other.

If the blocked process is performing a high-priority or real-time task, it is highly undesirable to halt its progress. Due to blocking, the classical synchronization paradigms using locks can incur many problems such as long delays, convoying, priority inversion and deadlock. Using locks also involves a trade-off between coarse-grained locking which can significantly reduce opportunities for parallelism, and fine-grained locking which requires more careful design and is more prone to bugs.

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, and increase performance by allowing extra concurrency.

Herlihy [27] has shown that the *compare-and-swap* (*CAS*) primitive and the similar *load-linked* (*LL*)/*store-conditional* (*SC*) are universal primitives that solve the consensus problem. A number of researchers [6, 9, 28, 29, 51, 54] have proposed techniques for designing *lock-free* implementations. The basis of these techniques is using some synchronization primitives such as *CAS*, or *LL/SC*.

Many machines provide either *CAS* or *LL/SC*, but not both. All architectures that support *LL/SC* restrict memory accesses between *LL* and *SC*. Furthermore, most kinds of hardware do not provide the complete semantics expected by program designers for the

implementations of *LL/SC*. For example, the cache-coherence mechanism may let *SC* fail spuriously, i.e., a *SC* operation may incorrectly fail in an implementation if a cached word is selected for replacement by the cache protocol. Some machines such as DEC Alpha and PowerPC, also restrict *LL/SC* operations from being concurrent executed since *LL* and *SC* are implemented using only one tag bit per processor.

Associated with most uses of *CAS* (and restricted *LL/SC*) is the ABA problem [41]. When swinging the pointer, we do not want the operation to succeed if the referred contents has changed since it was read. This problem occurs when the pointer has changed from *A* to *B*, but then subsequently changes back to *A* again. In that case, the *CAS* primitive will successfully change the value of the pointer, possibly corrupting the data structure because the referred contents may not satisfy the expected conditions. The simplest and most efficient solution to the ABA problem is to include a tag with the memory location such that the tag is incremented with each update of the target location [57]. Usually, however, this solution with tags in principle requires that the tags are unbounded. The practical solution of taking 32-bit integers for the tags gives an infinitesimal but positive probability of misbehaviour by wrap around. In our solution, this problem is eliminated.

The correctness properties of an implementation are seldom easy to verify. Our previous work (see chapter 2) shows that a proof may require huge amounts of effort, time, and skill. We therefore develop a reduction theorem that enables us to reason about a lock-free program to be designed on a higher level than the synchronization primitives. The reduction theorem is 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 theorem, fewer invariants are required and some invariants are easier to discover and easier to formulate, without considering the internal structure of the final implementation. In particular, nested loops in the algorithm may be eliminated at a time.

In chapter 3, we have shown a similar reduction theorem for reducing lock-free implementations using *LL/SC*. This time, we aim to provide correct lock-free transformation using *CAS*. Our algorithm is a variation of Herlihy's general methodology for lock-free transformation. The basis of our techniques is to poll different locations on reading and writing objects, in such a way that the consistency of an object can be checked by its location instead of its tag. It consists of simple code that can be easily implemented using C-like languages.

## 4.2 Synchronization primitives

Traditional multiprocessor architectures have included hardware support only for low level synchronization primitives such as *CAS* and *LL/SC*, while high level synchronization primitives such as locks, barriers, and condition variables have to be implemented in software.

*CAS* atomically compares the contents of a location with a value and, if they match, stores a new value at the location. The semantics of *CAS* is given by equivalent atomic statements below.

```
proc CAS(ref X; in old, new): bool =
  ⟨ if X = old then X := new; return true
    else return false; fi ⟩
```

*LL* and *SC* are a pair of instructions, closely related to the *CAS*, and together implement an atomic Read/Write cycle. Instruction *LL* first reads the content of a memory location, say *X*, and marks it as “reserved” (not “locked”). If no other processor changes the content of *X* in between, the subsequent *SC* operation of the same process succeeds and modifies the value stored; otherwise it fails. There is also a validate instruction *VL*, used to check whether *X* was not modified since the corresponding *LL* instruction was executed. For the semantics of *LL*, *SC* and *VL*, see chapter 3.

An atomic counter can be implemented by *fetch-and-increment* (*FAI*) and *fetch-and-decrement* (*FAD*) given below. Both operations return the original value of a memory location after atomically increment and decrement the counter, respectively. From hardware point of view, they are simpler versions of *CAS*.

```
proc FAI(ref X): int =
  ⟨ X := X + 1; return X - 1; ⟩
```

*FAD* is declared analogously. When *FAI* and *FAD* are not available on the machine architectures, then they can be easily implemented by *CAS* and *LL/SC*. E.g., *FAI* can be implemented by *CAS* in the following lock-free way.

```
proc FAI(ref X): int =
local Y;
loop
  Y := X;
```

```

    if CAS( $X$ ,  $Y$ ,  $Y + 1$ ) then return  $Y$ ; fi;
  end;
end.

```

### 4.3 The lock-free implementation using CAS

In chapter 3, we formalized Herlihy’s methodology [28] for transferring a sequential implementation  $\mathcal{S}_a$  (see Fig. 3.1) of any data structure into a lock-free synchronization  $\mathcal{S}_{ll/sc}$  given in Fig. 3.2, using synchronization primitives  $LL/SC$ . We now turn our attention to the lock-free implementation of the same interface using  $CAS$ , which is given by the algorithm  $\mathcal{S}_{cas}$  shown in Fig. 4.1. This lock-free implementation is inspired by the lock-free implementation  $\mathcal{S}_{ll/sc}$  (see Fig. 3.2). The lines  $c_2$ ,  $c_6$  and  $c_7$  of  $\mathcal{S}_{ll/sc}$  correspond in  $\mathcal{S}_{cas}$  to the fragments from  $d_{20}$  to  $d_{23}$ , from  $d_{60}$  to  $d_{65}$ , and from  $d_{70}$  to  $d_{71}$ , respectively.

The basic idea is to employ array **prot** to count the number of processes that are using an index for accessing a node, in such a way that the consistency of a node can be checked by its index: suppose process  $p$  first reads the index of node  $x.p$  to  $m.p$  (see line  $d_{20}$ ), then the consistency can be checked later by the predicate  $m.p = \text{indir}[x.p]$ . In  $\mathcal{S}_{ll/sc}$ ,  $LL/VL$  and  $LL/SC$  are taken as pairs of instructions, that together implement the atomic read/write cycle. In  $\mathcal{S}_{cas}$ , we therefore increment and decrement the corresponding counter (in array **prot**) at the beginning and the end of a cycle.

As in  $\mathcal{S}_{ll/sc}$ , we need to ensure all indices of shared nodes and “private” nodes (still declared in a public way) are mutually different. Moreover, after success of the analogue of  $SC$ , the previous shared node can not serve as a private node immediately (unlike that in  $\mathcal{S}_{ll/sc}$ , see line  $c_6$ ) if some process is still hanging on that node. Otherwise, interference may occur when the new “private” node is redirected to be a shared node again. Every “private” node for each process is now truly private since it does not allow some other process to have a peep at its content.

In  $\mathcal{S}_{cas}$ , we introduce a constant  $K \geq N + 2P$  for the sizes of the arrays **node** and **prot**. There is a trade-off between space and time that the user can choose: large  $K$  is faster when an unused index is chosen at line  $d_{64}$ , but large  $K$  requires more space.

The guard in line  $d_{22}$  is essential since it guarantees that the accessing node can not be taken as a private node during the read/write cycle. Decrement of **prot**[ $m$ ] in line  $d_{61}$  is necessary since  $m$  does not refer to a shared node when  $CAS$  in line  $d_{60}$  succeeds. When

<p><b>Constant</b></p> <p><math>P</math> = number of processes;  <math>N</math> = number of nodes;  <math>K = N + 2P</math>;</p> <p><b>Shared variable</b></p> <p>pub: aType;  node: array <math>[1 \dots K]</math> of nodeType;  indir: array <math>[1 \dots N]</math> of <math>1 \dots K</math>;  prot: array <math>[1 \dots K]</math> of <math>0 \dots K</math>;</p> <p><b>Private variable</b></p> <p>priv: bType; pc: <math>[d_{10} \dots d_{71}]</math>; suc: bool;  <math>x: 1 \dots N</math>; mp, m: <math>1 \dots K</math>; tm, tm1: cType;</p> <p><b>Program</b></p> <pre> loop d10:  noncrit(pub, priv, tm, x);       loop         loop d20:    m := indir[x]; d21:    &lt; prot[m]++; &gt; d22:    if m = indir[x] then break;      % goto d30 d23:    else &lt; prot[m]--; &gt; fi;         end; d30:    read(node[mp], node[m]); d40:    if guard(node[mp], priv) then d50:      com(node[mp], priv, tm1); d60:      if CAS(indir[x], m, mp) then         tm := tm1; d61:      &lt; prot[m]--; &gt; d62:      if prot[m] = 1 then mp := m;         else d63:        &lt; prot[m]--; &gt;         loop d64:          choose mp from <math>1 \dots K</math>           if CAS(prot[mp], 0, 1) then             break; fi;      % goto d10           end; fi;           break;      % goto d10         else d65:        &lt; prot[m]--; &gt; fi;      % goto d20         else d70:        suc := (m = indir[x]); d71:        &lt; prot[m]--; &gt;           if suc then break; fi; fi;      % goto d10 if suc, else goto d20         end;       end. </pre> <p><b>Initial conditions</b></p> $\Theta_{cas}: (\forall p: 1 \dots P: pc_p = d_{10} \wedge mp_p = N+p) \wedge (\forall i: 1 \dots N: indir[i] = i) \\ \wedge (\forall i: 1 \dots K: prot[i] = (i \leq N+P ? 1 : 0))$ <p><b>Liveness</b></p> $\mathcal{L}_{cas}: \Box(pc_p = d_{20} \longrightarrow \Diamond pc_p = d_{10})$
--

Figure 4.1: Lock-free implementation  $\mathcal{S}_{cas}$  of  $\mathcal{S}_a$

the check in line  $d_{62}$  finds that  $\text{prot}[m]$  equals 1, it means that only this process is hanging on that index, and the process can thus immediately treat that node as its private node. Otherwise, before the process starts to find an unused index for its private node, it needs to release the reading access to the node. When a new unused index, say  $mp$ , is chosen in line  $d_{64}$  for private use, the process increments  $\text{prot}[mp]$  to 1. Therefore, no other process will regard that chosen “index” as an unused index and take that for its private use.

#### 4.4 Correctness

In this section we prove that the concrete system  $\mathcal{S}_{cas}$  implements the abstract system  $\mathcal{S}_a$ . The correctness of the lock-free implementation  $\mathcal{S}_{cas}$  does not depend on the correctness of the lock-free implementation  $\mathcal{S}_{ll/sc}$ . Formally, as we did in chapter 3, we introduce  $N_c$  as the relation corresponding to command *noncrit* on  $(\mathbf{aType} \times \mathbf{bType} \times \mathbf{cType}, \mathbf{aType} \times \mathbf{bType} \times 1 \dots N)$ ,  $P_g$  as the predicate computed by *guard* on  $\mathbf{nodeType} \times \mathbf{bType}$ ,  $R_c$  as the relation corresponding to *com* on  $(\mathbf{nodeType} \times \mathbf{bType}, \mathbf{nodeType} \times \mathbf{cType})$ , and define

$$\begin{aligned} \Sigma_a &\triangleq (\mathbf{Node}[1 \dots N], \mathbf{pub}) \times (pc, x, \mathbf{priv}, tm)^P, \\ \Sigma_{cas} &\triangleq (\mathbf{node}[1 \dots K], \mathbf{indir}[1 \dots N], \mathbf{prot}[1 \dots K], \mathbf{pub}) \\ &\quad \times (pc, x, mp, m, \mathbf{suc}, \mathbf{priv}, tm, tm1)^P. \\ \Pi_a(\Sigma_a) &\triangleq (\mathbf{Node}[1 \dots N], \mathbf{pub}), \quad \Pi_{cas}(\Sigma_{cas}) \triangleq (\mathbf{node}[\mathbf{indir}[1 \dots N]], \mathbf{pub}). \\ \mathcal{N}_a &\triangleq \bigvee_{0 \leq i \leq 2} \mathcal{N}_{a_i}, \quad \mathcal{N}_{cas} \triangleq \bigvee_{10 \leq i \leq 71} \mathcal{N}_{d_i}. \end{aligned}$$

The transitions of the abstract system can be described:  $\forall s, t: \Sigma_a, p: 1 \dots P$ :

$$\begin{aligned} s[\llbracket \mathcal{N}_{a_0} \rrbracket_p] t &\triangleq s = t \quad (\text{to allow stuttering}) \\ s[\llbracket \mathcal{N}_{a_1} \rrbracket_p] t &\triangleq pc.p = a_1 \wedge pc'.p = a_2 \wedge Pres(\mathcal{V} - \{\mathbf{pub}, \mathbf{priv}.p, pc.p, x.p\}) \\ &\quad \wedge ((\mathbf{pub}, \mathbf{priv}.p, tm.p), (\mathbf{pub}, \mathbf{priv}.p, x.p)') \in N_c \\ s[\llbracket \mathcal{N}_{a_2} \rrbracket_p] t &\triangleq pc.p = a_2 \wedge pc'.p = a_1 \wedge (P_g(\mathbf{Node}[x.p], \mathbf{priv}.p) \\ &\quad \wedge ((\mathbf{Node}[x.p], \mathbf{priv}.p), (\mathbf{Node}[x.p], tm.p)') \in R_c \\ &\quad \wedge Pres(\mathcal{V} - \{pc.p, \mathbf{Node}[x.p], tm.p\}) \\ &\quad \vee \neg P_g(\mathbf{Node}[x.p], \mathbf{priv}.p) \wedge Pres(\mathcal{V} - \{pc.p\})). \end{aligned}$$

The transitions of the concrete system can be described in the same way. Here we only provide the description of concrete transitions  $d_{60}$  and  $d_{64}$ :  $\forall s, t: \Sigma_{cas}, p: 1 \dots P$ :

$$s[\llbracket \mathcal{N}_{d_{60}} \rrbracket_p] t \triangleq pc.p = d_{60} \wedge (\mathbf{indir}[x.p] = m.p \wedge pc'.p = d_{61} \wedge (\mathbf{indir}[x.p])' = mp.p$$

$$\begin{aligned}
& \wedge tm'.p = tm1.p \wedge Pres(\mathcal{V} - \{pc.p, \text{indir}[x.p], tm.p\}) \\
& \vee \text{indir}[x.p] \neq m.p \wedge pc'.p = d_{65} \wedge Pres(\mathcal{V} - \{pc.p\}). \\
s\llbracket(\mathcal{N}_{d_{64}})_p\rrbracket t \triangleq & pc.p = d_{64} \wedge \exists k: 1 \dots K: (\text{prot}[k] = 0 \wedge pc'.p = d_{10} \wedge (\text{prot}[k])' = 1 \\
& \wedge mp'.p = k \wedge Pres(\mathcal{V} - \{pc.p, \text{prot}[k], mp.p\})) \\
& \vee (\text{prot}[k] \neq 0 \wedge Pres(\mathcal{V})).
\end{aligned}$$

To prove that  $\mathcal{S}_{cas}$  implements  $\mathcal{S}_a$ , we define the state mapping  $\varphi: \Sigma_{cas} \rightarrow \Sigma_a$  by showing how each component of  $\Sigma_a$  is generated from components in  $\Sigma_{cas}$ :

$$\begin{aligned}
\forall x: 1 \dots N: \text{Node}_a[x] &= \text{node}_{cas}[\text{indir}_{cas}[x]], \\
\forall p: 1 \dots P: pc_a.p &= (pc_{cas}.p = d_{10} \vee pc_{cas}.p \in [d_{61} \dots d_{64}] \\
&\vee (pc_{cas}.p = d_{71} \wedge suc_p) ? a_1 : a_2),
\end{aligned}$$

where the subscript indicates the system a variable belongs to, and the remaining variables in  $\Sigma_a$  are identical to the variables occurring in  $\Sigma_{cas}$ .

#### 4.4.1 Invariants

We establish some invariants for the concrete system  $\mathcal{S}_{cas}$ , that will aid us in proving the refinement.

$$\begin{aligned}
I1: \quad & p \neq q \wedge pc.p \notin [d_{61} \dots d_{64}] \wedge pc.q \notin [d_{61} \dots d_{64}] \Rightarrow mp.p \neq mp.q \\
I2: \quad & pc.p \notin [d_{61} \dots d_{64}] \Rightarrow \text{indir}[x] \neq mp.p \\
I3: \quad & x \neq y \Rightarrow \text{indir}[x] \neq \text{indir}[y] \\
I4: \quad & pc.p = d_{60} \wedge m.p = \text{indir}[x.p] \\
& \Rightarrow P_g(\text{node}[m.p], \text{priv}.p) \wedge ((\text{node}[m.p], \text{priv}.p), (\text{node}[mp.p], tm1.p)) \in R_c \\
I5: \quad & pc.p = d_{70} \wedge m.p = \text{indir}[x.p] \Rightarrow \neg P_g(\text{node}[m.p], \text{priv}.p)
\end{aligned}$$

In the expression of invariants, free variables  $p$  and  $q$  range over  $1 \dots P$ , and  $x$  and  $y$  range over  $1 \dots N$ . Invariants  $I1$  and  $I2$  indicate that, for any process  $p$ ,  $\text{node}[mp.p]$  can be treated as a “private” node of process  $p$  since only process  $p$  can modify that. Invariant  $I3$  implies that all shared nodes are different. Invariant  $I4$  gives the precondition when process  $p$  arrives at line  $d_{60}$  and node  $x.p$  has not been changed since the last execution of line  $d_{20}$ . Invariant  $I5$  gives the precondition when process  $p$  arrives at line  $d_{70}$  and node  $x.p$  has not been changed since the last execution of line  $d_{20}$ .

To prove the invariances of  $I1$  to  $I5$ , we postulate

- I6:*  $\forall i: 1 \dots K: \mathbf{prot}[i] = \#(\{x: 1 \dots N \mid \mathbf{indir}[x] = i\})$   
 $+ \#(\{p \mid (pc_p \notin [d_{61} \dots d_{64}] \wedge mp_p = i) \vee (pc_p = d_{61} \wedge m_p = i)\})$   
 $+ \#(\{p \mid (pc_p \in [d_{22} \dots d_{71}] \wedge pc_p \neq d_{64} \wedge m_p = i)\})$
- I7:*  $pc.p \in [d_{30} \dots d_{71}] \wedge pc.p \neq d_{64} \wedge mp.q = m.p \Rightarrow pc.q \in [d_{61} \dots d_{64}]$
- I8:*  $pc.p \in [d_{40} \dots d_{50}] \wedge m.p = \mathbf{indir}[x.p] \Rightarrow \mathbf{node}[m.p] = \mathbf{node}[mp.p]$
- I9:*  $pc.p = d_{50} \Rightarrow P_g(\mathbf{node}[mp.p], \mathbf{priv}.p)$

Invariant *I6* precisely describes the counter  $\mathbf{prot}[i]$  for each  $i \in 1 \dots K$ . Invariant *I7* implies that process  $p$  cannot read the “private” node of other process  $q$ . Invariant *I8* indicates the postcondition after process  $p$  making a private copy of the chosen node  $x.p$  at line  $d_{30}$ . Invariant *I9* provides the precondition when process  $p$  arrives at line  $d_{50}$ .

#### 4.4.2 Refinement

It is straightforward to verify that the first premise of the refinement mapping and the first premise of the simulation hold. It is easy to verify the second premise of the simulation for most of the transitions. We examine in detail only transition  $d_{60}$ .

Transition  $d_{60}$  executed by process  $p$  is split up into two cases according to whether  $\mathbf{indir}[x.p] = m.p$  holds in the precondition. This gives rise to the following two verification conditions:

1.  $\forall s, t \in \Sigma_{cas}: \mathbf{indir}[x.p] = m.p \wedge s \ll (\mathcal{N}_{d_{60}})_p t \Rightarrow \varphi(s) \ll (\mathcal{N}_{a_2})_p \varphi(t)$ .

Using invariant *I4*, we obtain the following relation that holds between the concrete states  $s$  and  $t$ :

$$\begin{aligned}
 &pc.p = d_{60} \wedge pc'.p = d_{61} \wedge P_g(\mathbf{node}[\mathbf{indir}[x.p]], \mathbf{priv}.p) \\
 &\wedge ((\mathbf{node}[\mathbf{indir}[x.p]], \mathbf{priv}.p), (\mathbf{node}[\mathbf{indir}[x.p]], tm.p)') \in R_c \\
 &\wedge Pres(\mathcal{V} - \{\mathbf{indir}[x.p], pc.p, tm.p\}).
 \end{aligned}$$

This corresponds to the following relation that holds between the abstract states  $\varphi(s)$  and  $\varphi(t)$ :

$$\begin{aligned}
 &pc.p = a_2 \wedge pc'.p = a_1 \wedge P_g(\mathbf{Node}[x.p], \mathbf{priv}.p) \\
 &\wedge ((\mathbf{Node}[x.p], \mathbf{priv}.p), (\mathbf{Node}[x.p], tm.p)') \in R_c \\
 &\wedge Pres(\mathcal{V} - \{\mathbf{Node}[x.p], pc.p, tm.p\}).
 \end{aligned}$$



We then conclude  $\varphi(s) \llbracket (\mathcal{N}_{a_2})_p \rrbracket \varphi(t)$  holds.

2.  $\forall s, t \in \Sigma_{cas}: \text{indir}[x.p] \neq m.p \wedge s \llbracket (\mathcal{N}_{d_{60}})_p \rrbracket t \Rightarrow \varphi(s) \llbracket (\mathcal{N}_{a_0})_p \rrbracket \varphi(t)$ .

We obtain the following relation that holds between the concrete states  $s$  and  $t$ :

$$pc.p = d_{60} \wedge pc'.p = d_{65} \wedge Pres(\mathcal{V} - \{pc.p\}).$$

This corresponds to the following relation that holds between the abstract states  $\varphi(s)$  and  $\varphi(t)$ :

$$pc.p = a_2 \wedge pc'.p = a_2 \wedge Pres(\mathcal{V} - \{pc.p\}).$$

We then conclude  $\varphi(s) \llbracket (\mathcal{N}_{a_0})_p \rrbracket \varphi(t)$  holds.

For the third premise of refinement mapping, we deduce

$$\begin{aligned} \sigma \models \mathcal{L}_{cas} &\equiv \sigma \models \Box(pc = d_{20} \longrightarrow \Diamond pc = d_{10}) \\ &\Rightarrow \sigma \models \Box(pc \in [d_{20} \dots d_{71}] \longrightarrow \Diamond pc = d_{10}) \\ &\Rightarrow \varphi(\sigma) \models \Box(pc = a_2 \longrightarrow \Diamond pc = a_1) \\ &\equiv \mathcal{L}_a(\varphi(\sigma)) \end{aligned}$$

Consequently, we have the main reduction theorem for the lock-free implementation using CAS:

**Theorem 4.4.1** *The abstract system  $\mathcal{S}_a$  defined in Fig. 3.1 is implemented by the concrete system  $\mathcal{S}_{cas}$  defined in Fig. 4.1, that is,  $\exists \varphi: \mathcal{S}_{cas} \sqsubseteq \mathcal{S}_a$ .*

#### 4.4.3 Progress

Note that the liveness condition  $L_{cas}$  in  $\mathcal{S}_{cas}$  is postulated, and Theorem 4.4.1 does not require the proof of this liveness condition. In this section, we prove that the liveness condition  $L_{cas}$  in  $\mathcal{S}_{cas}$  is feasible.

The proof of liveness relies on the fairness conditions associated with a specification. The purpose for fairness conditions is to rule out executions where the system idles indefinitely with control at some internal point of a procedure and with some transition of that procedure enabled. Fairness arguments usually depend on safety properties of the system.

The lock-freedom property means that a non-faulty process will finish its task in a finite number of steps unless other processes are infinitely making progress. In the concrete system  $\mathcal{S}_{cas}$ , we assume

$$\Box(pc.p \in [d_{20} \dots d_{60}] \cup [d_{65} \dots d_{71}]) \longrightarrow \Diamond(pc.p \in \{d_{60}, d_{70}\} \wedge m.p = \text{indir}[x.p]).$$

This assumption indicates that for every process  $p$ , when process  $p$  remains in the region  $[d_{20} \dots d_{60}] \cup [d_{65} \dots d_{71}]$  and executes  $d_{20}$  infinitely often, it will eventually succeed in reaching  $d_{60}$  or  $d_{70}$  with precondition “ $m.p = \text{indir}[x.p]$ ”. Otherwise, other processes are infinitely making progress on modifying node  $x.p$  before process  $p$  finishes its task, which falsifies the antecedent of the lock-freedom.

Using rule (SF1) (presented in section 1.2) with the following substitutions:

$$\begin{aligned} R &: pc.p \in [d_{20} \dots d_{60}] \cup [d_{65} \dots d_{71}]; \quad \mathcal{N} : \mathcal{N}_{cas}; \\ Q &: pc.p \in [d_{61}, d_{64}] \vee (pc.p = d_{71} \wedge suc.p); \\ \mathcal{A} &: (m.p = \text{indir}[x.p] \wedge \mathcal{N}_{d_{60}}) \vee (m.p = \text{indir}[x.p] \wedge \mathcal{N}_{d_{70}}); \\ I &: true, \end{aligned}$$

we then obtain:

$$pc.p \in [d_{20} \dots d_{60}] \cup [d_{65} \dots d_{71}] \quad o \rightarrow \quad pc.p \in [d_{61} \dots d_{64}] \vee (pc.p = d_{71} \wedge suc.p).$$

Similarly, but more obviously, we have

$$\begin{aligned} pc.p = d_{71} \wedge suc.p &\quad o \rightarrow \quad pc.p = d_{10}. \\ pc.p \in [d_{61} \dots d_{64}] &\quad o \rightarrow \quad pc.p = d_{64}. \end{aligned}$$

The invariant I6 implies that, while process  $p$  is at  $d_{64}$ , we have  $\sum_{i=1}^K \text{prot}[i] \leq N + 2P - 2 \leq K - 2$ . It then follows from the pigeon hole principle that there exists  $j \in [1, K]$  such that  $\text{prot}[j] = 0$ . By the antecedent of lock-freedom, no other process can change  $\text{prot}[j]$  to be non-zero infinitely often while process  $p$  stays at  $d_{64}$  without finishing its task. Therefore, if process  $p$  acts infinitely often and chooses its value  $mp$  in line  $d_{64}$  by round robin, process  $p$  will exit the loop and arrive at  $d_{10}$  eventually. That is:

$$pc.p = d_{64} \quad o \rightarrow \quad pc.p = d_{10}.$$

According to the transitivity of “leadsto” ( $o \rightarrow$ ) relation, we finally obtain:

$$pc.p = d_{20} \quad o \rightarrow \quad pc.p = d_{10},$$

which is the liveness property defined in the concrete system  $\mathcal{S}_{cas}$ .

## 4.5 Conclusions

Lock-free algorithms offer significant reliability and performance advantages over conventional lock-based implementations. Many machines provide either *CAS* or *LL/SC*, but not both. This chapter presents a general lock-free pattern based on the weaker atomic primitive *CAS* without causing the ABA problem or problems with wrap around. The lock-free pattern makes it easier to develop the lock-free implementations of any data structures. It is a *CAS* variation of Herlihy's *LL/SC* methodology for lock-free transformation. It provides clear evidence that *CAS* is sufficient for practical implementations of lock-free data structures.

We present the lock-free pattern as a reduction theorem. Application of this theorem simplifies the verification effort for lock-free algorithms since fewer invariants are required and some invariants are easier to discover and easier to formulate without considering the internal structure of the final implementation. Apart from safety properties, we have also considered the important problem of proving liveness properties using the strong fairness assumption.

Formal verification is desirable because there could be subtle bugs as the complexity of algorithms increases. To ensure our proof is not flawed, we used the higher-order interactive theorem prover PVS for mechanical support. All invariants as well as the simulation relation have been completely verified with PVS. We felt that using PVS to prove the liveness does not give enough advantages over the handwritten proof to justify the investment and the delay in publication. We therefore defer a PVS proof of the liveness to future work. For the complete mechanical proof of safety, we refer the reader to [34].

## 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  $1 \leq j \leq \mathbf{arity}[x]$ .

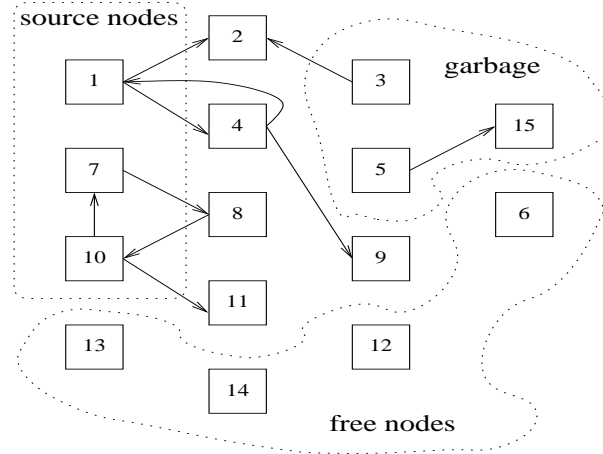


Figure 5.1: A graph representation of the memory

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 **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 **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

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

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

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

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:

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

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()
   $\langle$  choose  $x \in \text{Node}$  such that  $\neg R(x) \wedge x \notin \text{free}; \text{free} := \text{free} \cup x; \mathbf{\rangle}$ 
```

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;

$\langle$  **when** available **extract**  $x$  **from** **free**;

    arity[ $x$ ] := 0; roots<sub>self</sub> := roots<sub>self</sub>  $\cup$  { $x$ };  $\rangle$

**return**  $x$ ;

**proc** *AddChild*( $x$ ,  $y$ : Node): Bool

{  $R(\text{self}, x) \wedge R(\text{self}, y)$  }

**local**  $suc$  : Bool;

$\langle$   $suc := (\text{arity}[x] < C)$ ;

**if**  $suc$  **then** arity[ $x$ ]++; child[ $x$ , arity[ $x$ ]] :=  $y$ ; **fi**  $\rangle$

**return**  $suc$ ;

**proc** *GetChild*( $x$ : Node,  $rth$ :  $\mathbb{N}$ ): Node  $\cup$  {0}

{  $R(\text{self}, x)$  }

**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:  $\mathbb{N}$ ; % number of children
    child: array [1... $C$ ] of 1... $N$ ; % pointers to children
    color: colorType; % holds the color of the node
    srcnt:  $\mathbb{N}$ ; % reference counter for a source node
    freecnt:  $\mathbb{N}$ ; % dereference counter for a source node
    ari:  $\mathbb{N}$ ; % number of children at the beginning of GC
    father:  $\mathbb{N} \cup \{-1\}$ ; % records the parent node GC traverses
    round:  $\mathbb{N}$ ; % 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:  $\mathbb{N}$ ; % the version of the current round of GC
Private variables
  roots: a subset of 1... $N$ ; % a set of root nodes
  rnd:  $\mathbb{N}$ ; % private copy of "shRnd", initially 0!
  toBeC: a subset of 1... $N$ ; % a set of nodes to be checked
Initialization:
  shRnd = 1  $\wedge \forall x: 1 \dots N: \text{round}[x] = 1$ ;
  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>:

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

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

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

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  $\mathit{rnd}_p = \mathbf{shRnd}$ . 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  $\llbracket \rrbracket$  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  $rnd_{self} = \mathbf{shRnd}$  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:

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

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  $rnd_{self} \neq \mathbf{shRnd}$ ) 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*

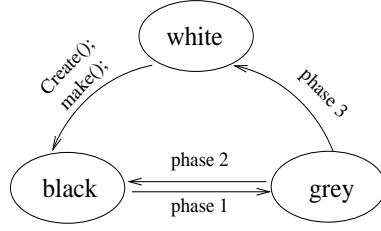


Figure 5.4: Transitions between these colors.

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*

The reduction theorems require that every labeled atomic group of statements in the higher-level algorithm refers to at most one shared node. In the procedure *Mark\_stack*, local variables *ch* and *k* are therefore introduced to store the old children of a node, temporarily. This also prevents processes from visiting a shared node unnecessarily. It adds a proof obligation that these local variables preserve the information of the node when the process is not delayed.

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
        GCollect(); 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
        GCollect(); 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:     $\llbracket \text{Mbox}[r, \text{self}] := 0; \rrbracket$   $\langle \text{assert } x \in \text{roots}; \rangle$  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  $R(\text{self}, x) \wedge R(\text{self}, y) \rightarrow \text{AddChild}(x, y)$ ;
      (G, x, rth) with  $R(\text{self}, x) \rightarrow \text{GetChild}(x, \text{rth})$ ;
      (M, c, n) with  $\forall j: 1 \dots n: R(\text{self}, c[j]) \rightarrow \text{Make}(c, n)$ ;
      (P, x) with  $R(\text{self}, x) \wedge x \notin \text{roots}_{\text{self}} \rightarrow \text{Protect}(x)$ ;
      (U, z) with  $z \in \text{roots}_{\text{self}} \rightarrow \text{UnProtect}(z)$ ;
      (S, x, r) with  $R(\text{self}, x) \wedge \text{Mbox}(\text{self}, r) = 0 \rightarrow \text{Send}(x, r)$ ;
      (R, r) with  $\text{Mbox}(r, \text{self}) \neq 0 \rightarrow \text{Receive}(r)$ ;

```

```

    (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  $\llbracket \cdot \rrbracket$ . 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:

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

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  $\circ\rightarrow$ ) relation that was developed for UNITY in [12]. For assertions  $P$  and  $Q$ , we use the formal definition:

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

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,

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

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

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” ( $\text{o} \rightarrow$ ) 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  $(P \text{o} \rightarrow Q)$  then  $\Diamond Q$ .*

(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  $m$ ,  $\mathbf{shRnd} = m \wedge \neg R(x) \circ \rightarrow \mathbf{shRnd} \leq m + 2 \wedge \mathbf{white}(x)$ .

An invariant has the form of  $Q \Rightarrow \Box Q$ , 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  $\mathbf{color}[x] = \mathbf{white}$  by  $\mathbf{white}(x)$ , 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:

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

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  $m$ ,

- (a)  $A_0(m) \wedge \neg R1(x) \wedge \text{black}(x) \wedge \text{round}[x] = m$   
 $\mathcal{U} \quad A_1(m) \wedge \neg R1(x) \wedge \text{black}(x) \wedge \text{round}[x] = m$   
 $\mathcal{U} \quad A_1(m) \wedge \neg R1(x) \wedge \text{grey}(x) \wedge \text{round}[x] = m + 1$   
 $\mathcal{U} \quad A_2(m) \wedge \neg R1(x) \wedge \text{grey}(x) \wedge \text{round}[x] = m + 1$   
 $\mathcal{U} \quad A_3(m) \wedge \neg R1(x) \wedge \text{grey}(x) \wedge \text{round}[x] = m + 1$   
 $\mathcal{U} \quad A_3(m) \wedge \text{white}(x)$
- (b)  $\text{shRnd} = m = \text{round}[x] - 1 \wedge \neg R1(x) \wedge \neg \text{white}(x)$   
 $\mathcal{U} \quad (\text{shRnd} = m \wedge \text{white}(x)) \vee (\text{shRnd} = m + 1 = \text{round}[x] \wedge \neg R1(x) \wedge \neg \text{white}(x))$
- (c)  $A_0(m) \wedge \neg R(x) \wedge \neg \text{white}(x) \wedge \neg R(w) \wedge \text{black}(w) \wedge \text{srcnt}(w) > 0 \wedge \text{round}[w] = m$   
 $\mathcal{U} \quad A_1(m) \wedge \neg R(x) \wedge \neg \text{white}(x) \wedge \neg R(w) \wedge \text{black}(w) \wedge \text{srcnt}(w) > 0 \wedge \text{round}[w] = m$   
 $\mathcal{U} \quad A_1(m) \wedge \neg R(x) \wedge \neg \text{white}(x) \wedge \neg R(w) \wedge \text{grey}(w) \wedge \text{srcnt}(w) > 0 \wedge \text{round}[w] = m + 1$   
 $\mathcal{U} \quad A_2(m) \wedge \neg R(x) \wedge \neg \text{white}(x) \wedge \neg R(w) \wedge \text{grey}(w) \wedge \text{srcnt}(w) > 0 \wedge \text{round}[w] = m + 1$   
 $\mathcal{U} \quad A_2(m) \wedge \neg R(x) \wedge \neg \text{white}(x) \wedge \text{srcnt}(w) = 0$
- (d)  $\text{shRnd} = m = \text{round}[w] - 1 \wedge \neg R(x) \wedge \neg \text{white}(x) \wedge \neg R(w) \wedge \text{srcnt}(w) > 0$   
 $\mathcal{U} \quad (\text{shRnd} = m \wedge \text{white}(x)) \vee (\text{shRnd} = m \wedge \neg R(x) \wedge \neg \text{white}(x) \wedge \text{srcnt}(w) = 0)$   
 $\vee (\text{shRnd} = m + 1 = \text{round}[w] \wedge \neg R(x) \wedge \neg \text{white}(x) \wedge \neg R(w) \wedge \text{srcnt}(w) > 0)$
- (e)  $\text{shRnd} \leq m \wedge \neg R(x) \wedge \neg \text{white}(x) \wedge \neg (\text{srcnt}(w) > 0 \wedge (w \xrightarrow{*} x))$   
 $\mathcal{U} \quad (\text{shRnd} \leq m \wedge \text{white}(x)) \vee \text{shRnd} > m$
- (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  $\Box \neg A_4(m)$ . 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 *GCollect* and *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  $m$ ,*

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

**Proof:** By invariant *I16*, we obtain  $black(x)$ . 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  $m$ ,*

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

**Proof:** By invariant *I13*, we know  $round[x] = m \vee round[x] = m + 1$ . 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  $m$ ,*

$$shRnd = m = round[w] \wedge \neg R(x) \wedge \neg white(x) \wedge \neg R(w) \wedge srcnt(w) > 0 \quad o \rightarrow \\ shRnd = m \wedge \neg R(x) \wedge \neg white(x) \wedge srcnt(w) = 0$$

**Proof:** Obviously, we have  $R1(w)$ . By invariants *I16* and *I18*, we then obtain  $black(w)$ . 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  $m$ ,*



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

**Proof:** It holds obviously if  $\neg(srcnt(w) > 0 \wedge (w \xrightarrow{*} x))$  is true. Otherwise, by definition of relation  $R$  and transitivity of “ $\xrightarrow{*}$ ”, we obtain  $\neg R(w)$ . By invariant *I13*, we obtain  $round[w] = m \vee round[w] = m + 1$ . 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  $I$ ;  $\neg(srcnt(w) > 0 \wedge (w \xrightarrow{*} x))$  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  $\langle \dots \rangle$  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.

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.

# Appendix A

## For lock-free dynamic hash tables

### A.1 Invariants

Some abbreviations.

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

Axioms on functions *key* and *ADR*.

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

Main correctness properties

$$\begin{aligned} Co1: \quad pc = 14 &\Rightarrow val(r_{fi}) = rS_{fi} \\ Co2: \quad pc \in \{25, 26\} &\Rightarrow suc_{del} = sucS_{del} \\ Co3: \quad pc \in \{41, 42\} &\Rightarrow suc_{ins} = sucS_{ins} \\ Cn1: \quad pc = 14 &\Rightarrow cnt_{fi} = 1 \\ Cn2: \quad pc \in \{25, 26\} &\Rightarrow cnt_{del} = 1 \\ Cn3: \quad pc \in \{41, 42\} &\Rightarrow cnt_{ins} = 1 \\ Cn4: \quad pc = 57 &\Rightarrow cnt_{ass} = 1 \end{aligned}$$

The absence of memory loss is shown by

$$\text{No1: } \#(nbSet1) \leq 2 * P$$

$$\text{No2: } \#(nbSet1) = \#(nbSet2)$$

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

$$nbSet1 = \{k \mid k < \text{H\_index} \wedge \text{Heap}(k) \neq \perp\}$$

$$nbSet2 = \{i \mid \text{H}(i) \neq 0 \vee (\exists r : pc.r = 71 \wedge i_{rA}.r = i)\}$$

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

$$deSet1 = \{k \mid k < \text{curSize} \wedge Y[k] = \mathbf{del}\}$$

$$deSet2 = \{r \mid \text{index}.r = \text{currInd} \wedge pc.r = 25 \wedge \text{suc}_{del}.r\}$$

$$deSet3 = \{k \mid k < \text{H}(\text{next}(\text{currInd})).\text{size} \wedge \text{H}(\text{next}(\text{currInd})).\text{table}[k] = \mathbf{del}\}$$

$$ocSet1 = \{r \mid \text{index}.r \neq \text{currInd}$$

$$\vee pc.r \in [30, 41] \vee pc.r \in [46, 57] \vee pc.r \in [59, 65] \wedge \text{return}_{gA}.r \geq 30$$

$$\vee pc.r \in [67, 72] \wedge (\text{return}_{rA}.r = 59 \wedge \text{return}_{gA}.r \geq 30$$

$$\vee \text{return}_{rA}.r = 90 \wedge \text{return}_{ref}.r \geq 30)$$

$$\vee (pc.r = 90 \vee pc.r \in [104, 105]) \wedge \text{return}_{ref}.r \geq 30\}$$

$$ocSet2 = \{r \mid pc.r \geq 125 \wedge b_{mE}.r \wedge to.r = \text{H}(\text{currInd})\}$$

$$ocSet3 = \{r \mid \text{index}.r = \text{currInd} \wedge pc.r = 41 \wedge \text{suc}_{ins}.r$$

$$\vee \text{index}.r = \text{currInd} \wedge pc.r = 57 \wedge r_{ass}.r = \mathbf{null}\}$$

$$ocSet4 = \{k \mid k < \text{curSize} \wedge \text{val}(Y[k]) \neq \mathbf{null}\}$$

$$ocSet5 = \{k \mid k < \text{H}(\text{next}(\text{currInd})).\text{size} \wedge \text{val}(\text{H}(\text{next}(\text{currInd})).\text{table}[k]) \neq \mathbf{null}\}$$

$$ocSet6 = \{k \mid k < \text{H}(\text{next}(\text{currInd})).\text{size} \wedge \text{H}(\text{next}(\text{currInd})).\text{table}[k] \neq \mathbf{null}\}$$

$$ocSet7 = \{r \mid pc.r \geq 125 \wedge b_{mE}.r \wedge to.r = \text{H}(\text{next}(\text{currInd}))\}$$

$$prSet1(i) = \{r \mid \text{index}.r = i \wedge pc.r \notin \{0, 59, 60\}\}$$

$$prSet2(i) = \{r \mid \text{index}.r = i \wedge pc.r \in \{104, 105\} \vee i_{rA}.r = i \wedge \text{index}.r \neq i \wedge pc.r \in [67, 72]$$

$$\vee i_{nT}.r = i \wedge pc.r \in [81, 84] \vee i_{mig}.r = i \wedge pc.r \geq 97\}$$

$$prSet3(i) = \{r \mid \text{index}.r = i \wedge pc.r \in [61, 65] \cup [104, 105] \vee i_{rA}.r = i \wedge pc.r = 72$$

$$\vee i_{nT}.r = i \wedge pc.r \in [81, 82] \vee i_{mig}.r = i \wedge pc.r \in [97, 98]\}$$

$$prSet4(i) = \{r \mid \text{index}.r = i \wedge pc.r \in [61, 65] \vee i_{mig}.r = i \wedge pc.r \in [97, 98]\}$$

$$buSet1(i) = \{r \mid \text{index}.r = i \wedge (pc.r \in [1, 58] \cup (62, 68] \wedge pc.r \neq 65$$

$$\vee pc.r \in [69, 72] \wedge \text{return}_{rA}.r > 59 \vee pc.r > 72)\}$$

$$buSet2(i) = \{r \mid \text{index}.r = i \wedge pc.r = 104 \vee i_{rA}.r = i \wedge \text{index}.r \neq i \wedge pc.r \in [67, 68]$$

$$\vee i_{nT}.r = i \wedge pc.r \in [82, 84] \vee i_{mig}.r = i \wedge pc.r \geq 100\}$$

We have the following invariants concerning the Heap

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

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

- He3:  $\text{Heap}(\text{H}(\text{currInd})) \neq \perp$   
 He4:  $pc \in [1, 58] \vee pc > 65 \wedge \neg(pc \in [67, 72] \wedge i_{rA} = \text{index}) \Rightarrow \text{Heap}(\text{H}(\text{index})) \neq \perp$   
 He5:  $\text{Heap}(\text{H}(i)) \neq \perp \Rightarrow \text{H}(i).\text{size} \geq P$   
 He6:  $\text{next}(\text{currInd}) \neq 0 \Rightarrow \text{Heap}(\text{H}(\text{next}(\text{currInd}))) \neq \perp$

Invariants concerning hash table pointers

- Ha1:  $\text{H\_index} > 0$   
 Ha2:  $\text{H}(i) < \text{H\_index}$   
 Ha3:  $i \neq j \wedge \text{Heap}(\text{H}(i)) \neq \perp \Rightarrow \text{H}(i) \neq \text{H}(j)$   
 Ha4:  $\text{index} \neq \text{currInd} \Rightarrow \text{H}(\text{index}) \neq \text{H}(\text{currInd})$

Invariants about counters for calling the specification.

- Cn5:  $pc \in [6, 7] \Rightarrow \text{cnt}_{f_i} = 0$   
 Cn6:  $pc \in [8, 13] \vee pc \in [59, 65] \wedge \text{return}_{gA} = 10$   
 $\vee pc \in [67, 72] \wedge (\text{return}_{rA} = 59 \wedge \text{return}_{gA} = 10 \vee \text{return}_{rA} = 90 \wedge \text{return}_{ref} = 10)$   
 $\vee pc \geq 90 \wedge \text{return}_{ref} = 10$   
 $\Rightarrow \text{cnt}_{f_i} = \sharp(r_{f_i} = \mathbf{null} \vee a_{f_i} = \text{ADR}(r_{f_i}))$   
 Cn7:  $pc \in [16, 21] \wedge pc \neq 18 \vee pc \in [59, 65] \wedge \text{return}_{gA} = 20$   
 $\vee pc \in [67, 72] \wedge (\text{return}_{rA} = 59 \wedge \text{return}_{gA} = 20 \vee \text{return}_{rA} = 90 \wedge \text{return}_{ref} = 20)$   
 $\vee pc \geq 90 \wedge \text{return}_{ref} = 20$   
 $\Rightarrow \text{cnt}_{del} = 0$   
 Cn8:  $pc = 18 \Rightarrow \text{cnt}_{del} = \sharp(r_{del} = \mathbf{null})$   
 Cn9:  $pc \in [28, 33] \vee pc \in [59, 65] \wedge \text{return}_{gA} = 30$   
 $\vee pc \in [67, 72] \wedge (\text{return}_{rA} = 59 \wedge \text{return}_{gA} = 30 \vee \text{return}_{rA} = 77 \wedge \text{return}_{nT} = 30)$   
 $\vee \text{return}_{rA} = 90 \wedge \text{return}_{ref} = 30$   
 $\vee pc \in [77, 84] \wedge \text{return}_{nT} = 30 \vee pc \geq 90 \wedge \text{return}_{ref} = 30$   
 $\Rightarrow \text{cnt}_{ins} = 0$   
 Cn10:  $pc \in [35, 37] \vee pc \in [59, 65] \wedge \text{return}_{gA} = 36$   
 $\vee pc \in [67, 72] \wedge (\text{return}_{rA} = 59 \wedge \text{return}_{gA} = 36 \vee \text{return}_{rA} = 90 \wedge \text{return}_{ref} = 36)$   
 $\vee pc \geq 90 \wedge \text{return}_{ref} = 36$   
 $\Rightarrow \text{cnt}_{ins} = \sharp(a_{ins} = \text{ADR}(r_{ins}) \vee \text{suc}_{ins})$   
 Cn11:  $pc \in [44, 52] \vee pc \in [59, 65] \wedge \text{return}_{gA} \in \{46, 51\}$   
 $\vee pc \in [67, 72] \wedge (\text{return}_{rA} = 59 \wedge \text{return}_{gA} \in \{46, 51\}$   
 $\vee \text{return}_{rA} = 77 \wedge \text{return}_{nT} = 46 \vee \text{return}_{rA} = 90 \wedge \text{return}_{ref} \in \{46, 51\})$   
 $\vee pc \in [77, 84] \wedge \text{return}_{nT} = 46 \vee pc \geq 90 \wedge \text{return}_{ref} \in \{46, 51\}$   
 $\Rightarrow \text{cnt}_{ass} = 0$

Invariants about old hash tables, current hash table and the auxiliary hash table Y. Here, we universally quantify over all non-negative integers  $n < \text{curSize}$ .



- Cu1:  $H(index) \neq H(currInd) \wedge k < H(index).size$   
 $\wedge (pc \in [1, 58] \vee pc > 65 \wedge \neg(pc \in [67, 72] \wedge i_{rA} = index))$   
 $\Rightarrow H(index).table[k] = \mathbf{done}$
- Cu2:  $\sharp(\{k \mid k < curSize \wedge Y[k] \neq \mathbf{null}\}) < curSize$
- Cu3:  $H(currInd).bound + 2 * P < curSize$
- Cu4:  $H(currInd).dels + \sharp(deSet2) = \sharp(deSet1)$
- Cu5: Cu5 has been eliminated, but the numbering has been kept.
- Cu6:  $H(currInd).occ + \sharp(ocSet1) + \sharp(ocSet2) \leq H(currInd).bound + 2 * P$
- Cu7:  $\sharp(\{k \mid k < curSize \wedge Y[k] \neq \mathbf{null}\}) = H(currInd).occ + \sharp(ocSet2) + \sharp(ocSet3)$
- Cu8:  $next(currInd) = 0 \Rightarrow \neg oldp(H(currInd).table[n])$
- Cu9:  $\neg(oldp(H(currInd).table[n])) \Rightarrow H(currInd).table[n] = Y[n]$
- Cu10:  $oldp(H(currInd).table[n]) \wedge val(H(currInd).table[n]) \neq \mathbf{null}$   
 $\Rightarrow val(H(currInd).table[n]) = val(Y[n])$
- Cu11:  $LeastFind(a, n) \Rightarrow X(a) = val(Y[key(a, curSize, n)])$
- Cu12:  $X(a) = val(Y[key(a, curSize, n)]) \neq \mathbf{null} \Rightarrow LeastFind(a, n)$
- Cu13:  $X(a) = val(Y[key(a, curSize, n)]) \neq \mathbf{null} \wedge n \neq m < curSize$   
 $\Rightarrow ADR(Y[key(a, curSize, m)]) \neq a$
- Cu14:  $X(a) = \mathbf{null} \wedge val(Y[key(a, curSize, n)]) \neq \mathbf{null} \Rightarrow ADR(Y[key(a, curSize, n)]) \neq a$
- Cu15:  $X(a) \neq \mathbf{null} \Rightarrow \exists m < curSize : X(a) = val(Y[key(a, curSize, m)])$
- Cu16:  $\exists(f : [\{m : 0 \leq m < curSize\} \wedge val(Y[m]) \neq \mathbf{null}\} \rightarrow$   
 $\{v : v \neq \mathbf{null} \wedge (\exists k < curSize : v = val(Y[k]))\}) : f \text{ is bijective}$

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

- Ne1:  $currInd \neq next(currInd)$
- Ne2:  $next(currInd) \neq 0 \Rightarrow next(next(currInd)) = 0$
- Ne3:  $pc \in [1, 59] \vee pc \geq 62 \wedge pc \neq 65 \Rightarrow index \neq next(currInd)$
- Ne4:  $pc \in [1, 58] \vee pc \geq 62 \wedge pc \neq 65 \Rightarrow index \neq next(index)$
- Ne5:  $pc \in [1, 58] \vee pc \geq 62 \wedge pc \neq 65 \wedge next(index) = 0 \Rightarrow index = currInd$
- Ne6:  $next(currInd) \neq 0$   
 $\Rightarrow \sharp(ocSet6) \leq \sharp(\{k \mid k < curSize \wedge Y[k] \neq \mathbf{null}\}) - H(currInd).dels - \sharp(deSet2)$
- Ne7:  $next(currInd) \neq 0$   
 $\Rightarrow H(currInd).bound - H(currInd).dels + 2 * P \leq H(next(currInd)).bound$
- Ne8:  $next(currInd) \neq 0$   
 $\Rightarrow H(next(currInd)).bound + 2 * P < H(next(currInd)).size$
- Ne9:  $next(currInd) \neq 0 \Rightarrow H(next(currInd)).dels = \sharp(deSet3)$
- Ne9a:  $next(currInd) \neq 0 \Rightarrow H(next(currInd)).dels = 0$
- Ne10:  $next(currInd) \neq 0 \wedge k < h.size \Rightarrow h.table[k] \notin \{\mathbf{del}, \mathbf{done}\},$   
 where  $h = H(next(currInd))$

- Ne11:  $\text{next}(\text{currInd}) \neq 0 \wedge k < H(\text{next}(\text{currInd})).\text{size}$   
 $\Rightarrow \neg \text{oldp}(H(\text{next}(\text{currInd})).\text{table}[k])$
- Ne12:  $k < \text{curSize} \wedge H(\text{currInd}).\text{table}[k] = \text{done} \wedge m < h.\text{size} \wedge \text{LeastFind}(h, a, m)$   
 $\Rightarrow X(a) = \text{val}(h.\text{table}[\text{key}(a, h.\text{size}, m)]),$   
 where  $a = \text{ADR}(Y[k])$  and  $h = H(\text{next}(\text{currInd}))$
- Ne13:  $k < \text{curSize} \wedge H(\text{currInd}).\text{table}[k] = \text{done} \wedge m < h.\text{size}$   
 $\wedge X(a) = \text{val}(h.\text{table}[\text{key}(a, h.\text{size}, m)]) \neq \text{null}$   
 $\Rightarrow \text{LeastFind}(h, a, m),$   
 where  $a = \text{ADR}(Y[k])$  and  $h = H(\text{next}(\text{currInd}))$
- Ne14:  $\text{next}(\text{currInd}) \neq 0 \wedge a \neq 0 \wedge k < h.\text{size} \wedge X(a) = \text{val}(h.\text{table}[\text{key}(a, h.\text{size}, k)]) \neq \text{null}$   
 $\Rightarrow \text{LeastFind}(h, a, k),$   
 where  $h = H(\text{next}(\text{currInd}))$
- Ne15:  $k < \text{curSize} \wedge H(\text{currInd}).\text{table}[k] = \text{done} \wedge X(a) \neq \text{null}$   
 $\wedge m < h.\text{size} \wedge X(a) = \text{val}(h.\text{table}[\text{key}(a, h.\text{size}, m)]) \wedge n < h.\text{size} \wedge m \neq n$   
 $\Rightarrow \text{ADR}(h.\text{table}[\text{key}(a, h.\text{size}, n)]) \neq a,$   
 where  $a = \text{ADR}(Y[k])$  and  $h = H(\text{next}(\text{currInd}))$
- Ne16:  $k < \text{curSize} \wedge H(\text{currInd}).\text{table}[k] = \text{done} \wedge X(a) = \text{null} \wedge m < h.\text{size}$   
 $\Rightarrow \text{val}(h.\text{table}[\text{key}(a, h.\text{size}, m)]) = \text{null}$   
 $\vee \text{ADR}(h.\text{table}[\text{key}(a, h.\text{size}, m)]) \neq a,$   
 where  $a = \text{ADR}(Y[k])$  and  $h = H(\text{next}(\text{currInd}))$
- Ne17:  $\text{next}(\text{currInd}) \neq 0 \wedge m < h.\text{size} \wedge a = \text{ADR}(h.\text{table}[m]) \neq 0$   
 $\Rightarrow X(a) = \text{val}(h.\text{table}[m]) \neq \text{null},$   
 where  $h = H(\text{next}(\text{currInd}))$
- Ne18:  $\text{next}(\text{currInd}) \neq 0 \wedge m < h.\text{size} \wedge a = \text{ADR}(h.\text{table}[m]) \neq 0$   
 $\Rightarrow \exists n < \text{curSize} : \text{val}(Y[n]) = \text{val}(h.\text{table}[m]) \wedge \text{oldp}(H(\text{currInd}).\text{table}[n]),$   
 where  $h = H(\text{next}(\text{currInd}))$
- Ne19:  $\text{next}(\text{currInd}) \neq 0 \wedge m < h.\text{size} \wedge m \neq n < h.\text{size}$   
 $\wedge a = \text{ADR}(h.\text{table}[\text{key}(a, h.\text{size}, m)]) \neq 0$   
 $\Rightarrow \text{ADR}(h.\text{table}[\text{key}(a, h.\text{size}, n)]) \neq a,$   
 where  $h = H(\text{next}(\text{currInd}))$
- Ne20:  $k < \text{curSize} \wedge H(\text{currInd}).\text{table}[k] = \text{done} \wedge X(a) \neq \text{null}$   
 $\Rightarrow \exists m < h.\text{size} : X(a) = \text{val}(h.\text{table}[\text{key}(a, h.\text{size}, m)]),$   
 where  $a = \text{ADR}(Y[k])$  and  $h = H(\text{next}(\text{currInd}))$
- Ne21: Ne21 has been eliminated.
- Ne22:  $\text{next}(\text{currInd}) \neq 0 \Rightarrow \sharp(\text{ocSet6}) = H(\text{next}(\text{currInd})).\text{occ} + \sharp(\text{ocSet7})$
- Ne23:  $\text{next}(\text{currInd}) \neq 0$   
 $\Rightarrow H(\text{next}(\text{currInd})).\text{occ} \leq H(\text{next}(\text{currInd})).\text{bound}$

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

Ne25:  $\text{next}(\text{currInd}) \neq 0$

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

Ne26:  $\text{next}(\text{currInd}) \neq 0$

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

Ne27:  $\text{next}(\text{currInd}) \neq 0 \wedge (\exists n < h.\text{size} : \text{val}(h.\text{table}[n]) \neq \text{null})$

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

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

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

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

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

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

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

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

fi7:  $pc = 13 \wedge h_{fi} = H(\text{currInd}) \wedge \neg \text{Find}(r_{fi}, a_{fi}) \wedge m < n_{fi}$   
 $\Rightarrow \neg \text{Find}(Y[\text{key}(a_{fi}, \text{curSize}, m)], a_{fi})$

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

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

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

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

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

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

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

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

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

de5:  $pc \in \{16, 17\} \vee \text{Deleting} \Rightarrow \neg \text{suc}_{del}$

de6:  $\text{Deleting} \wedge \text{suc}_{del} \Rightarrow r_{del} \neq \text{null}$

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

- de8:  $pc \in \{17, 18\} \wedge h_{del} = H(\text{currInd}) \Rightarrow n_{del} < \text{curSize}$   
de9:  $pc = 18 \wedge h_{del} = H(\text{currInd}) \wedge (\text{val}(r_{del}) \neq \text{null} \vee r_{del} = \text{del})$   
 $\Rightarrow r \neq \text{null} \wedge (r = \text{del} \vee \text{ADR}(r) = \text{ADR}(r_{del})),$   
where  $r = Y[\text{key}(a_{del}, h_{del}.\text{size}, n_{del})]$   
de10:  $pc \in \{17, 18\} \wedge h_{del} = H(\text{currInd}) \wedge m < n_{del} \Rightarrow \neg \text{Find}(Y[\text{key}(a_{del}, \text{curSize}, m)], a_{del})$   
de11:  $pc \in \{17, 18\} \wedge \text{Find}(t, a_{del}) \Rightarrow X(a_{del}) = \text{val}(t),$   
where  $t = h_{del}.\text{table}[\text{key}(a_{del}, l_{del}, n_{del})]$   
de12:  $pc = 18 \wedge \text{oldp}(r_{del}) \wedge \text{index} = \text{currInd} \Rightarrow \text{next}(\text{currInd}) \neq 0$   
de13:  $pc = 18 \Rightarrow k_{del} < H(\text{index}).\text{size}$

*Deleting* is characterized by

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

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

- in1:  $a_{ins} = \text{ADR}(v_{ins}) \wedge v_{ins} \neq \text{null}$   
in2:  $pc \in [32, 35] \Rightarrow l_{ins} = h_{ins}.\text{size}$   
in3:  $pc \in [28, 41] \wedge pc \notin \{30, 36\} \Rightarrow h_{ins} = H(\text{index})$   
in4:  $pc \in \{33, 35\} \Rightarrow k_{ins} = \text{key}(a_{ins}, l_{ins}, n_{ins})$   
in5:  $pc \in [32, 33] \vee \text{Inserting} \Rightarrow \neg \text{suc}_{ins}$   
in6:  $\text{Inserting} \wedge \text{suc}_{ins} \Rightarrow \text{ADR}(r_{ins}) \neq a_{ins}$   
in7:  $pc = 35 \wedge \neg \text{oldp}(h_{ins}.\text{table}[k_{ins}]) \Rightarrow h_{ins} = H(\text{currInd})$   
in8:  $pc \in \{33, 35\} \wedge h_{ins} = H(\text{currInd}) \Rightarrow n_{ins} < \text{curSize}$   
in9:  $pc = 35 \wedge h_{ins} = H(\text{currInd}) \wedge (\text{val}(r_{ins}) \neq \text{null} \vee r_{ins} = \text{del})$   
 $\Rightarrow r \neq \text{null} \wedge (r = \text{del} \vee \text{ADR}(r) = \text{ADR}(r_{ins})),$   
where  $r = Y[\text{key}(a_{ins}, h_{ins}.\text{size}, n_{ins})]$   
in10:  $pc \in \{32, 33, 35\} \wedge h_{ins} = H(\text{currInd}) \wedge m < n_{ins}$   
 $\Rightarrow \neg \text{Find}(Y[\text{key}(a_{ins}, \text{curSize}, m)], a_{ins})$   
in11:  $pc \in \{33, 35\} \wedge \text{Find}(t, a_{ins}) \Rightarrow X(a_{ins}) = \text{val}(t),$   
where  $t = h_{ins}.\text{table}[\text{key}(a_{ins}, l_{ins}, n_{ins})]$   
in12:  $pc = 35 \wedge \text{oldp}(r_{ins}) \wedge \text{index} = \text{currInd} \Rightarrow \text{next}(\text{currInd}) \neq 0$   
in13:  $pc = 35 \Rightarrow k_{ins} < H(\text{index}).\text{size}$

*Inserting* is characterized by

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

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

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

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

- rA1:  $h_{rA} < \mathbf{H\_index}$
- rA2:  $pc \in [70, 71] \Rightarrow h_{rA} \neq 0$
- rA3:  $pc = 71 \Rightarrow \mathbf{Heap}(h_{rA}) \neq \perp$
- rA4:  $pc = 71 \Rightarrow \mathbf{H}(i_{rA}) = 0$
- rA5:  $pc = 71 \Rightarrow h_{rA} \neq \mathbf{H}(i)$
- rA6:  $pc = 70 \Rightarrow \mathbf{H}(i_{rA}) \neq \mathbf{H}(\text{currInd})$
- rA7:  $pc = 70 \wedge (pc.r \in [1, 58] \vee pc.r > 65 \wedge \neg(pc.r \in [67, 72] \wedge i_{rA}.r = \text{index}.r))$   
 $\Rightarrow \mathbf{H}(i_{rA}) \neq \mathbf{H}(\text{index}.r)$
- rA8:  $pc = 70 \Rightarrow i_{rA} \neq \text{next}(\text{currInd})$
- rA9:  $pc \in [68, 72] \wedge (h_{rA} = 0 \vee h_{rA} \neq \mathbf{H}(i_{rA})) \Rightarrow \mathbf{H}(i_{rA}) = 0$
- rA10:  $pc \in [67, 72] \wedge \text{return}_{rA} \in \{0, 59\} \Rightarrow i_{rA} = \text{index}$
- rA11:  $pc \in [67, 72] \wedge \text{return}_{rA} \in \{77, 90\} \Rightarrow i_{rA} \neq \text{index}$
- rA12:  $pc \in [67, 72] \wedge \text{return}_{rA} = 77 \Rightarrow \text{next}(\text{index}) \neq 0$
- 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:  $pc \in [81, 82] \Rightarrow \mathbf{Heap}(\mathbf{H}(i_{nT})) = \perp$
- nT2:  $pc \in [83, 84] \Rightarrow \mathbf{Heap}(\mathbf{H}(i_{nT})) \neq \perp$

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

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

$mi1: pc = 98 \vee pc \in \{104, 105\} \Rightarrow \text{index} \neq \text{currInd}$   
 $mi2: pc \geq 95 \Rightarrow i_{mig} \neq \text{index}$   
 $mi3: pc = 94 \Rightarrow \text{next}(\text{index}) > 0$   
 $mi4: pc \geq 95 \Rightarrow i_{mig} \neq 0$   
 $mi5: pc \geq 95 \Rightarrow i_{mig} = \text{next}(\text{index})$   
 $mi6: pc.r = 70 \wedge (pc \in [95, 102] \wedge \text{index} = \text{currInd} \vee pc \in [102, 103] \vee pc \geq 110)$   
 $\quad \Rightarrow i_{rA}.r \neq i_{mig}$   
 $mi7: pc \in [95, 97] \wedge \text{index} = \text{currInd} \vee pc \geq 99 \Rightarrow i_{mig} \neq \text{next}(i_{mig})$   
 $mi8: (pc \in [95, 97] \vee pc \in [99, 103] \vee pc \geq 110) \wedge \text{index} = \text{currInd}$   
 $\quad \Rightarrow \text{next}(i_{mig}) = 0$   
 $mi9: (pc \in [95, 103] \vee pc \geq 110) \wedge \text{index} = \text{currInd} \Rightarrow H(i_{mig}) \neq H(\text{currInd})$   
 $mi10: (pc \in [95, 103] \vee pc \geq 110) \wedge \text{index} = \text{currInd} \wedge (pc.r \in [1, 58] \vee pc.r \geq 62 \wedge pc.r \neq 65)$   
 $\quad \Rightarrow H(i_{mig}) \neq H(\text{index}.r)$   
 $mi11: pc = 101 \wedge \text{index} = \text{currInd} \vee pc = 102 \Rightarrow h_{mig} = H(i_{mig})$   
 $mi12: pc \geq 95 \wedge \text{index} = \text{currInd} \vee pc \in \{102, 103\} \vee pc \geq 110 \Rightarrow \text{Heap}(H(i_{mig})) \neq \perp$   
 $mi13: pc = 103 \wedge \text{index} = \text{currInd} \wedge k < \text{curSize} \Rightarrow H(\text{index}).\text{table}[k] = \text{done}$   
 $mi14: pc = 103 \wedge \text{index} = \text{currInd} \wedge n < H(i_{mig}).\text{size} \wedge \text{LeastFind}(H(i_{mig}), a, n)$   
 $\quad \Rightarrow X(a) = \text{val}(H(i_{mig})[\text{key}(a, H(i_{mig}).\text{size}, n)])$

- mi15:  $pc = 103 \wedge index = \text{currInd} \wedge n < H(i_{mig}).\text{size}$   
 $\wedge X(a) = \text{val}(H(i_{mig}).\text{table}[\text{key}(a, H(i_{mig}).\text{size}, n)]) \neq \text{null}$   
 $\Rightarrow \text{LeastFind}(H(i_{mig}), a, n)$
- mi16:  $pc = 103 \wedge index = \text{currInd} \wedge k < H(i_{mig}).\text{size}$   
 $\Rightarrow \neg \text{oldp}(H(i_{mig}).\text{table}[k])$
- mi17:  $pc = 103 \wedge index = \text{currInd} \wedge X(a) \neq \text{null} \wedge k < h.\text{size}$   
 $\wedge X(a) = \text{val}(h.\text{table}[\text{key}(a, h.\text{size}, k)]) \wedge k \neq n < h.\text{size}$   
 $\Rightarrow \text{ADR}(h.\text{table}[\text{key}(a, h.\text{size}, n)]) \neq a,$   
 where  $h = H(i_{mig})$
- mi18:  $pc = 103 \wedge index = \text{currInd} \wedge X(a) = \text{null} \wedge k < h.\text{size}$   
 $\Rightarrow \text{val}(h.\text{table}[\text{key}(a, h.\text{size}, k)]) = \text{null} \vee \text{ADR}(h.\text{table}[\text{key}(a, h.\text{size}, k)]) \neq a,$   
 where  $h = H(i_{mig})$
- mi19:  $pc = 103 \wedge index = \text{currInd} \wedge X(a) \neq \text{null}$   
 $\Rightarrow \exists m < h.\text{size} : X(a) = \text{val}(h.\text{table}[\text{key}(a, h.\text{size}, m)]),$   
 where  $h = H(i_{mig})$
- mi20:  $pc = 117 \wedge X(a) \neq \text{null} \wedge \text{val}(H(index).\text{table}[i_{mC}]) \neq \text{null}$   
 $\vee pc \geq 126 \wedge X(a) \neq \text{null} \wedge index = \text{currInd}$   
 $\vee pc = 125 \wedge X(a) \neq \text{null} \wedge index = \text{currInd}$   
 $\wedge (b_{mE} \vee \text{val}(w_{mE}) \neq \text{null} \wedge a_{mE} = \text{ADR}(w_{mE}))$   
 $\Rightarrow \exists m < h.\text{size} : X(a) = \text{val}(h.\text{table}[\text{key}(a, h.\text{size}, m)]),$   
 where  $a = \text{ADR}(Y[i_{mC}])$  and  $h = H(\text{next}(\text{currInd}))$

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

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

$$\begin{aligned}
mC12: & pc \geq 116 \wedge index = \text{currInd} \wedge val(v_{mC}) \neq \text{null} \\
& \Rightarrow val(v_{mC}) = val(Y[i_{mC}])
\end{aligned}$$

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

$$\begin{aligned}
mE1: & pc \geq 120 \Rightarrow val(v_{mC}) = v_{mE} \\
mE2: & pc \geq 120 \Rightarrow v_{mE} \neq \text{null} \\
mE3: & pc \geq 120 \Rightarrow to = H(i_{mig}) \\
mE4: & pc \geq 121 \Rightarrow a_{mE} = \text{ADR}(v_{mC}) \\
mE5: & pc \geq 121 \Rightarrow m_{mE} = to.size \\
mE6: & pc \in \{121, 123\} \Rightarrow \neg b_{mE} \\
mE7: & pc = 123 \Rightarrow k_{mE} = \text{key}(a_{mE}, to.size, n_{mE}) \\
mE8: & pc \geq 123 \Rightarrow k_{mE} < H(i_{mig}).size \\
mE9: & pc = 120 \wedge to.table[\text{key}(\text{ADR}(v_{mE}), to.size, 0)] = \text{null} \\
& \Rightarrow index = \text{currInd} \\
mE10: & pc \in \{121, 123\} \wedge to.table[\text{key}(a_{mE}, to.size, n_{mE})] = \text{null} \\
& \Rightarrow index = \text{currInd} \\
mE11: & pc \in \{121, 123\} \wedge pc.r = 103 \wedge to.table[\text{key}(a_{mE}, to.size, n_{mE})] = \text{null} \\
& \Rightarrow index.r \neq \text{currInd} \\
mE12: & pc \in \{121, 123\} \wedge \text{next}(\text{currInd}) \neq 0 \wedge to = H(\text{next}(\text{currInd})) \\
& \Rightarrow n_{mE} < H(\text{next}(\text{currInd})).size \\
mE13: & pc \in \{123, 125\} \wedge w_{mE} \neq \text{null} \\
& \Rightarrow \text{ADR}(w_{mE}) = \text{ADR}(to.table[k_{mE}]) \vee to.table[k_{mE}] \in \{\text{del}, \text{done}\} \\
mE14: & pc \geq 123 \wedge w_{mE} \neq \text{null} \Rightarrow H(i_{mig}).table[k_{mE}] \neq \text{null} \\
mE15: & pc = 117 \wedge val(v_{mC}) \neq \text{null} \vee pc \in \{121, 123\} \wedge n_{mE} > 0 \vee pc = 125 \\
& \Rightarrow h.table[\text{key}(\text{ADR}(v_{mC}), h.size, 0)] \neq \text{null}, \\
& \text{where } h = H(i_{mig}) \\
mE16: & pc \in \{121, 123\} \\
& \vee (pc = 125 \wedge \neg b_{mE} \wedge (val(w_{mE}) = \text{null} \vee a_{mE} \neq \text{ADR}(w_{mE}))) \\
& \Rightarrow \forall m < n_{mE} : \neg \text{Find}(to.table[\text{key}(a_{mE}, to.size, m)], a_{mE})
\end{aligned}$$

Invariants about the integer array *prot*.

$$\begin{aligned}
pr1: & prot[i] = \#(prSet1(i)) + \#(prSet2(i)) + \#(\text{currInd} = i) + \#(\text{next}(\text{currInd}) = i) \\
pr2: & prot[\text{currInd}] > 0 \\
pr3: & pc \in [1, 58] \vee pc \geq 62 \wedge pc \neq 65 \Rightarrow prot[index] > 0 \\
pr4: & \text{next}(\text{currInd}) \neq 0 \Rightarrow prot[\text{next}(\text{currInd})] > 0 \\
pr5: & prot[i] = 0 \Rightarrow \text{Heap}(H[i]) = \perp \\
pr6: & prot[i] \leq \#(prSet3(i)) \wedge busy[i] = 0 \Rightarrow \text{Heap}(H[i]) = \perp \\
pr7: & pc \in [67, 72] \Rightarrow prot[i_{rA}] > 0
\end{aligned}$$



$pr8: \quad pc \in [81, 84] \Rightarrow \mathbf{prot}[i_{nT}] > 0$   
 $pr9: \quad pc \geq 97 \Rightarrow \mathbf{prot}[i_{mig}] > 0$   
 $pr10: \quad pc \in [81, 82] \Rightarrow \mathbf{prot}[i_{nT}] = \#(prSet4(i_{nT})) + 1$

Invariants about the integer array **busy**.

$bu1: \quad \mathbf{busy}[i] = \#(buSet1(i)) + \#(buSet2(i)) + \#(\mathbf{currInd} = i) + \#(\mathbf{next}(\mathbf{currInd}) = i)$   
 $bu2: \quad \mathbf{busy}[\mathbf{currInd}] > 0$   
 $bu3: \quad pc \in [1, 58] \vee pc > 65 \wedge \neg(i_{rA} = \mathbf{index} \wedge pc \in [67, 72])$   
 $\quad \Rightarrow \mathbf{busy}[\mathbf{index}] > 0$   
 $bu4: \quad \mathbf{next}(\mathbf{currInd}) \neq 0 \Rightarrow \mathbf{busy}[\mathbf{next}(\mathbf{currInd})] > 0$   
 $bu5: \quad pc = 81 \Rightarrow \mathbf{busy}[i_{nT}] = 0$   
 $bu6: \quad pc \geq 100 \Rightarrow \mathbf{busy}[i_{mig}] > 0$

Some other invariants we have postulated:

$Ot1: \quad \mathbf{x}(0) = \mathbf{null}$   
 $Ot2: \quad \mathbf{x}(a) \neq \mathbf{null} \Rightarrow \mathbf{ADR}(\mathbf{x}(a)) = a$

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

$Ot3: \quad \mathbf{return}_{gA} = \{1, 10, 20, 30, 36, 46, 51\} \wedge \mathbf{return}_{rA} = \{0, 59, 77, 90\}$   
 $\quad \wedge \mathbf{return}_{ref} = \{10, 20, 30, 36, 46, 51\} \wedge \mathbf{return}_{nT} = \{30, 46\}$   
 $Ot4: \quad pc \in \{0, 1, 5, 6, 7, 8, 10, 11, 13, 14, 15, 16, 17, 18, 20, 21, 25, 26, 27, 28, 30,$   
 $\quad 31, 32, 33, 35, 36, 37, 41, 42, 43, 44, 46, 47, 48, 49, 50, 51, 52, 57, 59, 60,$   
 $\quad 61, 62, 63, 65, 67, 68, 69, 70, 71, 72, 77, 78, 81, 82, 83, 84, 90, 94, 95, 97,$   
 $\quad 98, 99, 100, 101, 102, 103, 104, 105, 110, 111, 114, 116, 117, 118, 120, 121,$   
 $\quad 123, 125, 126\}$

## A.2 Dependencies between invariants

Let us write “ $\varphi$  **from**  $\psi_1, \dots, \psi_n$ ” to denote that  $\varphi$  is proved to be an invariant using that  $\psi_1, \dots, \psi_n$  hold. We write “ $\varphi \Leftarrow \psi_1, \dots, \psi_n$ ” to denote that predicate  $\varphi$  is implied by the conjunction of  $\psi_1, \dots, \psi_n$ . We have verified the following “**from**” and “ $\Leftarrow$ ” relations mechanically:

$\mathbf{Co1}$  **from**  $\mathbf{fi10}$ ,  $\mathbf{Ot3}$ ,  $\mathbf{fi1}$   
 $\mathbf{Co2}$  **from**  $\mathbf{de5}$ ,  $\mathbf{Ot3}$ ,  $\mathbf{de6}$ ,  $\mathbf{del}$ ,  $\mathbf{de11}$   
 $\mathbf{Co3}$  **from**  $\mathbf{in5}$ ,  $\mathbf{Ot3}$ ,  $\mathbf{in6}$ ,  $\mathbf{in1}$ ,  $\mathbf{in11}$

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  $\leftarrow$  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, mi4,  
 mE3, Ne3, de7, in7, as5, Ne22, mi9, mi10, rA5, He3, mi12, mi1, Cu9, de1, in1, as1  
 Cu8 **from** Cu8, 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  $\Leftarrow$  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  $\Leftarrow$  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, 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, 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, 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, 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  $\Leftarrow$  Cu6, cu7, Ne6, Ne7, He3, He4, Ne22, He6  
 Ne24  $\Leftarrow$  Ne27, He6  
 Ne25  $\Leftarrow$  Ne19, Ne17, Ne18, He6  
 Ne26  $\Leftarrow$  Ne17, Ne18, He6  
 Ne27  $\Leftarrow$  Cu16, Ne25, Ne26, Ne17, Ne18, He6  
 fi1, del, 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  $\Leftarrow$  Cu2, de10, in10, as8, fi8, He3, He4  
 fi6 **from** Ot3, fi1, del, 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, del, 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, del, in1, as1, rA6, rA7, Ha2, nT13, nT12, mi9, mi10, Cu9, Cu10,  
 He3, He4, Cu1, Ha4, in3, as3, rA5  
 fi9  $\Leftarrow$  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  $\Leftarrow$  de3, in3, as3, Cu1, Ha4, de13, in13, as11  
 de9 **from** Ot3, del, in1, as1, rA6, rA7, Ha2, nT13, nT12, mi9, mi10, Cu9, Cu10, de3, de7,  
 in7, as5, rA5  
 de10 **from** de3, de9, Ot3, del, in1, as1, rA6, rA7, Ha2, nT13, nT12, mi9, mi10, Cu9, Cu10,  
 de7, in7, as5, He3, He4, rA5  
 de11  $\Leftarrow$  de10, de2, de3, He3, He4, Cu1, Ha4, Cu9, Cu10, Cu11, Cu12, fi5, de8, in8, as6  
 de13, in13, as11  $\Leftarrow$  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, del, in1, as1, rA6, rA7, Ha2, nT13, nT12, mi9, mi10, Cu9, Cu10, He3, He4,  
 in3, de7, in7, as5, rA5  
 in10 **from** in9, fi2, Ot3, del, in1, as1, rA6, rA7, Ha2, nT13, nT12, mi9, mi10, Cu9, Cu10,  
 He3, He4, in3, de7, in7, as5, rA5  
 in11  $\Leftarrow$  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, del, in1, as1, rA6, rA7, Ha2, nT13, nT12, mi9, mi10, Cu9, Cu10, as3, de7,  
 in7, as5, rA5  
 as8 **from** as7, Ot3, del, in1, as1, rA6, rA7, Ha2, nT13, nT12, mi9, mi10, Cu9, Cu10, He3,  
 He4, as3, de7, in7, as5, rA5  
 as9  $\Leftarrow$  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



# Appendix B

## For lock-free parallel GC

### B.1 Invariants

In the invariants and the lemmas given below, we use the relations  $R(x)$ ,  $R(p, x)$ ,  $R1(p, x)$ ,  $\triangleright_q$  defined in sections 5.2, 5.2, 5.3.1 and 5.4.2, respectively. The relation  $\xrightarrow{*}$  is defined in section 5.2. The relation  $\xrightarrow{M*}$  is the reflexive transitive closure of relation  $\xrightarrow{M}$  on nodes defined by:

$$z \xrightarrow{M} x \equiv (\text{color}[z] = \text{black} \wedge \text{aux}[z] \wedge \exists k: 1 \dots \text{arity}[z]: \text{child}[z, k] = x) \\ \vee (\text{color}[z] = \text{grey} \wedge \exists k: 1 \dots \text{ari}[z]: \text{child}[z, k] = x)$$

We define the  $j$ -th ancestor of a node  $x$  by the recursive function:

$$\text{anc}(x, j) \equiv ((j = 0 \vee \text{father}[x] \leq 0) ? x : \text{anc}(\text{father}[x], j - 1))$$

#### Main invariants:

- I1:  $\text{color}[x] = \text{white} \Rightarrow \neg R(x)$
- I2:  $\text{color}[x] = \text{white} \equiv x \in \text{free}$
- I3:  $554 \leq pc_p \leq 559 \Rightarrow x_p \in \text{roots}_p$
- I4:  $\neg(\exists p: \text{rnd}_p = \text{shRnd}) \Rightarrow \text{color}[x] \neq \text{grey}$
- I5:  $\text{srcnt}[x] - \text{freecnt}[x] = \sharp(\{p \mid x \in \text{roots}_p\}) + \sharp(\{(p, q) \mid$   
 $(\text{Mbox}(p, q) = x \wedge \neg(pc_q = 559 \wedge p = r_q)) \vee (pc_p = 508 \wedge x_p = x \wedge q = r_p)\})$

#### Invariants about the stability of the preconditions in the offered procedures:

- 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:  $300 \leq pc_p \leq 308 \vee (100 \leq pc_p \leq 180 \wedge return_p = 300) \Rightarrow \forall k: 1 \dots n_p: R(p, c_p[k])$   
 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:  $550 \leq pc_p \leq 559 \Rightarrow \text{Mbox}[r_p, p] \neq 0$

### Invariants that hold globally:

- I12:  $rnd_p \leq \text{shRnd}$   
 I13:  $\text{shRnd} \leq \text{round}[x] \leq \text{shRnd} + 1$   
 I14:  $\neg \text{aux}[x] \Rightarrow \text{round}[x] = \text{shRnd} + 1$   
 I15:  $\neg(\exists p: rnd_p = \text{shRnd}) \Rightarrow \text{round}[x] \leq \text{shRnd}$   
 I16:  $\text{round}[x] \leq \text{shRnd} \Rightarrow \text{color}[x] \neq \text{grey}$   
 I17:  $\text{color}[x] = \text{grey} \Rightarrow \neg \text{aux}[x]$   
 I18:  $\text{color}[x] = \text{white} \Rightarrow \neg R1(x)$   
 I19:  $\text{color}[x] = \text{white} \Rightarrow \text{father}[x] \leq -1$   
 I20:  $\text{color}[x] = \text{grey} \vee \text{father}[x] \geq 0 \Rightarrow \text{ari}[x] \leq \text{arity}[x]$   
 I21:  $\text{color}[x] = \text{grey} \wedge \text{father}[x] > 0$   
      $\Rightarrow \exists k: 1 \dots \text{ari}[\text{father}[x]]: \text{child}[\text{father}[x], k] = x$   
 I22:  $x \neq \text{father}[x]$   
 I23:  $\text{father}[x] > 0 \Rightarrow \neg(\exists j: \mathbb{N}: x = \text{anc}(x, j))$   
 I24:  $\text{father}[x] = 0 \wedge \text{color}[x] = \text{grey} \Rightarrow \text{srcnt}[x] > 0$   
 I25:  $(\exists p: x \in \text{roots}_p) \Rightarrow \text{srcnt}[x] - \text{freecnt}[x] > 0$   
 I26:  $\neg R(x) \Rightarrow \text{srcnt}[x] - \text{freecnt}[x] = 0$   
 I27:  $R1(x) \wedge (\text{color}[x] = \text{grey} \vee (\text{color}[x] = \text{black} \wedge \text{aux}[x]))$   
      $\Rightarrow \exists w: \text{srcnt}[w] > 0 \wedge (\text{father}[w] = 0 \vee \text{aux}[w]) \wedge w \xrightarrow{M*} x$   
 I28:  $return_p = 200 \vee return_p = 300 \vee return_p = 450$

### Invariants about the first phase of GC:

- I29:  $101 \leq pc_p \leq 110 \wedge rnd_p = \text{shRnd} \wedge \neg(x \in \text{toBeC}_p \wedge \text{aux}[x])$   
      $\Rightarrow \text{round}[x] = rnd_p + 1$   
 I30:  $101 \leq pc_p \leq 110 \wedge rnd_p = \text{shRnd} \wedge \neg(\exists r: \neg(101 \leq pc_r \leq 110) \wedge rnd_r = \text{shRnd})$   
      $\Rightarrow \text{father}[x] \neq 0 \vee \text{aux}[x] \vee \text{color}[x] \neq \text{black}$   
 I31:  $101 \leq pc_p \leq 110 \wedge rnd_p = \text{shRnd} \wedge \neg(\exists r: \neg(101 \leq pc_r \leq 110) \wedge rnd_r = \text{shRnd})$   
      $\wedge (x \notin \text{toBeC}_p \vee \text{round}[x] = rnd_p + 1)$   
      $\Rightarrow \neg \text{aux}[x]$   
 I32:  $101 \leq pc_p \leq 110 \wedge rnd_p = \text{shRnd} \wedge \neg(\exists r: \neg(101 \leq pc_r \leq 110) \wedge rnd_r = \text{shRnd})$

$$\begin{aligned}
& \wedge \neg(x \in toBeC_p \wedge aux[x]) \\
& \Rightarrow father[x] = 0 \vee father[x] = -1 \\
I33: & 101 \leq pc_p \leq 110 \wedge rnd_p = shRnd \wedge \neg(\exists r: \neg(101 \leq pc_r \leq 110) \wedge rnd_r = shRnd) \\
& \wedge (x \notin toBeC_p \vee round[x] = rnd_p + 1) \wedge srcnt[x] = 0 \\
& \Rightarrow father[x] = -1
\end{aligned}$$

### Invariants about the second phase of GC:

$$\begin{aligned}
I34: & \neg(101 \leq pc_p \leq 110) \wedge rnd_p = shRnd \Rightarrow round[x] = rnd_p + 1 \\
I35: & \neg(101 \leq pc_p \leq 110) \wedge rnd_p = shRnd \Rightarrow \neg aux[x] \\
I36: & \neg(101 \leq pc_p \leq 110) \wedge rnd_p = shRnd \wedge father[w] \leq -1 \Rightarrow \neg(\exists x: father[x] = w) \\
I37: & \neg(101 \leq pc_p \leq 110) \wedge rnd_p = shRnd \wedge color[x] = grey \wedge father[x] > 0 \\
& \Rightarrow color[father[x]] = grey \wedge father[father[x]] \geq 0 \\
I38: & \neg(101 \leq pc_p \leq 110) \wedge rnd_p = shRnd \wedge father[x] > 0 \wedge color[x] = grey \\
& \Rightarrow \exists j: 1 \dots N: father[anc(x, j)] = 0 \wedge color[anc(x, j)] = grey \\
I39: & \neg(101 \leq pc_p \leq 110) \wedge rnd_p = shRnd \wedge \neg RI(x) \wedge color[x] = grey \\
& \Rightarrow father[x] = -1 \\
I40: & \neg(101 \leq pc_p \leq 110) \wedge rnd_p = shRnd \wedge color[w] = black \wedge father[w] \geq 0 \\
& \Rightarrow \forall k: 1 \dots ari[w]: (father[child[w, k]] = w \vee father[child[w, k]] < 0 \\
& \Rightarrow color[child[w, k]] = black) \\
I41: & pc_p = 121 \Rightarrow rnd_p \neq shRnd \vee toBeC_p = \emptyset \\
I42: & \neg(101 \leq pc_p \leq 121) \wedge rnd_p = shRnd \wedge srcnt[x] > 0 \wedge father[x] = 0 \\
& \wedge \neg(x \in toBeD_p \vee (150 \leq pc_p \leq 180 \wedge x = x_p)) \\
& \Rightarrow color[x] = black \\
I43: & (122 \leq pc_p \leq 127 \vee 150 \leq pc_p \leq 180) \wedge rnd_p = shRnd \wedge x \notin toBeC_p \\
& \Rightarrow father[x] \geq 0 \\
I44: & ((122 \leq pc_p \leq 127 \vee 150 \leq pc_p \leq 180) \wedge rnd_p = shRnd \wedge x \in toBeD_p \\
& \wedge father[x] = 0) \vee (pc_p = 150 \wedge rnd_p = shRnd \wedge x = x_p) \\
& \Rightarrow x \in toBeC_p \\
I45: & (122 \leq pc_p \leq 127 \vee 150 \leq pc_p \leq 180) \wedge rnd_p = shRnd \wedge x \notin toBeC_p \\
& \Rightarrow \neg(\exists w: father[x] = w \wedge (w \in set_p \vee w \in toBeC_p)) \\
I46: & 122 \leq pc_p \leq 180 \wedge rnd_p = shRnd \\
& \wedge \neg(x \in toBeC_p \vee (pc_p \geq 151 \wedge (x \in set_p \vee \exists i: 1 \dots head_p: x = stack_p[i]))) \\
& \Rightarrow color[x] \neq grey
\end{aligned}$$

### Invariants about procedure Mark\_stack:

$$I47: pc_p = 150 \Rightarrow x_p \notin toBeD_p$$

- I48:  $150 \leq pc_p \leq 180 \wedge rnd_p = \text{shRnd}$   
 $\Rightarrow (\text{color}[x_p] = \text{grey} \wedge \text{father}[x_p] = 0 \wedge \text{srcnt}[x_p] > 0)$   
 $\vee (\text{color}[x_p] = \text{black} \wedge \text{father}[x_p] = 0)$
- I49:  $151 \leq pc_p \leq 180 \wedge x \in \text{toBeC}_p$   
 $\Rightarrow \neg(x \in \text{set}_p \vee \exists i: 1 \dots \text{head}_p: x = \text{stack}_p[i])$
- I50:  $151 \leq pc_p \leq 180 \wedge (\exists i: 1 \dots \text{head}_p: x = \text{stack}_p[i])$   
 $\Rightarrow x \notin \text{set}_p \wedge x \notin \text{toBeC}_p$
- I51:  $151 \leq pc_p \leq 180 \wedge rnd_p = \text{shRnd}$   
 $\Rightarrow x_p \in \text{set}_p \vee \text{color}[x_p] = \text{black} \vee (\exists i: 1 \dots \text{head}_p: x_p = \text{stack}_p[i])$
- I52:  $151 \leq pc_p \leq 180 \wedge rnd_p = \text{shRnd} \wedge (x \in \text{set}_p \vee \exists i: 1 \dots \text{head}_p: x = \text{stack}_p[i])$   
 $\Rightarrow (\text{color}[x] = \text{grey} \vee \text{color}[x] = \text{black}) \wedge \text{father}[x] \geq 0$
- I53:  $151 \leq pc_p \leq 180 \wedge rnd_p = \text{shRnd} \wedge \text{color}[x] = \text{grey}$   
 $\wedge (x \in \text{set}_p \vee \exists i: 1 \dots \text{head}_p: x = \text{stack}_p[i])$   
 $\Rightarrow R1(x) \wedge \text{father}[x] \geq 0$
- I54:  $151 \leq pc_p \leq 180 \wedge rnd_p = \text{shRnd} \wedge (\exists i: 1 \dots \text{head}_p: w = \text{stack}_p[i])$   
 $\Rightarrow \forall k: 1 \dots \text{ari}[w]: (\text{father}[\text{child}[w, k]] \geq 0 \vee \text{color}[\text{child}[w, k]] = \text{black})$   
 $\wedge (\text{father}[\text{child}[w, k]] = w \Rightarrow \text{child}[w, k] \in \text{set}_p \vee \text{color}[\text{child}[w, k]] = \text{black})$   
 $\vee ((\exists j: 1 \dots \text{head}_p: \text{child}[w, k] = \text{stack}_p[j]) \wedge (\forall m, n: 1 \dots \text{head}_p:$   
 $w = \text{stack}_p[m] \wedge \text{child}[w, k] = \text{stack}_p[n] \Rightarrow m < n)))$   
 $\vee (158 \leq pc_p \leq 164 \wedge w_p = w \wedge k \geq j_p)$
- I55:  $pc_p = 158 \wedge rnd_p = \text{shRnd} \Rightarrow j_p = 1 \vee 1 < j_p \leq \text{ari}[w_p] + 1$
- I56:  $158 \leq pc_p \leq 164 \wedge rnd_p = \text{shRnd}$   
 $\Rightarrow k_p = \text{ari}[w_p] \wedge \forall j: 1 \dots k_p: \text{ch}_p[j] = \text{child}[w_p, j]$
- I57:  $158 \leq pc_p \leq 164 \wedge rnd_p = \text{shRnd}$   
 $\wedge \neg(x \in \text{toBeC}_p \vee \exists j: 1 \dots j_p - 1: x = \text{child}[w_p, j])$   
 $\Rightarrow \text{father}[x] \neq w_p$
- I58:  $158 \leq pc_p \leq 164 \wedge rnd_p = \text{shRnd}$   
 $\Rightarrow \forall k: 1 \dots j_p - 1: (\text{color}[\text{child}[w_p, k]] = \text{grey} \Rightarrow \text{father}[\text{child}[w_p, k]] \geq 0)$   
 $\wedge (\text{father}[\text{child}[w_p, k]] = w_p \Rightarrow \text{child}[w_p, k] \in \text{set}_p))$
- I59:  $158 \leq pc_p \leq 165 \Rightarrow \exists i: 1 \dots \text{head}_p: w_p = \text{stack}_p[i]$
- I60:  $159 \leq pc_p \leq 164 \wedge rnd_p = \text{shRnd} \Rightarrow 1 \leq j_p \leq \text{ari}[w_p] \wedge y_p = \text{child}[w_p, j_p]$
- I61:  $168 \leq pc_p \leq 180 \Rightarrow rnd_p \neq \text{shRnd} \vee \text{set}_p = \emptyset$
- I62:  $170 \leq pc_p \leq 176 \Rightarrow \text{head}_p \neq 0$
- I63:  $pc_p = 180 \Rightarrow rnd_p \neq \text{shRnd} \vee \text{head}_p = 0$

### Invariants about the third phase of GC:

- I64:  $129 \leq pc_p \leq 137 \wedge rnd_p = \text{shRnd} \wedge \text{color}[x] = \text{grey} \Rightarrow \neg R1(x)$

- I65:  $pc_p = 134 \wedge \text{round}[x] = \text{rnd}_p + 1 \wedge \text{color}[x] = \text{grey}$   
 $\Rightarrow \neg R(x) \wedge x \notin \text{free}$
- I66:  $\neg(101 \leq pc_p \leq 134 \vee 150 \leq pc_p \leq 180) \wedge \text{rnd}_p = \text{shRnd}$   
 $\Rightarrow \text{color}[x] \neq \text{grey}$
- I67:  $pc_p = 135 \Rightarrow \text{rnd}_p \neq \text{shRnd} \vee \text{toBeC}_p = \emptyset$
- I68:  $\neg(101 \leq pc_p \leq 135 \vee 150 \leq pc_p \leq 180) \Rightarrow \text{rnd}_p \neq \text{shRnd}$

### Invariants outside GC:

- I69:  $pc_p = 450 \vee (100 \leq pc_p \leq 180 \wedge \text{return}_p = 450) \Rightarrow R(p, z_p)$
- I70:  $500 \leq pc_p \leq 508 \Rightarrow \text{Mbox}[p, r_p] = 0$
- I71:  $552 \leq pc_p \leq 559 \Rightarrow x_p = \text{Mbox}[r_p, p] \wedge x_p \neq 0$
- I72:  $pc_p = 558 \Rightarrow \text{srcnt}[x_p] > 1$

### Main lemmas:

- V1:  $p \neq q \wedge R(p, x) \wedge I18 \wedge I25 \triangleright_q R(p, x)$
- V2:  $\text{color}[x] \neq \text{white} \wedge \neg R1(x) \wedge I6 \wedge I8 \wedge I9 \wedge I25 \wedge I63 \wedge I66 \triangleright \neg R1(x)$

## B.2 Dependencies between invariants

Let us write “ $\varphi$  **from**  $\psi_1, \dots, \psi_n$ ” to denote that  $\varphi$  can be proved to be an invariant using that  $\psi_1, \dots, \psi_n$  hold in the precondition of every step. We write “ $\varphi \Leftarrow \psi_1, \dots, \psi_n$ ” to denote that  $\varphi$  can be directly derived from  $\psi_1, \dots, \psi_n$ . We have verified the following “**from**” and “ $\Leftarrow$ ” relations mechanically:

- I1  $\Leftarrow$  I3, I5, I18, I71
- I2 **from** : *true*
- I3 **from** : I28
- I4  $\Leftarrow$  I12, I15, I16
- I5 **from** : I18, I25, I28, I70, I71
- I6 **from** : I18, I25, I28
- I7 **from** : I18, I25, I28
- I8 **from** : I18, I25, I28
- I9 **from** : I18, I25, I28
- I10 **from** : I28
- I11 **from** : I28
- I12 **from** : *true*

**I13 from** : I12, I34  
**I14 from** : I12, I13  
**I15 from** : I12, I13, I34  
**I16 from** : I12, I13, I66  
**I17 from** : I66  
**I18 from** : I6, I8, I9, I12, I16, I25, I64, I69  
**I19 from** : I12, I16, I18, I39, I64  
**I20 from** : I19  
**I21 from** : I12, I13, I14, I15, I17, I20, I32, I34, I37, I60, I66  
**I22**  $\Leftarrow$  I23  
**I23 from** : I13, I16, I36, I50, I59  
**I24 from** : *true*  
**I25**  $\Leftarrow$  I5  
**I26**  $\Leftarrow$  I3, I5, I9  
**I27 from** : I6, I8, I9, I12, I13, I14, I16, I17, I18, I20, I21, I24, I25, I35, I37, I38, I54, I61,  
I64, I66, I69  
**I28 from** : *true*  
**I29 from** : I12, I13, I14, I28  
**I30 from** : I12, I13, I14, I15, I16, I19, I28, I68  
**I31 from** : I12, I13, I15, I28, I35, I68  
**I32 from** : I12, I13, I14, I15, I28, I29, I31, I34, I68  
**I33 from** : I12, I13, I15, I28, I29, I34, I68  
**I34 from** : I12, I13, I29, I34  
**I35 from** : I12, I31, I34  
**I36 from** : I12, I32, I34, I52, I59  
**I37 from** : I12, I21, I28, I32, I34, I39, I40, I52, I54, I59, I60, I61, I64  
**I38**  $\Leftarrow$  I23, I32, I35, I37  
**I39 from** : I12, I18, I19, I20, I21, I24, I27, I28, I33, I34, I35, I37, I38, I40, I52, I53, I54,  
I59, I60, I61  
**I40 from** : I12, I19, I20, I28, I30, I31, I32, I34, I35, I54, I61, I62  
**I41 from** : I12, I28  
**I42 from** : I9, I12, I18, I24, I25, I34, I42, I51, I61, I63  
**I43 from** : I12, I28, I34, I43, I48  
**I44 from** : I12, I28, I34, I35, I47  
**I45 from** : I12, I28, I43, I44, I48, I50, I59  
**I46 from** : I12, I34, I61, I63  
**I47 from** : I28

I48 **from** : I12, I18, I19, I24, I28, I34, I39, I64  
 I49 **from** : I28  
 I50 **from** : I28, I49  
 I51 **from** : I12, I28, I34, I52  
 I52 **from** : I12, I28, I34, I48, I53, I64  
 I53 **from** : I12, I18, I19, I20, I21, I24, I27, I28, I34, I35, I37, I38, I40, I48, I52, I54, I59,  
           I60, I61  
 I54 **from** : I12, I18, I19, I20, I22, I28, I34, I35, I40, I43, I50, I52, I53, I55, I56, I57, I58,  
           I60, I62  
 I55 **from** : I12, I28, I34, I56, I60  
 I56 **from** : I12, I20, I28, I34, I52, I59  
 I57 **from** : I12, I19, I20, I28, I34, I43, I45, I52, I55, I59, I60  
 I58 **from** : I12, I20, I28, I34, I35, I43, I52, I55, I56, I57, I59, I60  
 I59 **from** : I28  
 I60 **from** : I12, I20, I28, I34, I52, I56, I59  
 I61 **from** : I12, I28  
 I62 **from** : I28  
 I63 **from** : I12, I28  
 I64 **from** : I6, I8, I9, I12, I25, I27, I34, I35, I42  
 I65  $\Leftarrow$  I2, I3, I5, I12, I16, I64, I71  
 I66 **from** : I12, I34, I46  
 I67 **from** : I12, I28  
 I68 **from** : I12  
 I69 **from** : I18, I25  
 I70 **from** : I28  
 I71 **from** : I28, I70  
 I72  $\Leftarrow$  I3, I5, I71

## B.3 The low-level lock-free algorithm

### B.3.1 Data Structure

#### Constant

$P$  = number of processes;  
 $N$  = number of nodes;  
 $C$  = upper bound of number of children;

#### Type

colorType: {white, black, grey};

```

nodeType: record =
  arity:  $\mathbb{N}$ ;
  child: array  $[1 \dots C]$  of  $1 \dots N$ ;
  color: colorType;
  srcnt, freecnt, ari:  $\mathbb{N}$ ;
  father:  $\mathbb{N} \cup \{-1\}$ ;
  round:  $\mathbb{N}$ ;
end
Shared variables
node: array  $[1 \dots N + P]$  of nodeType;
indir: array  $[1 \dots N]$  of  $1 \dots N + P$ ;
Mbox: array  $[1 \dots P, 1 \dots P]$  of  $0 \dots N$ ;
shRnd:  $\mathbb{N}$ ;
Private variables
roots: a subset of  $1 \dots N$ ;
rnd:  $\mathbb{N}$ ;
toBeC: a subset of  $1 \dots N$ ;
mp:  $1 \dots N + P$ ;

```

**Initialization:**

```

shRnd = 1  $\wedge \forall x: 1 \dots N: (\text{indir}[x] = x \wedge \text{round}[\text{indir}[x]] = 1)$ ;
 $\forall p: 1 \dots P: mp_p = N + p$ ;
all other variables are equal to be the minimal values in their respective domains.

```

### B.3.2 Algorithm

```

proc GCollect() =
  local  $m: 1 \dots N + P$ ;  $x: 1 \dots N$ ; toBeD: a subset of  $1 \dots N$ ;
  % first phase
100: rnd := shRnd; toBeC :=  $\{1, \dots, N\}$ ;
101: while shRnd = rnd  $\wedge$  toBeC  $\neq \emptyset$  do
  choose  $x \in \text{toBeC}$ ;
  while true do
102:    $m := LL(\text{indir}[x])$ ;
103:   node[mp] := node[m];
104:   if round[mp] = rnd then
105:     round[mp] := rnd + 1; ari[mp] := arity[mp];
     if color[mp] = black then color[mp] := grey; fi;

```



```

        if srcnt[mp] > 0 then father[mp] := 0; else father[mp] := -1; fi;
106:    if SC(indir[x], mp) then toBeC := toBeC - {x}; mp := m; break; fi;
107:    elseif VL(indir[x]) then toBeC := toBeC - {x}; break; fi;
    od;
  od;
% second phase
110:  toBeC := {1, ..., N}; toBeD := {1, ..., N};
111:  while shRnd = rnd ∧ toBeD ≠ ∅ do
    choose x ∈ toBedone;
    while true do
112:      m := LL(indir[x]);
113:      node[mp] := node[m];
114:      if father[mp] = 0 then
116:        if VL(indir[x]) then
          toBeD := toBeD - {x};
          Mark_stack(x); break; fi;
117:        elseif VL(indir[x]) then toBeD := toBeD - {x}; break; fi;
        od;
      od;
    od;
  od;
% last phase
120:  while shRnd = rnd ∧ toBeC ≠ ∅ do
    choose x ∈ toBeC;
    while true do
121:      m := LL(indir[x]);
122:      node[mp] := node[m];
123:      if round[mp] = rnd + 1 ∧ color[mp] = grey then
124:        color[mp] := white;
125:        if SC(indir[x], mp) then toBeC := toBeC - {x}; mp := m; break; fi;
126:        elseif VL(indir[x]) then toBeC := toBeC - {x}; break; fi;
      od;
    od;
  od;
127:  CAS(shRnd, rnd, rnd + 1);
128:  return;
end GCollect.

```

```

proc Mark_stack(x: 1...N) =
  local w, y: 1...N; suc: Bool; j, k: ℕ;

```

```

stack: Stack; head:  $\mathbb{N}$ ; set: a subset of  $1 \dots N$ ;
ch:  $[1 \dots C]$  of  $1 \dots N$ ; m, n:  $1 \dots N + P$ ;
150: toBeC := toBeC - {x}; set := {x}; head := 0;
151: while shRnd = rnd  $\wedge$  set  $\neq \emptyset$  do
    choose  $w \in$  set;
    while true do
152:       m := LL(indir[w]);
153:       node[mp] := node[m];
154:       if color[mp] = grey  $\wedge$  round[mp] = rnd + 1 then
155:         k := ari[mp];
         for j := 1 to k do ch[j] := child[mp, j]; od;
156:       if VL(indir[w]) then
         set := set - {w}; head++; stack[head] := w; j := 1;
157:       while shRnd = rnd  $\wedge$  j  $\leq$  k do
         y := ch[j];
         if y  $\in$  toBeC then
           while true do
158:             n := LL(indir[y]);
159:             node[mp] := node[n];
160:             if (father[mp] = -1  $\vee$  father[mp] = w)
                $\wedge$  round[mp] = rnd + 1 then
161:               if father[mp] = -1 then father[mp] := w; fi;
162:               if SC(indir[y], mp) then
                 toBeC := toBeC - {y}; mp := n;
                 set := set + {y}; break; fi;
163:             elseif VL(indir[y]) then break; fi;
               od; fi;
             j := j + 1;
           od;
           break; fi;
164:       elseif VL(indir[w]) then set := set - {w}; break; fi;
    od;
  od;
170: while shRnd = rnd  $\wedge$  head  $\neq$  0 do
  y := stack[head];
  while true do
171:    m := LL(indir[y]);

```

```

172:     node[mp] := node[m];
173:     if color[mp] = grey ∧ round[mp] = rnd + 1 then
174:         color[mp] := black;
           srcnt[mp] := srcnt[mp] - freecnt[mp]; freecnt[mp] := 0;
175:         if SC(indir[y], mp) then mp := m; head--; break; fi;
176:         elseif VL(indir[y]) then head--; break; fi;
           od;
       od;
180: return;
end Mark_stack.

```

```

proc Create(): 1...N =
    local m: 1...N + P; x: 1...N;
    while true do
200:     choose x ∈ 1...N;
201:     m := LL(indir[x]);
202:     node[mp] = node[m];
203:     if color[mp] = white then
204:         color[mp] := black; srcnt[mp] := 1; arity[mp] := 0;
205:         if SC(indir[x], mp) then
           roots := roots + {x};
           mp := m; break; fi;
206:         elseif time to do GC then
           GCollect(); fi;
       od;
207: return x
end Create.

```

```

proc AddChild(x, y: 1...N): Bool =
    {R(self, x) ∧ R(self, y)}
    local m: 1...N + P; suc: Bool;
250: suc := false;
    while true do
251:     m := LL(indir[x]);
252:     node[mp] := node[m];
253:     if arity[mp] < C then
254:         arity[mp]++;

```

```

        child[mp, arity[mp]] := y;
255:    if SC(indir[x], mp) then
        mp := m; suc := true; break; fi;
256:    elseif VL(indir[x]) then break; fi;
    od;
257: return suc
end AddChild.

```

```

proc GetChild( $x: 1 \dots N$ ,  $rth: 1 \dots N$ ):  $0 \dots N =$ 
{ $R(self, x)$ }
    local  $m: 1 \dots N + P$ ;  $y: 1 \dots N$ ;
    while true do
280:     $m := LL(indir[x]);$ 
281:     $node[mp] := node[m];$ 
282:    if  $1 \leq rth \leq arity[mp]$  then  $y := child[mp, rth]$ ; else  $y := 0$ ; fi;
283:    if  $VL(indir[x])$  then break; fi;
    od;
284: return  $y$ 
end GetChild.

```

```

proc Make( $c: \text{array } [1 \dots C]$  of  $1 \dots N$ ,  $n: 1 \dots C$ ):  $1 \dots N =$ 
{ $\forall j: 1 \dots n: R(self, c[j])$ }
    local  $m: 1 \dots N + P$ ;  $x: 1 \dots N$ ;  $j: \mathbb{N}$ ;
    while true do
300:    choose  $x \in [1 \dots N]$ ;
301:     $m := LL(indir(x));$ 
302:     $node[mp] := node[m];$ 
303:    if  $color[mp] = white$  then
304:     $color[mp] := black;$ 
         $srcnt[mp] := 1$ ;  $arity[mp] := n$ ;
        for  $j := 1$  to  $n$  do  $child[mp, j] := c[j]$  od;
305:    if  $SC(indir(x), mp)$  then
         $roots := roots + \{x\}$ ;
         $mp := m$ ; break; fi;
306:    elseif time to do GC then
        GCCollect(); fi;
    od;

```

307: **return**  $x$

**end** *Make*.

**proc** *Protect*( $x: 1 \dots N$ ) =

$\{R(\text{self}, x) \wedge x \notin \text{roots}\}$

**local**  $m: 1 \dots N + P$ ;

**while true do**

400:      $m := LL(\text{indir}[x])$ ;

401:      $\text{node}[mp] := \text{node}[m]$ ;

402:      $\text{srcnt}[mp]++$ ;

403:     **if**  $SC(\text{indir}[x], mp)$  **then**

$\text{roots} := \text{roots} + \{x\}$ ;

$mp := m$ ; **break**; **fi**;

**od**;

404: **return**

**end** *Protect*.

**proc** *UnProtect*( $z: 1 \dots N$ ) =

$\{z \in \text{roots}\}$

**local**  $m: 1 \dots N + P$ ;

**while true do**

450:      $m := LL(\text{indir}[z])$ ;

451:      $\text{node}[mp] := \text{node}[m]$ ;

452:      $\text{freecnt}[mp]++$ ;

453:     **if**  $SC(\text{indir}[x], mp)$  **then**

$\text{roots} := \text{roots} \setminus \{z\}$ ;

$mp := m$ ; **break**; **fi**;

**od**;

454: **return**

**end** *UnProtect*.

**proc** *Send*( $x: 1 \dots N, r: 1 \dots P$ ) =

$\{R(\text{self}, x) \wedge \text{Mbox}[\text{self}, r] = 0\}$

**local**  $m: 1 \dots N + P$ ;

**while true do**

500:      $m := LL(\text{indir}[x])$ ;

501:      $\text{node}[mp] := \text{node}[m]$ ;

```

502:   srcnt[mp]++;
503:   if SC(indir[x], mp) then
       mp := m;
504:   Mbox[self, r] := x; break; fi;
       od;
505: return
end Send.

```

```

proc Receive(r: 1...P): 0...N =
  {Mbox[r, self] ≠ 0}
  local x: 1...N;
550: x := Mbox[r, self];
551: if x ∉ roots then
       roots := roots ∪ {x};
       Mbox[r, self] := 0;
  else
       while true do
552:   m := LL(indir[x]);
553:   node[mp] := node[m];
554:   srcnt[mp]--;
555:   if SC(indir[x], mp) then
       mp := m;
556:   Mbox[r, self] := 0;
       break; fi;
       od; fi;
557: return
end Receive.

```

```

proc Check(r, q: 1...P): Bool
  local suc: Bool;
600: suc := (Mbox[r, q] = 0);
601: return suc;
end Receive.

```

# Bibliography

- [1] M. Abadi and L. Lamport. The existence of refinement mappings. *Theor. Comput. Sci.*, 82(2):253–284, 1991.
- [2] J.H. Anderson, S. Ramamurthy, and K. Jeffay. Real-time computing with lock-free shared objects. *ACM Trans. Comput. Syst.*, 15(2):134–165, 1997.
- [3] H. Attiya, A. Bar-Noy, D. Dolev, D. Peleg, and R. Reischuk. Renaming in an asynchronous environment. *J. ACM*, 37(3):524–548, 1990.
- [4] H. Azatchi, Y. Levanoni, H. Paz, and E. Petrank. An on-the-fly mark and sweep garbage collector based on sliding views. In *Proceedings of the 18th ACM SIGPLAN conference on Object-oriented programing, systems, languages, and applications*, pages 269–281. ACM Press, 2003.
- [5] R.J.R. Back and J. von Wright. Stepwise refinement of distributed systems: Models, formalism, correctness: Refinement calculus. In J.W. de Bakker, W.-P. de Roever, and G. Rozenberg, editors, *Stepwise Refinement of Distributed Systems*, volume 430 of *Lecture Notes in Computer Science*, pages 42–93. Springer-Verlag, 1990.
- [6] G. Barnes. A method for implementing lock-free data structures. In *Proceedings of the 5th ACM Symposium on Parallel Algorithms and Architectures*, pages 261–270, June 1993.
- [7] A. Bas-Noy and D. Dolev. Shared-memory vs. message-passing in an asynchronous distributed environment. In *Proceedings of the eighth annual ACM Symposium on Principles of distributed computing*, pages 307–318, 1989.
- [8] M. Ben-Ari. Algorithms for on-the-fly garbage collection. *ACM Transactions on programming Languages and Systems*, 6(3):333–344, 1984.

- [9] B.N. Bershad. Practical considerations for non-blocking concurrent objects. In *Proceedings of the Thirteenth International Conference on Distributed Computing Systems*, pages 264–274, 1993.
- [10] H. Boehm, A. J. Demers, and S. Shenker. Mostly parallel garbage collection. In *Proceedings of the ACM SIGPLAN 1991 conference on Programming language design and implementation*, pages 157–164. ACM Press, 1991.
- [11] F. Cassez, C. Jard, B. Rozoy, and M.D. Ryan. *Modeling and verification of parallel processes*. Springer-Verlag New York, Inc., 2001.
- [12] K.M. Chandy and J. Misra. *Parallel program design: a foundation*. Addison-Wesley Longman Publishing Co., Inc., 1988.
- [13] E. Clarke, O. Grumberg, and D. Long. Model checking and abstraction. *ACM Transactions on Programming Languages and Systems*, 16(5):1512–1542, 1994.
- [14] C. Cornes, J. Courant, and et al. The coq proof assistant - reference manual v 6.1, 1997.
- [15] D.L. Detlefs, P.A. Martin, M. Moir, and G.L. Steele Jr. Lock-free reference counting. *Distributed Computing*, 15(4):255–71, December 2002.
- [16] E.W. Dijkstra, L. Lamport, A.J. Martin, C.S. Scholten, and E.F.M. Steffens. On-the-fly garbage collection: An exercise in cooperation. *Communications of the ACM*, 21(11):966–975, November 1978.
- [17] D. Doligez and X. Leroy. A concurrent generational garbage collector for a multi-threaded implementation of ml. In *Proceedings of the 1993 ACM Symposium on Principles of Programming Languages*, pages 113–123, January 1993.
- [18] T. Endo, K. Taura, and A. Yonezawa. A scalable mark-sweep garbage collector on large-scale shared-memory machines. In *Proceedings of the 1997 ACM/IEEE conference on Supercomputing (CDROM)*, pages 1–14. ACM Press, 1997.
- [19] C. Flood, D. Detlefs, N. Shavit, and C. Zhang. Parallel garbage collection for shared memory multiprocessors. In *Usenix Java Virtual Machine Research and Technology Symposium (JVM '01)*, Monterey, CA, April 2001.



- [20] H. Gao, J.F. Groote, and W.H. Hesselink. Almost wait-free resizable hashtables (extended abstract). In *Proceedings of 18th International Parallel & Distributed Processing Symposium (IPDPS)*. IEEE Computer Society, April 2004.
- [21] H. Gao, J.F. Groote, and W.H. Hesselink. Lock-free dynamic hash tables with open addressing. *Distributed Computing*, 2004. ISSN: 0178-2770 (Paper) 1432-0452 (Online) DOI: 10.1007/s00446-004-0115-2.
- [22] H. Gao, J.F. Groote, and W.H. Hesselink. Lock-free parallel garbage collection by mark&sweep. Technical Report CS-Report CSR-04-31, Eindhoven University of Technology, The Netherlands, 2004.
- [23] H. Gao and W.H. Hesselink. A formal reduction for lock-free parallel algorithms. In *Proceedings of the 16th Conference on Computer Aided Verification (CAV)*, July 2004.
- [24] J.F. Groote, W.H. Hesselink, S. Mauw, and R. Vermeulen. An algorithm for the asynchronous write-all problem based on process collision. *Distributed Computing*, 14:75–81, 2001.
- [25] S.P. Harbison. *Modula-3*. Prentice-Hall, Inc., 1992.
- [26] K. Havelund. Mechanical verification of a garbage collector. In José Rolim et al., editors, *Parallel and Distributed Processing (Combined Proceedings of 11 Workshops)*, volume 1586 of *Lecture Notes in Computer Science*, pages 1258–1283. Springer-Verlag, April 1999. Presented at the Workshop on Formal Methods for Parallel Programming: Theory and Applications (FMPPTA).
- [27] M. Herlihy. Wait-free synchronization. *ACM Transactions on Programming Languages and Systems*, 13(1):124–149, January 1991.
- [28] M. Herlihy. A methodology for implementing highly concurrent data objects. *ACM Transactions on Programming Languages and Systems*, 15(5):745–770, November 1993.
- [29] M.P. Herlihy, V. Luchangco, and M. Moir. The repeat offender problem: A mechanism for supporting dynamic-sized, lock-free data structure. In *Proceedings of 16th International Symposium on Distributed Computing*, pages 339–353. Springer-Verlag, October 2002.

- [30] M.P. Herlihy and J.E.B. Moss. Lock-free garbage collection for multiprocessors. *IEEE Transactions on Parallel and Distributed Systems*, 3(3):304–311, 1992.
- [31] M.P. Herlihy and J.M. Wing. Linearizability: a correctness condition for concurrent objects. *ACM Trans. Program. Lang. Syst.*, 12(3):463–492, 1990.
- [32] W.H. Hesselink. [http://www.cs.rug.nl/~wim/mechver/garbage\\_collection](http://www.cs.rug.nl/~wim/mechver/garbage_collection)
- [33] W.H. Hesselink. <http://www.cs.rug.nl/~wim/mechver/hashtable>
- [34] W.H. Hesselink. [http://www.cs.rug.nl/~wim/mechver/lockfree\\_reduction](http://www.cs.rug.nl/~wim/mechver/lockfree_reduction)
- [35] W.H. Hesselink. Wait-free linearization with a mechanical proof. *Distributed Computing*, 9:21–36, 1995.
- [36] W.H. Hesselink. Bounded delay for a free address. *Acta Informatica*, 33:233–254, 1996.
- [37] W.H. Hesselink. Using eternity variables to specify and prove a serializable database interface. *Science of Computer Programming*, 51(1-2):47–85, 2004.
- [38] W.H. Hesselink and J.F. Groote. Wait-free concurrent memory management by Create, and Read until Deletion. *Distributed Computing*, 14(1):31–39, January 2001.
- [39] R. L. Hudson and J. E. B. Moss. Sapphire: copying gc without stopping the world. In *ISCOPE Conference on ACM 2001 Java Grande*, pages 48–57. ACM Press, 2001.
- [40] L. Huelsbergen and J. R. Larus. A concurrent copying garbage collector for languages that distinguish (im)mutable data. In *Proceedings of the fourth ACM SIGPLAN symposium on Principles and practice of parallel programming*, pages 73–82. ACM Press, 1993.
- [41] IBM. *IBM System/370 Extended Architecture, Principles of Operation*, 1983.
- [42] E.H. Jensen, G.W. Hagensen, and J.M. Broughton. A new approach to exclusive data access in shared memory multiprocessors. Technical Report UCRL-97663, Lawrence Livermore National Laboratory, January 1987.
- [43] R. Jones. *Garbage Collection: Algorithms for Automatic Dynamic Memory Management*. John Wiley and Sons, July 1996. With a chapter on Distributed Garbage Collection by Rafael Lins. Reprinted 1997 (twice), 1999, 2000.

- [44] R. Jones and R. Lins. *Garbage collection: algorithms for automatic dynamic memory management*. John Wiley & Sons, Inc., 1996.
- [45] J.E. Jonker. On-the-fly garbage collection for several mutators. *Distributed Computing*, 5:187–199, 1992.
- [46] P.C. Kanellakis and A. A. Shvartsman. *Fault-Tolerant Parallel Computation*. Kluwer Academic Publishers, 1997.
- [47] D.E. Knuth. *The art of computer programming, volume 3: (2nd ed.) sorting and searching*. Addison Wesley Longman Publishing Co., Inc., 1998.
- [48] A. LaMarca. A performance evaluation of lock-free synchronization protocols. In *Proceedings of the thirteenth annual ACM symposium on Principles of distributed computing*, pages 130–140. ACM Press, 1994.
- [49] L. Lamport. The temporal logic of actions. *ACM Transactions on Programming Languages and Systems*, 16(3):872–923, 1994.
- [50] Y. Levanoni and E. Petrank. An on-the-fly reference counting garbage collector for java. In *Proceedings of the 16th ACM SIGPLAN conference on Object oriented programming, systems, languages, and applications*, pages 367–380. ACM Press, 2001.
- [51] V. Luchangco, M. Moir, and N. Shavit. Nonblocking k-compare-single-swap. In *Proceedings of the fifteenth annual ACM symposium on Parallel algorithms and architectures*, pages 314–323. ACM Press, 2003.
- [52] N.A. Lynch. *Distributed Algorithms*. Morgan Kaufmann Publishers, 1996.
- [53] Z. Manna and A. Pnueli. *The temporal logic of reactive and concurrent systems: Specification*. Springer-Verlag New York, Inc., 1992.
- [54] H. Massalin and C. Pu. A lock-free multiprocessor os kernel. Technical Report CUCS-005-91, Columbia University, 1991.
- [55] M.M. Michael. High performance dynamic lock-free hash tables and list-based sets. In *Proceedings of the fourteenth annual ACM symposium on Parallel algorithms and architectures*, pages 73–82. ACM Press, 2002.

- [56] M.M. Michael. Safe memory reclamation for dynamic lock-free objects using atomic reads and writes. In *Proceedings of the twenty-first annual symposium on Principles of distributed computing*, pages 21–30. ACM Press, 2002.
- [57] M. Moir. Practical implementations of non-blocking synchronization primitives. In *Proceedings of the sixteenth annual ACM symposium on Principles of distributed computing*, pages 219–228. ACM Press, 1997.
- [58] A.J. Mooij. Non-blocking implementations of LL, VL and SC. Private Communication, 2004.
- [59] L. Moreau and J. Duprat. A construction of distributed reference counting. *Acta Inf.*, 37(8):563–595, 2001.
- [60] Y. Ossia, O. Ben-Yitzhak, I. Goft, E.K. Kolodner, V. Leikehman, and A. Owshanko. A parallel, incremental and concurrent gc for servers. *SIGPLAN Not.*, 37(5):129–140, 2002.
- [61] J. O’Toole and S. Nettles. Concurrent replicating garbage collection. In *Proceedings of the 1994 ACM conference on LISP and functional programming*, pages 34–42. ACM Press, 1994.
- [62] S. Owicki and L. Lamport. Proving liveness properties of concurrent programs. *ACM Trans. Program. Lang. Syst.*, 4(3):455–495, 1982.
- [63] S. Owre, N. Shankar, J.M. Rushby, and D.W.J. Stringer-Calvert. *PVS Version 2.4: System Guide, Prover Guide, PVS Language Reference*, 2001.
- [64] R. Rajwar and J.R. Goodman. Transactional lock-free execution of lock-based programs. In *Proceedings of the 10th international conference on Architectural support for programming languages and operating systems*, pages 5–17. ACM Press, 2002.
- [65] H. Rodrigues and R. Jones. Cyclic distributed garbage collection with group merger. In *Proceedings of 12th European Conference on Object-Oriented Programming, ECOOP98*, pages 249–273, Brussels, July 1998. Springer.
- [66] O. Shalev and N. Shavit. Split-ordered lists: lock-free extensible hash tables. In *Proceedings of the twenty-second annual symposium on Principles of distributed computing*, pages 102–111. ACM Press, 2003.

- [67] H. Sundell and P. Tsigas. Scalable and lock-free concurrent dictionaries. In *Proceedings of the 2004 ACM Symposium on Applied computing*, pages 1438–1445, 2004.
- [68] J.D. Valois. Implementing lock-free queues. In *Proceedings of the Seventh International Conference on Parallel and Distributed Computing Systems*, pages 64–69, Las Vegas, NV, 1994.
- [69] J.D. Valois. Lock-free linked lists using compare-and-swap. In *Proceedings of the fourteenth annual ACM symposium on Principles of distributed computing*, pages 214–222. ACM Press, 1995.
- [70] N. Wirth. *Algorithms + Data Structures = Programs*. Prentice Hall PTR, 1978.

# Summary

In modern computers, many processes run concurrently, e.g. the window manager, the mailer, the word processor, the clock, etc. All these processes need access to the shared memory, but if they would read and modify the shared memory in unrestricted ways, the data would become inconsistent. The classical way to avoid this is to enforce synchronization by means of locks. This may lead to blocking.

Due to blocking, the classical synchronization paradigms using locks can incur many problems such as convoying, priority inversion and deadlock. Over the past two decades a number of researchers have proposed techniques for designing lock-free implementations. These techniques allow concurrent update of shared data structures without resorting to critical sections protected by locks. Essential for such implementations are using advanced machine instructions such as *LL/SC* or *CAS*. Lock-free implementations guarantee that within a finite number of steps always some process trying to perform an operation on the object will complete its task. They can avoid many problems arising due to failures and priority inversion. Lock-free synchronization is very important especially in real-time systems since if the blocked process is performing a high-priority or real-time task, it is highly undesirable to halt its progress.

Ensuring the correctness of the design at the earliest possible stage is a major challenge in any responsible system development. Lock-free algorithms are in general very complex and hard to design correctly. The only technique we see is to specify the programming model of the behavior of the system in a formal language, and to mathematically verify that the system design and implementation satisfy certain properties such as safety and liveness. In general there are two verification methods for system design: model checking and theorem proving. Model checking relies on automatic exhaustive exploration of the reachable state space of the system. The lock-free algorithms presented in this thesis are too data-intensive and complicated for model checking. Theorem proving can avoid the so-called

state explosion by a compact (or logical) representation of states and state transformations. Therefore, we have chosen the interactive theorem prover PVS for mechanical support. Except informal proofs for a few liveness properties, all the algorithms and proofs presented in this thesis have been formalized and checked with PVS.

It is very difficult to fairly compare the performance of different concurrent implementations of one algorithm, since 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. Even a good comparison might be disputable, and become outdated by the introduction of other architectures. Therefore, we cannot offer full empirical support for the algorithms presented.

Chapter 2 presents an efficient lock-free algorithm for hash tables with open addressing. The algorithm is dynamic in the sense that it allows the hash table to grow and shrink as needed. Experiments indicate that the algorithm scales up linearly with the number of processes. It seems to require on average only constant time for insertion, deletion or accessing of elements. An apparent weakness of our algorithm is the worst-case space complexity proportional to the product of the number of processes and the size of the hash table. However, when all processes make ordinary progress and the hash table is not too small, the actual memory requirement is proportional to the size of the table.

Though PVS provided great help for managing and reusing the proofs, we have to admit that the verification for the algorithm was very complicated due to the complexity of the algorithm. The whole correctness proof of the algorithm contains around 200 invariants. The total verification effort can roughly be estimated to consist of two man years.

Chapter 3 formalizes Herlihy's general methodology for transferring a sequential implementation of any data structure into a lock-free synchronization, and presents a lock-free pattern as a reduction theorem. The reduction theorem enables us to reason about a lock-free program to be designed on a higher level than the synchronization primitives *LL/SC*. It is based on refinement mappings as described by Lamport. Application of this theorem simplifies the verification effort for lock-free algorithms since fewer invariants are required and some invariants are easier to discover and formulate without considering the internal structure of the final implementation. Moreover, two enhanced alternative algorithms are presented that avoid unnecessary copying for large objects in cases where only small part of the objects are modified.

Many machines provide either *CAS* or *LL/SC*, but not both. Chapter 4 presents a

similar lock-free pattern based on the weaker atomic primitive *CAS* without causing the so-called *ABA* problem or problems with wrap around. It is a variation of Herlihy’s general methodology for lock-free transformation.

Chapter 5 presents a lock-free parallel algorithm for mark&sweep garbage collection (GC) in a realistic model using synchronization primitives *LL/SC* or *CAS*. A number of mutators and collectors can simultaneously operate on the data structure. In particular no strict alternation between usage and cleaning up is necessary contrary to what is common in most other garbage collection algorithms. To simplify the proof, we first extend the specification to a high-level implementation, then verify the correctness of the high-level implementation, and finally apply the reduction theorem developed in chapter 3 to implement the higher-level atomic steps by means of the low-level primitives.

The algorithm employs a procedure *Mark\_stack*, which is mainly a form of graph search, and was initially designed as a recursive procedure. It is a surprise that the elimination of the recursion in favor of an explicit stack makes the proof possible. It would be much more difficult (if possible at all) to prove the correctness of the recursive procedure where we have to rely on the fixed point semantics of recursive procedures or some other denotational semantics.

Apart from safety properties, we have also considered the important problem of verifying liveness properties using the strong fairness assumption. Liveness properties are often expressed using the “*leads-to*” relation. They are widely thought to be harder to verify than safety properties. We found that the “*steps-to*” ( $\triangleright$ ) relation and the “*unless*” ( $\mathcal{U}$ ) relation are quite useful to prove the “*leads-to*” ( $\circ\rightarrow$ ) relation, since these two relations only involve a single step, and they can be checked directly by PVS with the help of invariants.

A main observation is that the PVS proof is surprisingly complex compared to the size of the algorithm proved. In [26], Havelund and Shankar admit that their reduction did not make the proof simpler because the major effort has gone to show the refinement relation. We have the impression that, in their case, the gap between the abstraction and the implementation is too big. In our case of the reduction theorem, there are only six invariants and it is not a burden to show the refinement relation between the abstraction and the implementation in the lock-free pattern. Using the reduction theorem, we only postulate 72 invariants in the whole correctness proof of the high-level implementation of lock-free GC since substantial pieces of the concrete program can be dealt with as atomic statements on the higher level. The total verification effort can roughly be estimated to



consist of half a man year. This is significantly less than what we afforded for the correctness of the lock-free hash tables.

# Samenvatting

Op moderne computers draaien veel processen gelijktijdig, b.v. de window manager, de mailer, de tekstverwerker, de klok enz. Al deze processen hebben toegang tot het gedeelde geheugen nodig, maar als zij zonder beperkingen dit gedeelde geheugen lezen en wijzigen kunnen de gegevens inconsistent worden. De klassieke manier om dit te vermijden is het afdwingen van synchronisatie door middel van *locks*. Dit kan echter tot een blokkade leiden.

Wegens blokkades kunnen klassieke synchronisatieparadigma's gebaseerd op locks vele problemen ondervinden zoals *convoying*, prioriteits-inversie en *deadlock*. In de afgelopen twee decennia hebben een aantal onderzoekers technieken voorgesteld om *lock*-vrije implementaties te ontwerpen. Deze technieken staan gezamenlijke bewerking van gedeelde datastructuren toe zonder gebruik te maken van kritieke secties die door locks beschermd worden. Essentieel voor dergelijke implementaties is het gebruik van geavanceerde machine-instructies zoals *LL/SC* of *CAS*. Lock-vrije implementaties garanderen dat binnen een eindig aantal stappen altijd één of ander proces dat een handeling op het object probeert uit te voeren zijn taak zal voltooien. Vele problemen die ontstaan door falen en prioriteits-inversie kunnen zij vermijden. Lock-vrije synchronisatie is zeer belangrijk, met name in *real-time* systemen waar het zeer onwenselijk is een geblokkeerd proces te stoppen als deze een belangrijke of real-time taak uitvoert.

De correctheid van een ontwerp in het vroegst mogelijke stadium waarborgen is een belangrijke uitdaging voor elke verantwoorde methode voor systeemontwikkeling. Lock-vrije algoritmen zijn over het algemeen zeer complex en moeilijk om correct te ontwerpen. De enige techniek die wij kennen is het in een formele taal specificeren van het programmeringsmodel van het gedrag van het systeem en het wiskundig verifiëren dat het systeemontwerp en de implementatie aan *safety* en *liveness*-eigenschappen voldoen. In het algemeen er zijn twee verificatie methodes voor systeemontwerp: model checking en stellingbewijzen. Model checking is gebaseerd op automatische en uitputtende verkenning van de bereikbare

toestandsruimte van het systeem. De in dit proefschrift gepresenteerde lock-vrije algoritmen zijn te ingewikkeld en te omvangrijk qua gegevens voor model checking. Stellingbewijzen kan deze zogenaamde toestandsruimte-explosie vermijden door een compacte (of logische) beschrijving van toestanden en toestandsovergangen te gebruiken. Wij hebben daarom gekozen voor de interactieve stellingbewijzer PVS als mechanisch hulpmiddel. Met uitzondering van enkele informele bewijzen voor een paar liveness-eigenschappen, zijn alle algoritmen en bewijzen uit dit proefschrift geformaliseerd en gecontroleerd met PVS.

Het is zeer moeilijk om een eerlijke prestatievergelijking te maken van verschillende, concurrente implementaties van één algoritme, aangezien de prestaties van parallelle verwerking enorm beïnvloed wordt door de machinearchitectuur, de relatieve omvang van de datastructuren in vergelijking met omvang van de caches, en zelfs de verdeling van de processen over de processoren. Elke redelijke vergelijking zal betwistbaar zijn en verouderd raken door de introductie van andere architecturen. We kunnen daarom geen volledig empirische onderbouwing geven voor de gepresenteerde algoritmen.

Hoofdstuk 2 geeft een efficiënt lock-vrij algoritme voor hash tables met open adressering. Het algoritme is dynamisch in de zin dat het, indien nodig, groeien en inkrimpen van de hash table toestaat. Experimenten wijzen er op dat het algoritme lineair schaalt met het aantal processen. Toevoegen, verwijderen of opvragen van elementen blijkt gemiddeld in constante tijd te kunnen. Een schijnbare tekortkoming van ons algoritme is de ruimte-complexiteit welke in het slechtste geval evenredig is met het aantal processen maal de omvang van de hash table. Echter, als alle processen gewoon voortgang boeken en de hash table niet te klein is, is de daadwerkelijke geheugenvereiste evenredig aan de omvang van de hash table.

Alhoewel PVS aanzienlijke hulp biedt voor het beheren en opnieuw gebruiken van bewijzen, moeten wij toegeven dat het controleren van het algoritme, wegens de complexiteit van het algoritme, zeer ingewikkeld was. Het gehele correctheidsbewijs van het algoritme omvat zo'n 200 invarianten. De totale inspanning voor de verificatie kan ruwweg geschat worden op twee manjaar.

Hoofdstuk 3 formaliseert de algemene methodologie van Herlihy voor het transformeren van een sequentiële implementatie van een datastructuur naar lock-vrije synchronisatie, en stelt een lock-vrij patroon als een reductiestelling voor. De reductiestelling stelt ons in staat te redeneren over een lock-vrij programma dat ontworpen wordt op een hoger niveau dan de synchronisatieprimitieven *LL/SC*. De stelling is gebaseerd op *refinement mappings* zoals beschreven door Lamport. Toepassing van deze stelling vereenvoudigt de verificatie voor

lock-vrije algoritmen aangezien er minder invarianten vereist zijn en sommige invarianten gemakkelijker te ontdekken en te formuleren zijn zonder de interne structuur van de definitieve implementatie te beschouwen. Voorts worden twee verbeterde, alternatieve algoritmen voorgesteld waar het onnodige kopiëren van een groot object vermeden wordt in gevallen waarbij slechts een klein deel van het object gewijzigd wordt.

Veel machines bieden CAS danwel LL/SC aan, maar niet allebei. Hoofdstuk 4 geeft een zelfde lock-vrij patroon dat gebaseerd is op de zwakkere atomaire primitieve CAS, zonder dat het zogenaamde ABA probleem of problemen met *wrap around* optreden. Het is een variatie op de algemene methodologie van Herlihy voor lock-vrije transformatie.

Hoofdstuk 5 presenteert een lock-vrij parallel algoritme voor mark&sweep *garbage collection* (GC) in een realistisch model met gebruikmaking van de synchronisatieprimitieven LL/SC of CAS. Een aantal mutators en collectors kunnen gelijktijdig de datastructuur bewerken. In het bijzonder is een strikte afwisseling tussen gebruik en het opruimen niet noodzakelijk, wat bij de meeste andere algoritmen voor garbage collection wel gebruikelijk is. Om het bewijs te vereenvoudigen breiden wij de specificatie uit tot een implementatie op hoog niveau, verifiëren dan de correctheid van de implementatie op hoog niveau, en passen als laatste de reductiestelling uit hoofdstuk 3 toe om atomaire stappen op het hoge niveau te implementeren met behulp van de primitieven van het lage niveau.

Het algoritme gebruikt een procedure *Mark\_stack* welke in feite een vorm van graph search is en aanvankelijk ontworpen is als een recursieve procedure. Het is verrassend dat het bewijs mogelijk wordt door de recursie weg te werken ten gunste van een expliciete stapel. Het zou veel moeilijker zijn (als het al mogelijk was) om de correctheid van de recursieve procedure te bewijzen, waarvoor we ons zouden moeten baseren op dekpuntssemantiek van recursieve procedures of een andere denotationele semantiek.

Naast safety eigenschappen, hebben wij ook onderzoek gedaan naar het belangrijke probleem van verificatie van liveness eigenschappen onder aanname van strong fairness. Liveness eigenschappen worden vaak in termen van een “*leads-to*” relatie uitgedrukt. Er wordt algemeen verondersteld dat deze moeilijker zijn te verifiëren dan safety eigenschappen. Wij bemerkten dat de “*steps-to*” ( $\triangleright$ ) relatie en de “*unless*” ( $\mathcal{U}$ ) relatie erg nuttig zijn om de “*leads-to*” ( $\mathcal{O} \rightarrow$ ) relatie te bewijzen, aangezien deze twee relaties slechts één enkele stap omvatten, en direct door PVS met behulp van invarianten kunnen worden gecontroleerd.

Een van de voornaamste observaties is dat het bewijs met PVS opzienbarend ingewikkeld is in vergelijking met de omvang van het bewezen algoritme. Havelund en Shankar, [26],

geven toe dat hun reductie het bewijs niet vereenvoudigde omdat de meeste inspanning ging zitten in het verifiëren van de refinement relatie. Wij hebben de indruk dat in hun geval het gat tussen abstractie en implementatie te groot is. In ons geval van de reductiestelling zijn er slechts zes invarianten en kost het niet veel moeite om de refinement relatie tussen abstractie en implementatie aan te tonen in het lock-vrije patroon. Bij gebruik van de reductiestelling postuleren we in totaal slechts 72 invarianten in het correctheidsbewijs van de implementatie op een hoog niveau van de lock-vrije GC, aangezien wezenlijke delen van het concrete programma als atomaire statements op het hoge niveau kunnen worden behandeld. De totale verificatie duur kan ruwweg geschat worden op een half manjaar. Dit is beduidend minder dan we ons voor de correctheid van de lock-vrije hashtables nodig hadden.

## Titles in the IPA Dissertation Series

**J.O. Blanco.** *The State Operator in Process Algebra.* Faculty of Mathematics and Computing Science, TUE. 1996-01

**A.M. Geerling.** *Transformational Development of Data-Parallel Algorithms.* Faculty of Mathematics and Computer Science, KUN. 1996-02

**P.M. Achten.** *Interactive Functional Programs: Models, Methods, and Implementation.* Faculty of Mathematics and Computer Science, KUN. 1996-03

**M.G.A. Verhoeven.** *Parallel Local Search.* Faculty of Mathematics and Computing Science, TUE. 1996-04

**M.H.G.K. Kessler.** *The Implementation of Functional Languages on Parallel Machines with Distrib. Memory.* Faculty of Mathematics and Computer Science, KUN. 1996-05

**D. Alstein.** *Distributed Algorithms for Hard Real-Time Systems.* Faculty of Mathematics and Computing Science, TUE. 1996-06

**J.H. Hoepman.** *Communication, Synchronization, and Fault-Tolerance.* Faculty of Mathematics and Computer Science, UvA. 1996-07

**H. Doornbos.** *Reductivity Arguments and Program Construction.* Faculty of Mathematics and Computing Science, TUE. 1996-08

**D. Turi.** *Functorial Operational Semantics and its Denotational Dual.* Faculty of Mathematics and Computer Science, VUA. 1996-09

**A.M.G. Peeters.** *Single-Rail Handshake Circuits.* Faculty of Mathematics and Computing Science, TUE. 1996-10

**N.W.A. Arends.** *A Systems Engineering Specification Formalism.* Faculty of Mechanical Engineering, TUE. 1996-11

**P. Severi de Santiago.** *Normalisation in Lambda Calculus and its Relation to Type Inference.* Faculty of Mathematics and Computing Science, TUE. 1996-12

**D.R. Dams.** *Abstract Interpretation and Partition Refinement for Model Checking.* Faculty of Mathematics and Computing Science, TUE. 1996-13

**M.M. Bonsangue.** *Topological Dualities in Semantics.* Faculty of Mathematics and Computer Science, VUA. 1996-14

**B.L.E. de Fluiter.** *Algorithms for Graphs of Small Treewidth.* Faculty of Mathematics and Computer Science, UU. 1997-01

**W.T.M. Kars.** *Process-algebraic Transformations in Context.* Faculty of Computer Science, UT. 1997-02

**P.F. Hoogendijk.** *A Generic Theory of Data Types.* Faculty of Mathematics and Computing Science, TUE. 1997-03

**T.D.L. Laan.** *The Evolution of Type Theory in Logic and Mathematics.* Faculty of Mathematics and Computing Science, TUE. 1997-04

**C.J. Bloo.** *Preservation of Termination for Explicit Substitution.* Faculty of Mathematics and Computing Science, TUE. 1997-05

**J.J. Vereijken.** *Discrete-Time Process Algebra.* Faculty of Mathematics and Computing Science, TUE. 1997-06

**F.A.M. van den Beuken.** *A Functional Approach to Syntax and Typing.* Faculty of Mathematics and Informatics, KUN. 1997-07

**A.W. Heerink.** *Ins and Outs in Refusal Testing.* Faculty of Computer Science, UT. 1998-01

**G. Naumoski and W. Alberts.** *A Discrete-Event Simulator for Systems Engineering.* Faculty of Mechanical Engineering, TUE. 1998-02

- J. Verriet.** *Scheduling with Communication for Multiprocessor Computation.* Faculty of Mathematics and Computer Science, UU. 1998-03
- J.S.H. van Gageldonk.** *An Asynchronous Low-Power 80C51 Microcontroller.* Faculty of Mathematics and Computing Science, TUE. 1998-04
- A.A. Basten.** *In Terms of Nets: System Design with Petri Nets and Process Algebra.* Faculty of Mathematics and Computing Science, TUE. 1998-05
- E. Voermans.** *Inductive Datatypes with Laws and Subtyping – A Relational Model.* Faculty of Mathematics and Computing Science, TUE. 1999-01
- H. ter Doest.** *Towards Probabilistic Unification-based Parsing.* Faculty of Computer Science, UT. 1999-02
- J.P.L. Segers.** *Algorithms for the Simulation of Surface Processes.* Faculty of Mathematics and Computing Science, TUE. 1999-03
- C.H.M. van Kemenade.** *Recombinative Evolutionary Search.* Faculty of Mathematics and Natural Sciences, UL. 1999-04
- E.I. Barakova.** *Learning Reliability: a Study on Indecisiveness in Sample Selection.* Faculty of Mathematics and Natural Sciences, RUG. 1999-05
- M.P. Bodlaender.** *Scheduler Optimization in Real-Time Distributed Databases.* Faculty of Mathematics and Computing Science, TUE. 1999-06
- M.A. Reniers.** *Message Sequence Chart: Syntax and Semantics.* Faculty of Mathematics and Computing Science, TUE. 1999-07
- J.P. Warners.** *Nonlinear approaches to satisfiability problems.* Faculty of Mathematics and Computing Science, TUE. 1999-08
- J.M.T. Romijn.** *Analysing Industrial Protocols with Formal Methods.* Faculty of Computer Science, UT. 1999-09
- P.R. D’Argenio.** *Algebras and Automata for Timed and Stochastic Systems.* Faculty of Computer Science, UT. 1999-10
- G. Fábíán.** *A Language and Simulator for Hybrid Systems.* Faculty of Mechanical Engineering, TUE. 1999-11
- J. Zwanenburg.** *Object-Oriented Concepts and Proof Rules.* Faculty of Mathematics and Computing Science, TUE. 1999-12
- R.S. Venema.** *Aspects of an Integrated Neural Prediction System.* Faculty of Mathematics and Natural Sciences, RUG. 1999-13
- J. Saraiva.** *A Purely Functional Implementation of Attribute Grammars.* Faculty of Mathematics and Computer Science, UU. 1999-14
- R. Schiefer.** *Viper, A Visualisation Tool for Parallel Program Construction.* Faculty of Mathematics and Computing Science, TUE. 1999-15
- K.M.M. de Leeuw.** *Cryptology and Statecraft in the Dutch Republic.* Faculty of Mathematics and Computer Science, UvA. 2000-01
- T.E.J. Vos.** *UNITY in Diversity. A stratified approach to the verification of distributed algorithms.* Faculty of Mathematics and Computer Science, UU. 2000-02
- W. Mallon.** *Theories and Tools for the Design of Delay-Insensitive Communicating Processes.* Faculty of Mathematics and Natural Sciences, RUG. 2000-03
- W.O.D. Griffioen.** *Studies in Computer Aided Verification of Protocols.* Faculty of Science, KUN. 2000-04
- P.H.F.M. Verhoeven.** *The Design of the MathSpad Editor.* Faculty of Mathematics and Computing Science, TUE. 2000-05

- J. Fey.** *Design of a Fruit Juice Blending and Packaging Plant.* Faculty of Mechanical Engineering, TUE. 2000-06
- M. Franssen.** *Cocktail: A Tool for Deriving Correct Programs.* Faculty of Mathematics and Computing Science, TUE. 2000-07
- P.A. Olivier.** *A Framework for Debugging Heterogeneous Applications.* Faculty of Natural Sciences, Mathematics and Computer Science, UvA. 2000-08
- E. Saaman.** *Another Formal Specification Language.* Faculty of Mathematics and Natural Sciences, RUG. 2000-10
- M. Jelasity.** *The Shape of Evolutionary Search Discovering and Representing Search Space Structure.* Faculty of Mathematics and Natural Sciences, UL. 2001-01
- R. Ahn.** *Agents, Objects and Events a computational approach to knowledge, observation and communication.* Faculty of Mathematics and Computing Science, TU/e. 2001-02
- M. Huisman.** *Reasoning about Java programs in higher order logic using PVS and Isabelle.* Faculty of Science, KUN. 2001-03
- I.M.M.J. Reymen.** *Improving Design Processes through Structured Reflection.* Faculty of Mathematics and Computing Science, TU/e. 2001-04
- S.C.C. Blom.** *Term Graph Rewriting: syntax and semantics.* Faculty of Sciences, Division of Mathematics and Computer Science, VUA. 2001-05
- R. van Liere.** *Studies in Interactive Visualization.* Faculty of Natural Sciences, Mathematics and Computer Science, UvA. 2001-06
- A.G. Engels.** *Languages for Analysis and Testing of Event Sequences.* Faculty of Mathematics and Computing Science, TU/e. 2001-07
- J. Hage.** *Structural Aspects of Switching Classes.* Faculty of Mathematics and Natural Sciences, UL. 2001-08
- M.H. Lamers.** *Neural Networks for Analysis of Data in Environmental Epidemiology: A Case-study into Acute Effects of Air Pollution Episodes.* Faculty of Mathematics and Natural Sciences, UL. 2001-09
- T.C. Ruys.** *Towards Effective Model Checking.* Faculty of Computer Science, UT. 2001-10
- D. Chkhaev.** *Mechanical verification of concurrency control and recovery protocols.* Faculty of Mathematics and Computing Science, TU/e. 2001-11
- M.D. Oostdijk.** *Generation and presentation of formal mathematical documents.* Faculty of Mathematics and Computing Science, TU/e. 2001-12
- A.T. Hofkamp.** *Reactive machine control: A simulation approach using  $\chi$ .* Faculty of Mechanical Engineering, TU/e. 2001-13
- D. Bošnački.** *Enhancing state space reduction techniques for model checking.* Faculty of Mathematics and Computing Science, TU/e. 2001-14
- M.C. van Wezel.** *Neural Networks for Intelligent Data Analysis: theoretical and experimental aspects.* Faculty of Mathematics and Natural Sciences, UL. 2002-01
- V. Bos and J.J.T. Kleijn.** *Formal Specification and Analysis of Industrial Systems.* Faculty of Mathematics and Computer Science and Faculty of Mechanical Engineering, TU/e. 2002-02
- T. Kuipers.** *Techniques for Understanding Legacy Software Systems.* Faculty of Natural Sciences, Mathematics and Computer Science, UvA. 2002-03
- S.P. Luttik.** *Choice Quantification in Process Algebra.* Faculty of Natural Sciences, Mathematics, and Computer Science, UvA. 2002-04



- R.J. Willemsen.** *School Timetable Construction: Algorithms and Complexity.* Faculty of Mathematics and Computer Science, TU/e. 2002-05
- M.I.A. Stoelinga.** *Alea Jacta Est: Verification of Probabilistic, Real-time and Parametric Systems.* Faculty of Science, Mathematics and Computer Science, KUN. 2002-06
- N. van Vugt.** *Models of Molecular Computing.* Faculty of Mathematics and Natural Sciences, UL. 2002-07
- A. Fehnker.** *Citius, Vilius, Melius: Guiding and Cost-Optimality in Model Checking of Timed and Hybrid Systems.* Faculty of Science, Mathematics and Computer Science, KUN. 2002-08
- R. van Stee.** *On-line Scheduling and Bin Packing.* Faculty of Mathematics and Natural Sciences, UL. 2002-09
- D. Tauritz.** *Adaptive Information Filtering: Concepts and Algorithms.* Faculty of Mathematics and Natural Sciences, UL. 2002-10
- M.B. van der Zwaag.** *Models and Logics for Process Algebra.* Faculty of Natural Sciences, Mathematics, and Computer Science, UvA. 2002-11
- J.I. den Hartog.** *Probabilistic Extensions of Semantical Models.* Faculty of Sciences, Division of Mathematics and Computer Science, VUA. 2002-12
- L. Moonen.** *Exploring Software Systems.* Faculty of Natural Sciences, Mathematics, and Computer Science, UvA. 2002-13
- J.I. van Hemert.** *Applying Evolutionary Computation to Constraint Satisfaction and Data Mining.* Faculty of Mathematics and Natural Sciences, UL. 2002-14
- S. Andova.** *Probabilistic Process Algebra.* Faculty of Mathematics and Computer Science, TU/e. 2002-15
- Y.S. Usenko.** *Linearization in  $\mu CRL$ .* Faculty of Mathematics and Computer Science, TU/e. 2002-16
- J.J.D. Aerts.** *Random Redundant Storage for Video on Demand.* Faculty of Mathematics and Computer Science, TU/e. 2003-01
- M. de Jonge.** *To Reuse or To Be Reused: Techniques for component composition and construction.* Faculty of Natural Sciences, Mathematics, and Computer Science, UvA. 2003-02
- J.M.W. Visser.** *Generic Traversal over Typed Source Code Representations.* Faculty of Natural Sciences, Mathematics, and Computer Science, UvA. 2003-03
- S.M. Bohte.** *Spiking Neural Networks.* Faculty of Mathematics and Natural Sciences, UL. 2003-04
- T.A.C. Willemse.** *Semantics and Verification in Process Algebras with Data and Timing.* Faculty of Mathematics and Computer Science, TU/e. 2003-05
- S.V. Nedeia.** *Analysis and Simulations of Catalytic Reactions.* Faculty of Mathematics and Computer Science, TU/e. 2003-06
- M.E.M. Lijding.** *Real-time Scheduling of Tertiary Storage.* Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2003-07
- H.P. Benz.** *Casual Multimedia Process Annotation – CoMPAs.* Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2003-08
- D. Distefano.** *On Modelchecking the Dynamics of Object-based Software: a Foundational Approach.* Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2003-09
- M.H. ter Beek.** *Team Automata – A Formal Approach to the Modeling of Collaboration Between System Components.* Faculty of Mathematics and Natural Sciences, UL. 2003-10

- D.J.P. Leijen.** *The  $\lambda$  Abroad – A Functional Approach to Software Components.* Faculty of Mathematics and Computer Science, UU. 2003-11
- W.P.A.J. Michiels.** *Performance Ratios for the Differencing Method.* Faculty of Mathematics and Computer Science, TU/e. 2004-01
- G.I. Jojgov.** *Incomplete Proofs and Terms and Their Use in Interactive Theorem Proving.* Faculty of Mathematics and Computer Science, TU/e. 2004-02
- P. Frisco.** *Theory of Molecular Computing – Splicing and Membrane systems.* Faculty of Mathematics and Natural Sciences, UL. 2004-03
- S. Maneth.** *Models of Tree Translation.* Faculty of Mathematics and Natural Sciences, UL. 2004-04
- Y. Qian.** *Data Synchronization and Browsing for Home Environments.* Faculty of Mathematics and Computer Science and Faculty of Industrial Design, TU/e. 2004-05
- F. Bartels.** *On Generalised Coinduction and Probabilistic Specification Formats.* Faculty of Sciences, Division of Mathematics and Computer Science, VUA. 2004-06
- L. Cruz-Filipe.** *Constructive Real Analysis: a Type-Theoretical Formalization and Applications.* Faculty of Science, Mathematics and Computer Science, KUN. 2004-07
- E.H. Gerding.** *Autonomous Agents in Bargaining Games: An Evolutionary Investigation of Fundamentals, Strategies, and Business Applications.* Faculty of Technology Management, TU/e. 2004-08
- N. Goga.** *Control and Selection Techniques for the Automated Testing of Reactive Systems.* Faculty of Mathematics and Computer Science, TU/e. 2004-09
- M. Niqui.** *Formalising Exact Arithmetic: Representations, Algorithms and Proofs.* Faculty of Science, Mathematics and Computer Science, RU. 2004-10
- A. Löh.** *Exploring Generic Haskell.* Faculty of Mathematics and Computer Science, UU. 2004-11
- I.C.M. Flinsenberg.** *Route Planning Algorithms for Car Navigation.* Faculty of Mathematics and Computer Science, TU/e. 2004-12
- R.J. Bril.** *Real-time Scheduling for Media Processing Using Conditionally Guaranteed Budgets.* Faculty of Mathematics and Computer Science, TU/e. 2004-13
- J. Pang.** *Formal Verification of Distributed Systems.* Faculty of Sciences, Division of Mathematics and Computer Science, VUA. 2004-14
- F. Alkemade.** *Evolutionary Agent-Based Economics.* Faculty of Technology Management, TU/e. 2004-15
- E.O. Dijk.** *Indoor Ultrasonic Position Estimation Using a Single Base Station.* Faculty of Mathematics and Computer Science, TU/e. 2004-16
- S.M. Orzan.** *On Distributed Verification and Verified Distribution.* Faculty of Sciences, Division of Mathematics and Computer Science, VUA. 2004-17
- M.M. Schrage.** *Proxima - A Presentation-oriented Editor for Structured Documents.* Faculty of Mathematics and Computer Science, UU. 2004-18
- E. Eskenazi and A. Fyukov.** *Quantitative Prediction of Quality Attributes for Component-Based Software Architectures.* Faculty of Mathematics and Computer Science, TU/e. 2004-19
- P.J.L. Cuijpers.** *Hybrid Process Algebra.* Faculty of Mathematics and Computer Science, TU/e. 2004-20
- N.J.M. van den Nieuwelaar.** *Supervisory Machine Control by Predictive-Reactive Scheduling.* Faculty of Mechanical Engineering, TU/e. 2004-21

**E. Abraham.** *An Assertional Proof System for Multithreaded Java -Theory and Tool Support-* . Faculty of Mathematics and Natural Sciences, UL. 2005-01

**R. Ruimerman.** *Modeling and Remodeling in Bone Tissue.* Faculty of Biomedical Engineering, TU/e. 2005-02

**C.N. Chong.** *Experiments in Rights Control - Expression and Enforcement.* Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2005-03

**H. Gao.** *Design and Verification of Lock-free Parallel Algorithms.* Faculty of Mathematics and Computing Sciences, RUG. 2005-04