TECHNISCHE UNIVERSITÄT MÜNCHEN
Institut für Informatik
Lehrstuhl für Netzarchitekturen und Netzdienste

An Execution Trace Verification Method on Linearizability

Kristijan Dragičević

Vollständiger Abdruck der von der Fakultät für Informatik der Technischen Universität München zur Erlangung des akademischen Grades eines

Doktors der Naturwissenschaften (Dr. rer. nat.)

genehmigten Dissertation.

Die Dissertation wurde am 24.01.2011 bei der Technischen Universität München eingereicht und durch die Fakultät für Informatik am 04.03.2011 angenommen.

Kurzfassung:

Multicore Prozessoren sind inzwischen zur Standardarchitektur für reguläre Prozessoren geworden. Folglich sind Programmierer immer mehr dazu angehalten, optimierte pa-rallele Software zu entwickeln. Die Verifikation von parallelem Code ist sehr komplex, was neue praktikable Verifikationsmethoden notwendig macht. Es fehlt immer noch an Tools, welche komplexe parallele Programme verifizieren können. Diese Dissertation stellt eine Methode vor, welche Ausführungen von Programmen testet und zeigt, dass diese Methode bereits wertvolle Resultate für den Anwendungsprogrammierer liefert. Die Verifikationsmethode wird verwendet um zu beweisen, dass ein Programm-Trace linearisierbar ist. Darüberhinaus werden einige nützliche allgemeine Eigenschaften von diversen Algorithmen herausgegriffen, welche die praktische Realisierung dieser Methode optimieren können. Ferner liefert diese Arbeit ein Fallbeispiel anhand von Priority Queues um zu zeigen, wie diese Methode angewendet werden kann. Das Fallbeispiel zeigt, dass praktikable Resultate mithilfe dieser Methode erzeugt werden können.

Abstract:

As multicore processors have become the standard architecture for general-purpose machines, programmers are required to write software optimized for parallelism. Verifying parallel code is very complex, and new practical methods for verification are therefore very important. Tools that provide verification for complex parallel code are still missing. This dissertation introduces a method for testing the execution of code and shows that this method already provides valuable results for the application programmer. The verification method is used to prove whether a program trace is linearizable. In addition, some useful properties of algorithms are highlighted by which a practical use of the method can be optimized. Furthermore, this work provides a case study on priority queues of how this method can be applied. Our case study shows that practicable results can be obtained with this method.

Acknowledgment

First of all I want to thank my Ph.D. supervisor Prof. Dr. Georg Carle for his steady support and encouragement during all the time of this doctorate. I also want to express my great gratitude to my supervisor at IBM Research, Dr. Daniel Bauer, who helped me at any time and who was always available for fruitful discussions. Many of the ideas in this thesis have been developed by the inspirations that were provided by him. Further, I want to thank Prof. Dr. Marcel Waldvogel for being the second supervisor and for his valuable comments and suggestions. I thank my managers at IBM Research, Dr. Andreas Kind and Dr. Paolo Scotton, for their efforts in all organizational aspects. I also thank my team colleagues Dr. Luis Garcés-Erice and Dr. Sean Rooney, for valuable discussions which have given me a wider horizon of the topic. Last, I want to thank my colleagues Florian Auernhammer, Phillip Frey, Marc Ph. Stoecklin, and Jon Rohrer for forming an enjoyable environment over the last three years.

Contents

List of Figures

List of Tables

1. Introduction

This dissertation addresses the problem of finding a practical solution for checking today's parallel implementations of complex data structures and systems against the linearizability criterion. Linearizability is a significant property of parallel output because it matches precisely the idea of how a transfer from a sequential program to a parallel one should be done intuitively. Solving this problem in a practical way increases the quality and efficiency of code development.

The proposed solution provides a formalism of how parallel implementations can be specified via invariants and checks the program against the invariants afterwards. The verification is performed by a generic verifier that conducts the verification process on actual executions of an input/output automaton.

This chapter describes the general background and motivation for this work. After starting with the motivation, the challenge and significance of the field of verification is explained. After that, the contribution of this thesis is stated and the work that forms the base of it is described. Finally, the structure of the entire document is outlined.

1.1 Motivation

Since the development of computer systems, the problem of programming errors has always been present. There are different degrees of risk if a program is malfunctioning. In many cases these so-called "bugs" are merely irritating, but sometimes they can be dangerous depending on the application. Thus, producing correct programs is of high importance.

For decades, researchers have met the challenge of verification of sequential programs. Home computers and business machines were running on single-core processors. Thus, most applications were naturally sequential programs because there was no gain in making programs executed with parallel threads. Today, parallel systems are realized even in home computers and thus it is urgent to have parallel code exploiting the potential of todays multi-core hardware.

As a consequence, many of the sequential algorithms will have to be adapted to the requirements of multi core architectures in order to scale well on such systems. Otherwise, software will not take any advantage of the newly engineered parallel processors. The design of parallel software is especially challenging for concurrently accessed and modified data structures because mechanisms that avoid inconsistencies are required. These mechanisms go along with additional overhead and complexity. Consequently, it is also urgent to have verification tools for concurrent programs.

No matter which approach is chosen for tackling the issue of implementing a parallel program, there still remains the question whether the implementation or algorithm is actually correct. Not only designing, but also writing correct and efficient parallel code is hard in most cases. Even if the design of a parallel algorithm is sound on paper, it is not assured whether the implemented code itself is correct due to the complexity of such algorithms. Hence, it is very important to have a tool or method that is capable of delivering a perception for its correctness. This dissertation proposes a technique, that is already applicable and therefore is of practical use to today's programmers.

1.2 A Short History of Parallel Computing

Until about the year 2000, the operating clock frequency of CPUs increased comparably to Moore's law for transistors. Generally speaking, Moore's law states that the number of transistors on an integrated circuit doubles every two years. This law is not really a law but rather a rule of thumb that is used in business practice and is also applied to other performance units like clock frequency from time to time. Figure 1.1 shows the evolution of clock

Table 1.1: Issue dates of Intel processors

Intel arch.	4004	8008	8080	8086	80286	80386	80486	Pentium
Release	1971	1972	1974	1978	1982	1985	1989	1993

Intel arch.	Pent. II	Pent. III	Pent. IV	Itanium 2	Core 2	Core i7
Release	1997	1999	2000	2002	2006	2008

frequencies over the past decades for the Intel processors in Table 1.1 and compares it to a line that depicts a double up for every 30 months.

The clock frequency was the main performance criterion for CPUs. However, physical limits, like thermal issues and architectural concerns, have caused a shift in processor design. Nevertheless, the number of transistors per chip still increases according to Moore's law with a double up for every two years as illustrated in Figure 1.2. Chip designers take advantage of this development by increasing the number of processing cores on a single chip. Furthermore, support for hardware threads allows for additional parallelism and decreases the cost of context switches of software threads. This trend means that writing efficient parallel programs will be the main ambition for advanced future software technologies, and not tuning single-thread performance as it has been until now.

However, parallel programming already exists since the 1950's but the execution of parallel code was realized by distributed systems or symmetric multiprocessors. These systems mainly dealt with data parallelism which means that data is divided up into chunks that are treated by different processors executing the same code. These parallel applications were specialized, scientific programs that did not use shared memory but rather message passing approaches.

Todays's parallel systems deal not only with simple data parallelism but also task parallelism. Task parallelism means that a certain task is divided up into sub-activities which are executed by different threads or processors. Therefore, many possibilities for thread synchronization have been invented. One way of concurrency handling is to use message passing as previously mentioned. The prevalent way is to use locking for synchronization between threads. Coarse-

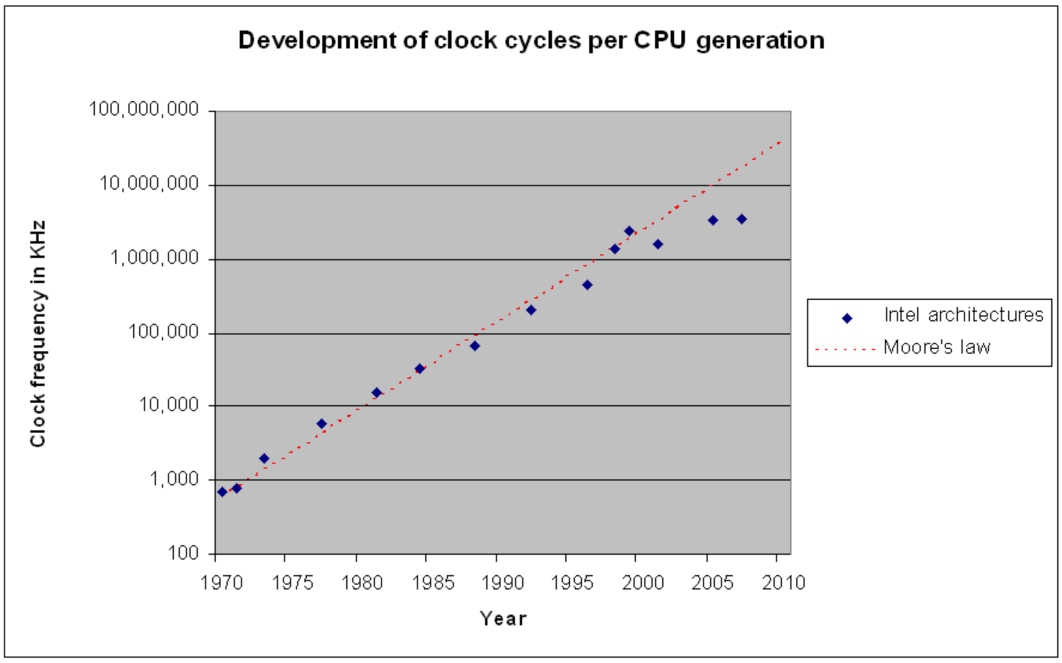


Figure 1.1: Evolution of the clock cycle performance for Intel processor architectures of Table 1.1

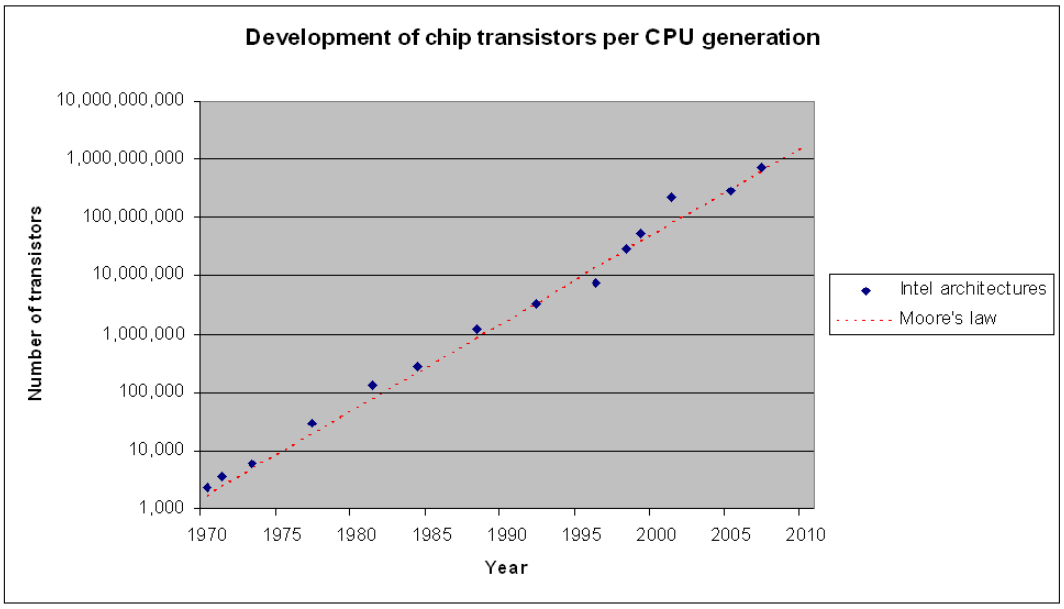


Figure 1.2: Evolution of the transistor number per chip for Intel processor architectures of Table 1.1

grained locking denotes the protection of the data structure as a whole by one single lock. Whereas, fine-grained locking is a method where only parts of the data structure are protected by a set of locks. Locks serialize accesses to data structures, allowing at most one single thread to read or manipulate the protected part. All other threads are blocked if they try to access the locked data structure. It is a well-known fact that solutions using coarse-grained locking lack scalability for increasing numbers of threads as they prevent parallel execution. On the other hand, approaches with fine-grained locking often prove to be complex and error-prone in both design and implementation. Moreover, the scalability of this method is still limited due to the sequential nature of locks in general and their acquisition and release costs. Readers and Writers locks are an improvement concerning the flexibility of locks such that there might be more reading threads. Nevertheless, in principle they still have similar lacks as classical locks.

One attempt for overcoming the disadvantages of lock-based methods is to use lock-free approaches which typically implement concurrency control using atomic read-modify-write instructions such as compare-and-swap (CAS), test-and-set, or fetch-and-add (FAA) that are provided by modern processors. CAS compares the contents of a memory location to a given value and, if successful, modifies the content. Test-and-set returns the current value of a memory location and updates it by the new value. FAA increases the content of some memory location by one. All three operations are executed atomically by the processor. It should be mentioned that some atomic lock-free operations can be used for forming lock-like constructs.

Another attempt for avoiding locks is to use the new programming paradigm transactional memory (TM, [HeMo93]) or software transactional memory (STM, [ShTo95]) as a technique for synchronization. This innovative paradigm gives the programmer a more abstract view of the program and its resources. STM and TM take over concurrency handling so that the programmer does not have to deal with the synchronization details. They use an optimistic approach in which the updates are done before the validation of results is conducted. If the results are invalid due to a conflict with a concurrent thread, they are discarded. Otherwise, the results are committed and thus, made visible to all other threads. Unfortunately, TM and STM have still a long path to go before becoming feasible for programmers to use. Firstly, there are performance problems due to the overhead generated by the generic concurrency handling. Secondly, it is not as simple to use as desired for programmers.

1.3 The Challenge of Verification of Parallel Implementations

Verification of code is a broad area. There are many different aspects to verify and there are different phases during development for design analysis. The earliest phase possible for code analysis is before the first line of code has been actually written. A formal analysis of the design on paper can be accomplished for verifying that the principle of the implementation idea is correct. The next phase is during compile time. A compiler that issues warnings is already a supporter for the programmer concerning correctness. However, there are much more complex verifiers. Static source code analysis is one possibility that has been realized by a lot of proprietary and research tools [FLLN$^+$02, Cous07, Vola06, ABDD$^+$07]. The latest point in time when verification may take place is post-mortem. In this case, the code is simulated or executed and the produced output is analyzed. This is an approach which is commonly chosen for hardware development as in [GoYS04].

Now, the big challenge when verifying parallel programs is that it is not simply possible to check whether the execution of a certain line of code leads or would lead from one state to a specific other state. There is a context to be considered in the sense that there are interferences from other threads of execution. It is not easy to determine at which specific state a system is at a precise point in time. Often it is not even possible to determine a definite state. This leads to a huge state space for parallel programs which makes verification very complex. The following simple example shall exemplify the complexity of concurrent programs compared to a sequential program with increasing number of instructions.

A program is essentially a sequence of instructions. Let us consider an instruction sequence $I = (i_1, i_2, ..., i_n)$ without branches. From an abstract point

Table 1.2: Number of possible instruction sequences per constellation

	2 instructions	3 instructions	4 instructions
1 thread	1	1	1
2 threads	6	90	2520
3 threads	20	1680	369600
4 threads	70	34650	$\approx 6.3 \cdot 10^7$

of view the processor will first execute i_1 then i_2 and at last i_n. Although, a single processor might reorder instructions for performance reasons it still ensures that the outcome of the program is the same as if the instructions had been executed in the precise sequential order. Consequently, there is only one sequence to be verified for correctness namely I.

Now, if we have two concurrent threads T_1 and T_2 executing the same sequence each there are already $\binom{2\cdot n}{n}$ possibilities of different sequences due to overlaps. This results in a factorial growth of possibilities with increasing number of threads which is an extremely disadvantageous effect. In the case $n = 2$ there are already 6 possible sequences. These are

1. $T_1(i_1), T_1(i_2), T_2(i_1), T_2(i_2)$
2. $T_1(i_1), T_2(i_1), T_1(i_1), T_2(i_2)$
3. $T_1(i_1), T_2(i_1), T_2(i_2), T_1(i_2)$
4. $T_2(i_1), T_1(i_1), T_2(i_2), T_1(i_2)$
5. $T_2(i_1), T_1(i_1), T_1(i_2), T_2(i_2)$
6. $T_2(i_1), T_2(i_2), T_1(i_1), T_1(i_2)$

where $T_i(x)$ means thread T_i executes instruction x. The general formula for k threads and n instructions where $k, n > 1$ is

$$\prod_{i=2}^{k} \binom{i \cdot n}{n} \tag{1.1}$$

Table 1.2 lists the number of possible instruction sequences for a given number of threads and a number of atomic instructions executed by all threads. For 4 threads executing 4 atomic instructions each there are about 63 million different sequences possible that have to be covered in a complete analysis of the parallel program. These numbers evidence that analyzing concurrent programs is a hard challenge.

There are different correctness criteria like sequential consistency [Lamp79] or linearizability [HeWi90]. Depending on the correctness criterion performing a verification can vary in complexity. This document is addressing the linearizability correctness criterion which takes timing issues into consideration. It is well-known that this criterion cannot be proved trivially.

1.4 Claims

At the moment the only practical solution for verification of parallel programs is to do a testing or simulation approach. This thesis demonstrates a method and a tool for a systematic execution analysis of test runs of a parallel program for a non-trivial example. This work covers verification of high-level data structures like priority queues, lists and sets.

Currently, the problem of verification is that the more precise a verification is, the less it is practical. Verification approaches range from very complex but not generally applicable ones over very abstract and thus not precise ones to concrete and generally applicable but not waterproof ones.

The introduced method performs a verification of concrete parallel code by managing a set of sets of operations. This set describes all possible states of a program at a certain point of execution time. In this thesis the focus is on the verification of data structure implementations. A data structure is a particular arrangement of data for efficient manipulation and accessibility. The contributions of the dissertation are stated in the following.

1. New state representation

The brute-force verifier for parallel programs of today usually operate on objects. This lowers the flexibility of these kind of applications because there exist numerous data structures that are based on primitive values instead of objects. Therefore, another view on the state of a system is introduced by the *metastate* construct which also allows for operating on primitive values.

2. *Metastate* compression

In state exploration algorithms, one of the biggest problems is the state space explosion for parallel programs. Therefore, the newly introduced *metastate* construct is elaborated for potential compression of its representation such that there is a huge gain for the verification performance in using it.

3. New verification methodology

The core work is a formal description of the applied verification methodology. It proves the validity of the method. It is the first non-brute-force method that can be used for linearizability checks which does not need any backtracking within the verification procedure whereas brute-force approaches go back and forth in the execution stream.

4. Generic verifier framework

The implementation of a generic verifier framework provides interfaces that allow a wide range of use cases to be implemented and execute a verification against it. It also gives a generic base implementation that can be used by verifiers using this method. This generic verifier framework is the first step towards a realization of the introduced methodology.

5. Foundation for systematic state pruning

There is given concrete implementations for two different use cases that have already been verified before which are the list and the set. However, the verification tools implemented here are more generic than the known ones because it can operate merely on keys of elements whereas other approaches needed keys and unique values of elements in collections. Furthermore, this method does not require any backtracking which increases the chance for finding coding errors.

6. High-performance verification of the priority queue use case

Implementations for the priority queue use case have never been verified before. The implemented tool is the first verifier applied for priority queues and shows a high performance compared to a brute-force approach.

7. Fine-grained STM-based priority queue

A case study of a sophisticated software transactional memory-based data structure is given. This case study shows an example for a priority queue which made verification necessary for the development of a parallel implementation especially in the case of STM.

1.5 Outline

Chapter 2 describes the basic approaches for today's parallel programs and motivates the thesis work. There are many implemented algorithms for which a verification is necessary. Here some of them are illustrated and it is shown how and why testing and simulation is useful.

Chapter 3 introduces related work and places this work in the area of correctness reasoning. It also explains why the approach of the thesis work has been chosen.

Chapter 4 elucidates the formalism that forms the core of the thesis work. Furthermore, a proof that shows that the method provides the expected results is provided.

The implementation issues are depicted in chapter 5. Not only the usage of the tool is described, but also optimizations that have been implemented for an efficient processing of the verification.

Chapter 6 contains a set of case studies in which this method has been applied. It also includes statistics about performance and optimization effects.

Finally, chapter 7 concludes and summarizes the dissertation and gives an outlook of what could be done as future research.

2. Concurrent Programming

The term *concurrent programming* denotes the idea of having multiple threads of execution that perform tasks in parallel. Usually a program is executed by a single thread of execution. If a task can be executed by multiple threads in parallel this can be realized by a parallel program. The term concurrent programming refers to the implementation of parallel code.

This chapter provides background information about concurrent programming and its issues. First, it starts with describing the different classes of parallel systems. Then, it introduces the *memory model*. After that, it explains different mechanisms as locks, semaphores, monitors, non-blocking and STM-based synchronization with examples of their realization. During the development phase of these examples the verification method introduced in this dissertation has been used for assisting the implementation of them.

2.1 Classification of Parallel Systems

Flynn [Flyn72] introduced a simple classification of parallel systems summarized in Table 2.1. He distinguishes among systems that use single or multiple instruction sets, and single or multiple data sets.

SISD The single-instruction-single-data class denotes the class of sequential programs. For one instruction set applied on a single data set there is no more than a single processing unit necessary.

SIMD Single-instruction-multiple-data summarizes those programs that consist of threads performing the same instructions on different data sets

Table 2.1: Flynn's taxonomy

	Single Instruction	Multiple Instructions
Single Data	SISD	MISD
Multiple Data	SIMD	MIMD

each. Array processing is a typical example where SIMD systems are used. Problems which can be solved by data parallel architectures are often called *embarrassingly parallel*.

MISD Multiple-instruction-single-data includes parallel programs where its threads perform different instructions on the same data. It is a rather uncommon class in practice.

MIMD In a Multiple-instruction-multiple-data system threads perform different instructions on different data sets. Distributed systems are typical MIMD systems which operate on distributed but also on shared memory. Computer systems with a multi-core processor can also run MIMD programs which they do on shared memory.

A technical report of a group around David Patterson at the University of California, Berkeley [ABCG+06] summarizes the main problems of significance to be tackled by concurrent programs. Some of them are simple to solve because it is easy to split them up into distinct non-overlapping partial problems that can be solved by SIMD systems. Other problems need many accesses on shared memory and thus the different tasks perform a lot of overlapping and interfering operations. In this case, it must be assured, that all accesses have been made based on a consistent view of the memory among all threads. Since there are numerous substantially different software and hardware architectures, they have to specify a memory model for an understanding of what a consistent view of memory is.

2.2 Memory Model

A memory model defines the assumptions that a compiler is allowed to make when compiling code for systems using segmented memory. Memory segmentation is usually implemented for memory protection in such a way that there is program, data and stack segment so that for example only instructions are read from a program segment but no data.

Memory models are necessary for performing compiler optimizations in parallel programs especially when shared variables are involved. A compiler optimization influences the order of read and write operations of shared and unshared variables which can lead to race conditions. A race condition is a situation where the result of an instruction sequence depends on the timing of its instructions. Therefore, a compiler may not optimize concurrent programs without a memory model.

In Java for example, the memory model defines memory barriers that are realized by well-defined synchronization operations or the behavior of a *volatile* variable [GoJS96]. The idea of race conditions is entirely defined over the order of operations with respect to these memory barriers. Reordering instructions in a code block without synchronization barriers is assumed to be safe by the compiler.

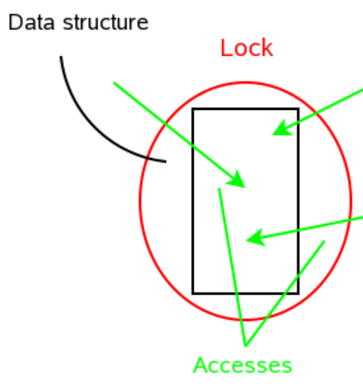
In the following sections a selection of concurrent programming paradigms for synchronization and concurrency control are described. These paradigms are synchronization with locks, semaphores, monitors, lock-free synchronization, and transactional memory.

The inducement for this dissertation has been the work for a survey of concurrent priority queues [DrBa08]. As an exemplification of the different synchronization mechanisms some of the investigated queues will be used to produce verification cases for this dissertation.

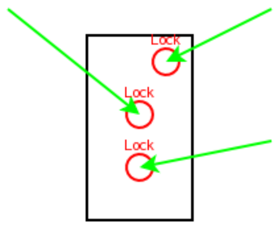
A priority queue is a data structure that defines at least two operations which are *insert* and *remove*. An *insert* operation inserts an element into the queue with a certain priority. A *remove* operation fetches the element of highest priority in the queue and deletes it.

2.3 Mutual Exclusion Locks

The classic approach for concurrency control is using *mutual-exclusion* locks or *mutex* for short to turn non-concurrent algorithms into concurrent ones (Figure 2.3). There are two classes of lock-based synchronization.



(a) Coarse-grained locking



(b) Fine-grained locking

Figure 2.1: General idea of locking approaches

2.3.1 Coarse-Grained Locking

Coarse-grained locking means that a single global lock is used that regulates access to a data structure as illustrated in Figure 2.1(a). By doing this, it is ensured that only one thread at a time is allowed to manipulate the data structure. This method is the least complex solution when trying to implement concurrent programs. However, real parallelism is avoided because the accesses are sequentialized. Consequently, the full potential of multi core architectures is not exploited.

2.3.2 Fine-Grained Locking

Another opportunity is to use fine-grained locking which is schematized in Figure 2.1(b). In this approach, not the entire data structure is protected by a lock, but different critical parts of it. Thus, for performing a complex operation there is a sophisticated pattern necessary that gives a rule in which order the locks have to be acquired. Due to the complexity of the application of fine-grained locking there are a number of known problems with this approach. For example, *convoying* is the undesired effect that occurs when a set of threads of execution try to acquire repeatedly for the same lock. The consequence is that performance is lowered because all threads execute similar code fragments and thus hinder each other frequently from progressing.

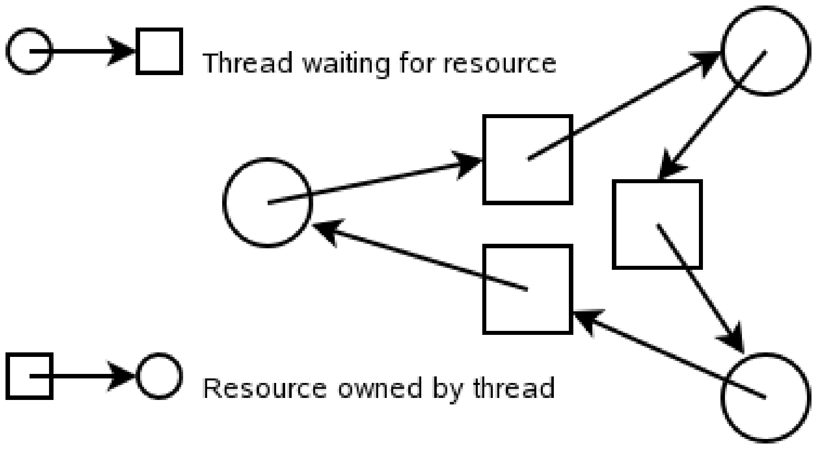


Figure 2.2: The schema of a deadlock

Another well-known problem is the occurrence of *deadlocks* [Zö83]. Threads are in a deadlock when they are waiting for each other to release a resource but never do. In a thread/resource graph, a deadlock can be identified by a cycle of access arrows as in Figure 2.2. In the Figure each thread illustrated as a circle is waiting for a resource represented by a square which is hold by another thread. Hence nobody is making any progress and the program is stuck. Deadlocks are very difficult to debug because it is often hard to identify the constellation that results in a deadlock.

There are different degrees of granularity for fine-grained locking approaches. The transition from coarse-grained locking to fine-grained locking is smooth.

2.4 Semaphores

A semaphore is a counter for a set of available resources unlike a mutex which is essentially a flag of a single resource. Dijkstra developed semaphores as a solution to preventing race conditions [Dijknd, Dijk68]. Nevertheless, they do not prevent resource deadlocks.

The value of a semaphore is the number of free resource units. If there is only one resource being managed, the semaphore reduces to a *binary semaphore* with values 0 or 1 which could be also realized by a mutex.

There are two operations for semaphores. During the so-called P operation the process busy-waits or sleeps until a resource is available. If a resource is available it is assigned to the process. The V operation makes a resource available after the process has finished using it. Both operations must be atomic.

From a logical point of view, the *binary semaphore* is the same as a *mutex*, in principle. However, there maybe platform dependent differences in their realization. As mutexes, semaphores do not avoid deadlocks because of following possible situation:

there are n resources of A and m resources of B. n and m different threads holding one resource of A and B each, respectively. Now, if the n threads are waiting for a resource of B to be released whereas the m threads are waiting for a resource A to be released, there is a deadlock situation.

2.5 Monitors

As mutexes and semaphores, monitors realize synchronization among threads by mutual exclusion. Monitors were invented by Hoare and Hansen [Hoar74, Hans75]. The concept also provides the possibility of threads temporarily giving up exclusive access and waiting for some condition. If the condition is met, the thread regains access and resumes its task. Furthermore, monitors have a mechanism for signaling other threads that a condition has been met. Monitors are a higher level concept so that programmers do not need to handle synchronization primitives explicitly. They are often used in object-oriented programming language like Java.

2.6 Non-Blocking Synchronization

Another class of algorithms is known as non-blocking algorithms. As the name indicates, they refrain from using locks or other blocking instructions and instead are based on atomic lock-free instructions as the basic means of concurrency control. There have been a number of basic strategies applied for non-blocking algorithms like recursive helping or versioning which will be explained later in section 2.8. Despite the fact that there exist simple implementation examples for linked-lists or FIFO-queues, a common problem when avoiding locks is the high complexity of these kind of implementations.

2.6.1 Non-Blocking Properties

The main advantage of non-blocking algorithms in comparison to blocking algorithms is that they have non-blocking properties which guarantee a certain degree of liveness of the implementation in the sense that a stalled process will not make other processes stall. There are three properties of importance when reasoning about non-blocking implementations.

Wait-freeness guarantees that every concurrent operation completes within bounded time. Wait-freeness is of theoretical importance, but most algorithms of practical relevance are not wait-free owing to complexity and efficiency problems.

Lock-freeness states that the system as a whole will always make progress while it is possible that an individual operation is blocked by other concurrent operations. For most practical systems, lock-freeness is a sufficient condition. It ensures overall progress.

Obstruction-freeness is the weakest property [Herl03]. It states that an operation completes in bounded time if it is executed in isolation, for example as when no concurrent operations are executed. Obstruction-free algorithms do not guarantee progress, and livelocks are possible.

The term livelock is usually only used in the context of optimistic algorithms, especially software transactional memory. Optimistic algorithms perform actions with the option that all effects might be discarded afterwards. Having this in mind, a livelock is a situation when a group of threads repeatedly acquire resources from each other and thus provoke abortions of the other threads. The aborted thread will discard its actions and again reacquire the resource from the competing thread. If the action takes a relatively long time, none of the threads is able to complete its actions and consequently they are aborting each other forever.

This is not a desired behavior. However, when the synchronization mechanism is combined with a contention manager that resolves livelocks by for example exponential backoff, obstruction-free algorithms provide a simpler alternative to lock-free or wait-free algorithms as has been shown in [FLMS05, Herl03].

2.6.2 Non-Blocking Primitives

Herlihy showed that atomic primitives, for example compare-and-swap (CAS) or fetch-and-add (FAA), differ in their expressiveness [Herl88]. Some primitives cannot implement a wait-free version of other primitives. This fact implies a classification of atomic primitives.

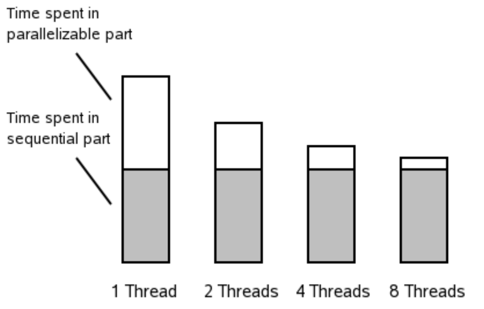
The class with the most expressive atomic primitives are called *universal*. A primitive is universal if it can be used to solve the general consensus problem for n processes [FiLP85]. The consensus problem considers an asynchronous system of potentially unreliable processes. The problem is to agree on a binary value for the reliable processes.

One universal primitive is the CAS instruction. It was originally implemented in IBM's System/370 and is now supported by many modern multi-core processors in hardware. Other processors implement for the same functionality the load-linked/store-conditional (LL/SC) instruction pair. It can be used to realize a CAS implementation [Alph96].

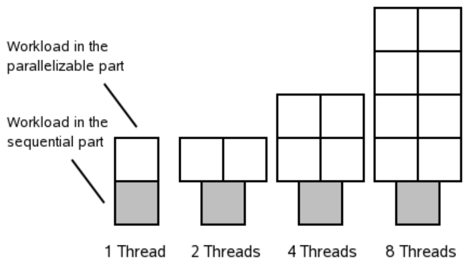
The reason why non-blocking instructions have been invented is that they do not increase the size of sequential parts in a parallel programs when used as blocking constructs like locks, semaphores and monitors naturally do. In the best case, the increase of the performance $\Pi_n(P)$ of a program P would be linear to the increase of the number n of processing cores

$$\Pi_n(P) = \Pi_1(P) \cdot n \tag{2.1}$$

However, in practice if shared memory is involved there is hardly any algorithm that achieves this performance. The possible performance increase of an algorithm on a parallel processor is described by Amdahl's law [Amda67]. It states that the non-parallelizable part of the program limits the maximum performance available from parallelization. Most problems typically consist of


Time spent in parallelizable part
Time spent in sequential part
1 Thread
2 Threads
4 Threads
8 Threads

(a) Amdahl's time efficiency for increasing numbers of threads and constant workload


Workload in the parallelizable part
Workload in the sequential part
1 Thread
2 Threads
4 Threads
8 Threads

(b) Gustafson's workload performance for increasing numbers of threads and constant time

Figure 2.3: Amdahl's and Gustafson's laws

parallelizable parts and sequential parts. If s is the ratio of all sequential parts and p the ratio of all parallelizable parts of a program P then

$$\Pi_n(P) = \Pi_1(P) \cdot \frac{1}{s + \frac{p}{n}}, \tag{2.2}$$

where of course $p + s = 1$. Note, that if $s = 0$ in equation 2.2 it reduces to equation 2.1. Consequently, if 90% of a program is parallelizable, the maximum performance increase is 10× according to Amdahl. Thus, it is desirable to have as few sequential parts as possible and therefore use non-blocking approaches because they avoid sequential parts in a program. However, it has to be noted that Amdahl's law considers problems of a fixed size. Figure 2.3(a) illustrates the performance gain in time according to Amdahl.

Gustafson approached the issue from a different angle [Gust88]. He regarded problems of fixed time with variable workload sizes which results in

$$\Pi_n(P) = \Pi_1(P) \cdot (p \cdot n + s) \tag{2.3}$$

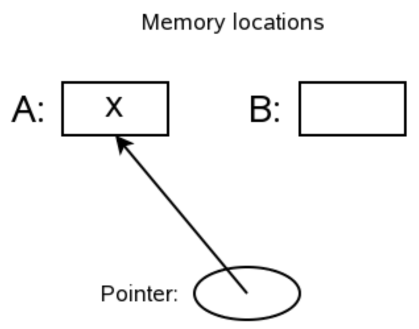
Here, the performance gain is expressed by the additional workload that can be processed in the same time for increasing numbers of processors. An example where Gustafson's law can be applied is a raytracer in 3D image modeling. Each additional processor can be used to refine the picture of the model by increasing the number of pixels or the recursion depth for more precise results. Figure 2.3(b) shows the performance increase in workloads according to Gustafson. Nevertheless, it is worthwhile to keep the sequential parts as low as possible in both cases.

2.6.3 The ABA-Problem

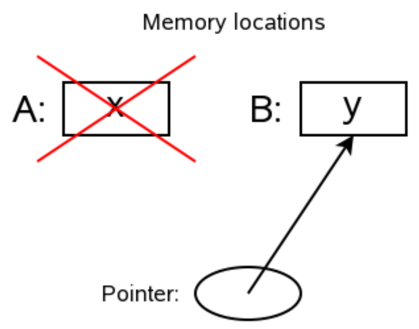
The so-called ABA-problem [Mich04] is one of the practical problems for non-blocking implementations. The ABA-problem refers to the issue that a thread T_1 reads the value A from a certain address and the value is changed twice by another thread T_2 before T_1 reads the value again. The intricate point is that T_2 changes the value first to B and then back to A. If T_1 reads the value after the two modifications of T_2 without having read the values between the write actions, then T_1 is not aware of the fact that the value at the address has already been modified. This sequence of value changes is troublesome when using CAS, because this instruction cannot solve the ABA-problem but has to be taken care of by the program designer.

The following example algorithm with a pointer illustrates the problem. In the algorithm a thread reads a pointer and obtains the value from the pointed memory address. Then, the thread performs computations and creates a new value at another memory location based on the value read before. In the last step, the pointer is changed with a CAS instruction from the old memory address to the newly initialized memory location. If a conflict with another operation executed by another thread occurs, the CAS instruction fails because the pointer has changed in the meanwhile. The program notifies the failure of the CAS instruction and hence, can react properly on it.

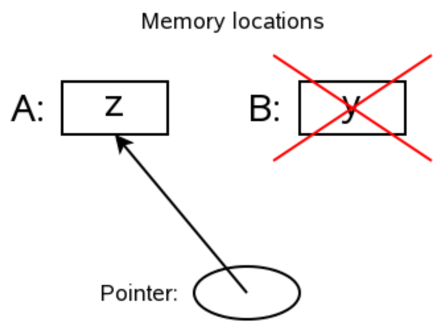
Now assume, a thread T_1 reads the pointer in Figure 2.4(a) that points to a memory address A storing value x. After reading the value, T_1 performs computations based on the assumption that the pointer refers to value x. Another thread T_2 also starts the operation and reads x. Now, T_2 finishes its computations and hence, erases value x and creates a new value y at address B. Then, it changes the pointer to the new address B as shown in Figure 2.4(b). However, before T_1 even completes its own first operation, T_2 starts a new operation. This time it deletes value y and creates a new value z but accidentally at the memory address A again. If T_2 finishes the changing of the pointer as in



(a) Pointer points to address A with value x



(b) Pointer points to address B with value y; value at address A has been deleted



(c) Pointer points to address A with value z; value at address B has been deleted

Figure 2.4: Example for the ABA-problem

Figure 2.4(c), before T_1 can execute the last CAS instruction, T_1 will change the pointer again without value z ever being read!

The ABA-problem for pointers can be solved by careful garbage collection so that a memory address is never reinitialized if there is still a thread using the value at the location. Anyhow, the ABA-problem can occur in other settings and must always be considered during the design of non-blocking implementations.

2.7 Software Transactional Memory

STM is a relatively new programming paradigm that has recently been an area of intense research. The goal of STM is to provide atomicity for critical sections, with the possibility of discarding and retrying computations. A transaction is a code block containing a critical section that must be executed atomically with respect to other transactions.

A globally accessible variable or object where updates may be discarded is called a transactional memory object or TM object for short. The accesses to such an object are handled and coordinated by the underlying STM implementation. A transaction might be aborted during its execution or at the end, after a final check, if a conflict occurred. If no conflict occurred, the changes are committed and thus atomically made visible. Let us have a look at the following code example.

```
1                          x := 0;
2                          y := 0;
3
4   atomic {                       |   atomic {
5        if ( x == 0 )             |        if ( y == 0 )
6            y++;                  |            x++;
7   }                              |   }
```

In this example, there are two variables, x and y that are initialized to 0. Here, x and y are TM objects. Now, if there exist critical code fragments, synchronization is done by wrapping the fragments into atomic blocks as denoted in the example. Hence, all the synchronization management is performed by the underlying STM. If there was no atomic block wrapping of the two critical code fragments, due to bad timing it might happen that a thread first executes the check in line 5 on the left-hand side. Then another thread might execute the check in line 5 and the incrementation of x in line 6, both on the right-hand side. The first thread would not notice that the value of x has changed in the meantime and still execute the incrementation of y. This would lead to the undesired situation that both, x and y have been incremented, despite the obvious intent of avoiding exactly this.

With the aid of STM, the second thread commits its result. The first thread still increments y but now during validation of the operation (or rather *trans-*

action as called in an STM context) the conflict with the operation of the second thread is detected and the result is not committed. Instead the thread will retry the entire atomic block and realize that in line 5, x is no longer 0 but 1.

One of the big challenges for STM is to realize a correct validation and commit phase. There are many suggestions on how to realize STM to solve this and other problems. On the one hand STM implementations can be lock-based [Enna06, DiSS06, GSVW$^+$09, SATHM$^+$06]. On the other hand they can also be non-blocking [Fras03, HaFr03, HLMI03, TMGH$^+$09, ViIS05].

There are many identified issues with STM for example concerning isolation of transactions and ordering of operations [SMATB$^+$07]. Furthermore, in many cases the use of STM simplifies the implementation but does not lead to better performance than locking approaches. In this section, an example for a naive STM-based priority queue implementation will be given and is compared to a sophisticated fine-grained implementation. This example shows what kind of modifications are sometimes necessary for obtaining a feasible performance. Since the fine-grained implementation is one of the contributions of this dissertation, it is explained in detail.

2.8 Concurrent Priority Queue Examples

The verifier implemented in this dissertation used several use cases of concurrent priority queues for verification. In the following a number of implementations of concurrent priority queues are described that use one of the previously introduced programming paradigms each for showing how the use of the different synchronization mechanism can look like.

2.8.1 Lock-Based Priority queues

The coarse-lock approach for a priority queue uses a single global lock that protects accesses to a binary heap. The global lock forces *insert* and *remove* operations to be executed strictly sequentially. This is a very simple implementation because it has only to be ensured that the sequential implementation of the binary heap is correct. Nevertheless it is a simple case study for the verifier implemented in this dissertation.

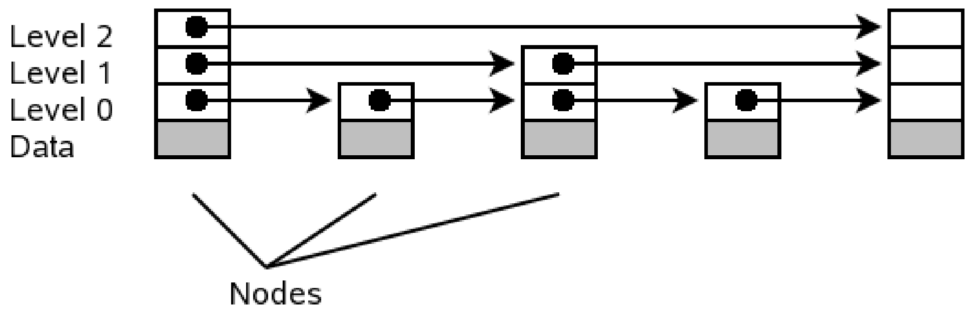


Figure 2.5: Illustration of a skip-list

Hunt et al. [HMPS96] present a refinement of the coarse-locked approach. It uses mutual exclusion locks to protect the heap size variable as well as each node in the binary heap. Consequently, Hunt's fine-grained locking approach increases parallelism. The heap size lock is only held at the beginning of an operation. As soon as the heap size variable is updated and the necessary initial locks for the operation are acquired, the heap size lock is released. The *insert* and *remove* operations are similar to their sequential counterparts. Each node that has to be modified is locked before executing the modification of its content. *Remove* operations are considered to be of higher priority than *insert* operations. The node's locks acquired by a *remove* operation are not released until the operation is completed.

In contrary, the *insert* operation releases its lock after every swapping of elements, even if the swapping has not succeeded. Nodes are also tagged to indicate whether a node is empty, valid, or in a transient state due to an ongoing *insert* or *remove* operation. Using these tags, global consistency is maintained while *insert* and *remove* operations can be executed in parallel. Finally, the algorithm reduces contention during *insert* operations by spreading out the insertion points of consecutive *removes* across disjoint sub-trees so that conflicts between two *insert* operations occur as late as possible. Hunt et al. call this the "bit-reversal" technique, which is basically the same idea as the LR-algorithm [Ayan90].

2.8.2 Hand-Crafted Lock-Free Priority Queue Example

Sundell and Tsigas present in [SuTs05] a fast lock-free priority queue that is based on a sorted skip-list [Pugh90]. A skip-list is an extension of a linked-list with additional references that act as short cuts as shown in Figure 2.5. These short cuts allow for efficient searches comparable to binary trees. Short cuts are randomized and follow a geometric distribution, based on the "level" of a node. A node on level n contains n short cuts, a node on level 0 has no short cuts and references its direct successor only.

One of the main problems with non-blocking list algorithms is that multiple references have to be changed in a consistent way for each operation. The lock-free skip-list achieves this by marking not only nodes but also the references that point to nodes that are worked on. References are always changed starting at level 0 up to level n in order to assure consistent changes. If a reference is marked by a concurrent insertion, a helper function is invoked that completes the operation that has been started by the other task. This helping strategy is essential to achieve the lock-free property as it allows a task to continue even though another task has unfinished work.

Sundell's algorithm is lock-free and linearizable. The version of the algorithm described in [SuTs05] is not quite as general as other implementations because it only allows a single element per priority. However, there exists an accelerated, commercial version of this algorithm by them that has been modified to deal with this issue. For the verification tests, the version published in [SuTs05] is used.

2.8.3 Comparison of the Performance of Lock-Free and Lock-Based Approaches

Priority queues are often used at the core of schedulers. The performance of the queue is a critical factor because bad or erratic performance affects scheduling. Scalability in the number of threads is another property that is crucial for concurrent algorithms. The three algorithms have been implemented in Java, and the performance tests are executed on a Java 6.0 runtime environment on a Linux SMP system.

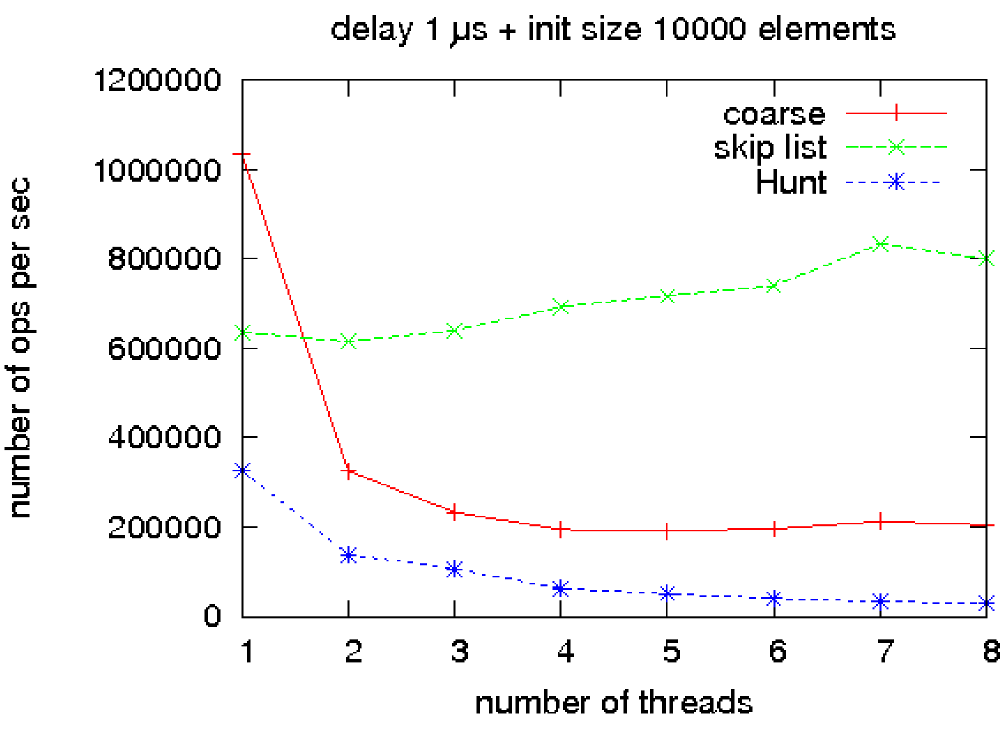


Figure 2.6: Comparison of the lock-free skip-list, Hunt heap and coarse-locked binary heap

Figure 2.6 presents example results showing of the three example priority queues described before. The scalability of the lock-free priority queue is obviously higher than the scalability of the lock-based pendants. The priority queues were initialized with 10,000 elements before the test started.

Each thread chooses randomly whether it inserts or removes an item. So at some random point in time, there may be any number of insertions and deletions executed concurrently which is only limited by the number of threads initialized. In the case of an insertion, an item with random priority is put into the queue. For a more realistic behavior, after each operation, there is spent a time of 1 μs in local work.

These results motivate the use of lock-free instead of lock-based approaches for multicore systems. For a detailed analysis of the three queues presented here and additional queue examples refer to [DrBa08].

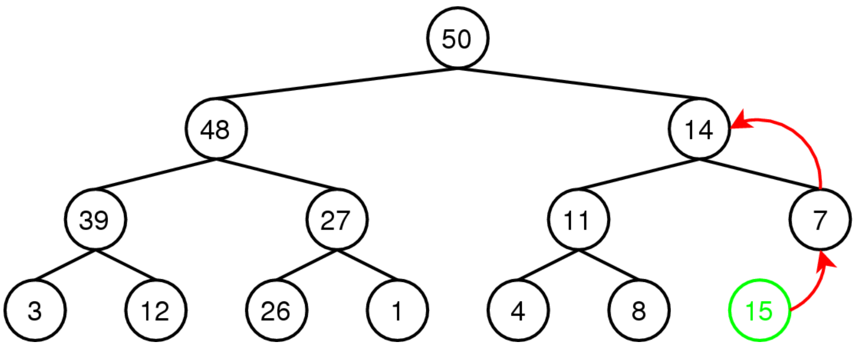


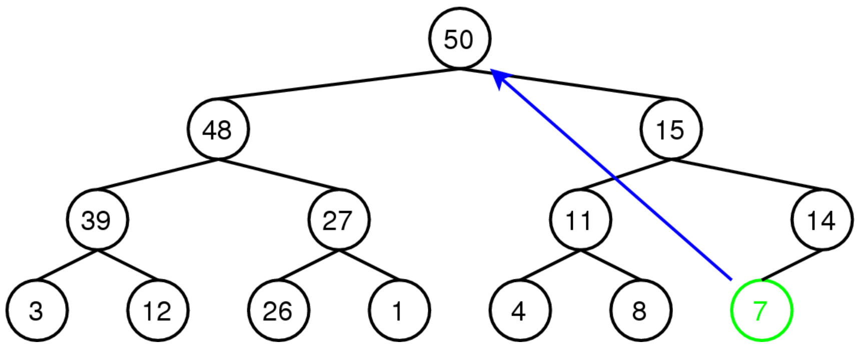
Figure 2.7: Inserting an element into a binary heap

2.8.4 Naive STM-Based Concurrent Binary Heap

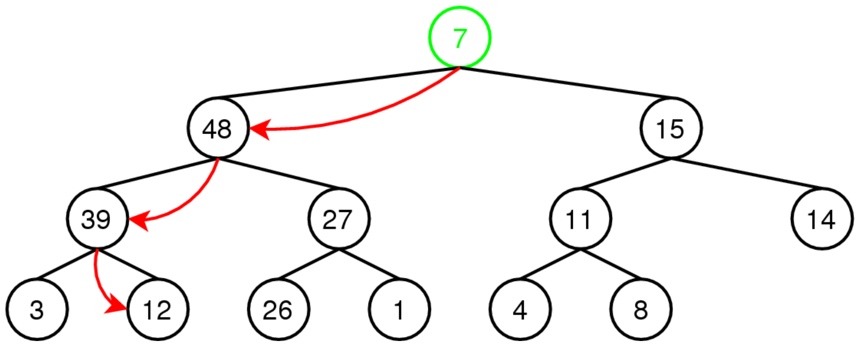
STM can be applied in different ways for implementing a binary heap. One possibility is the naive STM-based heap, where both operations, *insert* and *remove*, are performed exactly in the same manner as for the classic sequential binary heap. The only difference is that each operation is a transaction, whereas in the sequential version only one operation on the heap may be performed at a time. The heap is represented by an array of TM objects. The array has a pre-determined capacity.

An insert operation inserts an element as a leaf node at the first free position in the heap as illustrated in Figure 2.7. For determining this position, a size variable is used, which is also a TM-object. Then the element is bubbled up meaning that it switches positions with parents of lower priority until it encounters a parent node of higher priority then itself. Finally, the heap size variable is updated, which finishes the insertion. All these steps have to appear atomic with respect to other operations, otherwise inconsistencies may appear.

The same applies for the *remove* operation. The steps illustrated in Figure 2.8 have to appear atomic and are therefore executed in a single transaction. First, the peak element, which is by definition the element of highest priority, is removed by replacing it by the last element in the heap (see Figure 2.8(a)). Second, the replacing element is bubbled down by switching positions with the child with higher priority (Figure 2.8(b)). The switching of positions ends when either both children are of lower priority or the bottom has been reached.



(a) Replacing the peak by last element



(b) Consolidating the heap

Figure 2.8: Deleting the element of highest priority from a binary heap

2.8.5 Fine-Grained STM-Based Concurrent Heap

A clear drawback of this implementation is that every operation will conflict with another operation if they occur concurrently. Each operation accesses the heap size variable, and thus these operations obstruct each other. This means that effectively only one operation at a time takes effect. Therefore a more sophisticated solution has been developed that avoids those disadvantages [DrBa09].

The first new concept that is introduced to overcome the issues of the naive implementation is the dissociation of the heap from a fixed size value. Only a range is provided for having a rough idea of where the end of the heap is. This is done by introducing two variables "DENSE" and "LAST" which give an indication up to which place in the array representing the heap, there are no "FREE" items, and where the last element in the heap is located, respectively. Those two variables form basically the administrative data. They have not to be precise. However, they have to be made ABA robust.

The next feature is the reindexing borrowed from Hunt et al. [HMPS96] and Ayani [Ayan90]. Lastly, multiple transactions are used to realize an operation. The important point here is that by executing the transactions, the logical heap invariant is never broken before or after a transaction. Furthermore a tag is introduced for each position in the heap that can have the values "FREE", "DIRTY", "HOLE" or "OCCUPIED". A position tagged as "FREE" means that it contains no element. An insert operation sets a "FREE" position to be "DIRTY", for indicating that an insertion is already running. A position will contain an "OCCUPIED" node if it contains an element of the heap. A delete operation replaces the "OCCUPIED" node of highest priority by a "HOLE". With the aid of these tags, a division of the operations into multiple transactions can be realized.

For example the *insert* operation is divided into multiple transactions instead of a single one. Four steps are executed in four transactions:

1. Find a position that is tagged "FREE" from where we can start inserting our element (one transaction)

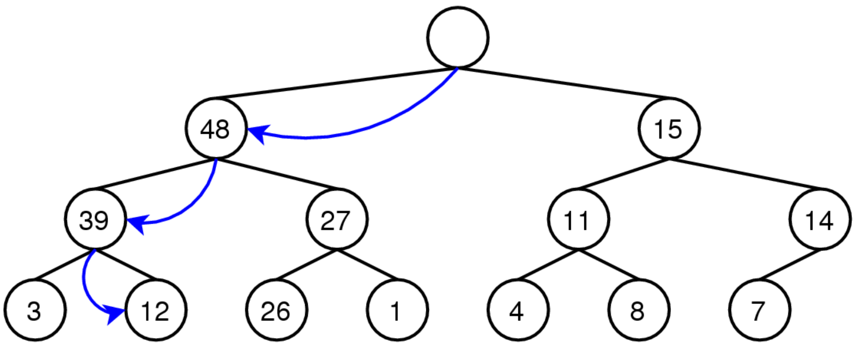
2. Update administrative data
3. Perform "bubble-up" of the element (one transaction)
4. Update administrative data (two transactions)

The second step can be done by a mere CAS instruction on the "LAST" variable. The third step is a common insertion step for binary heaps. The pseudo code for the first and last step is shown bellow.

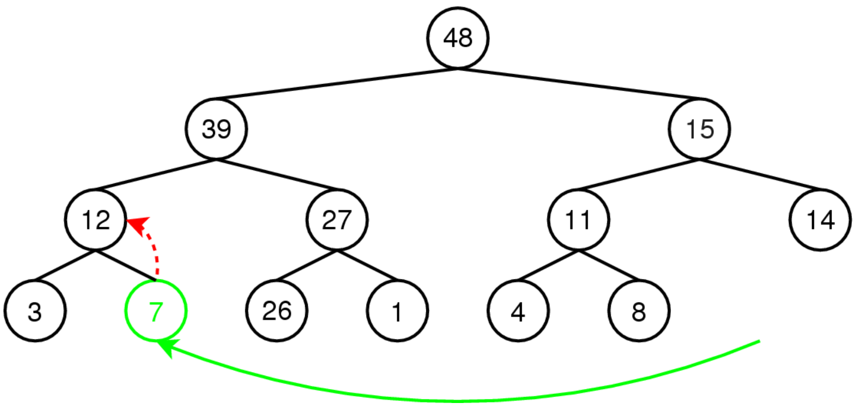
The *remove* operation is also divided into multiple steps. Beside the fact that all concurrent *remove* operations access the heap size variable in the naive version, the main problem is that the target element for a removal is always the peak element. To overcome this problem, the node value "HOLE" is introduced and the *remove* algorithm is changed. Instead of replacing the peak element by the last element in the heap as done in the naive version, it is replace by a "HOLE". This "HOLE" is handled only by the removing operation and ignored by all other operations. Now many concurrent *remove* operations can make progress because if the first element in the heap is already a "HOLE", simply the next logical peak element is searched recursively. Furthermore, by this new tag value, there is an implicit solution for the difficult case when insertions and removals occur concurrently. The entire operation is divided into four steps (compare Figure 2.9):

1. Find the peak element and replace it by a "HOLE"
2. Bubble down the "HOLE" (multiple transactions)
3. Replace the "HOLE" by one of the last elements in the heap and bubble it up
4. Update administrative data (two transactions)

The fine-grained STM-based heap only uses STM constructs that are provided by Fraser's implementation for reading and writing transactional objects. But there still remains the question of how some aspects of the heap might be



(a) The peak element is replaced by a "HOLE" that is bubbled down to the bottom



(b) The last element replaces the "HOLE" and is bubbled up if necessary (not the case here)

Figure 2.9: Deleting the element of highest priority from the STM-based binary heap

improved. An entire transaction for an update of “DENSE” and “LAST” seems extreme for such a small task. Furthermore, checking the nodes during the update of the administrative data is a read-only transaction which does not even need to be fully precise. Therefore, two other aspects of the heap are optimized.

So far, the variables “DENSE” and “LAST” are updated within transactions in the fine-grained implementation. We improve this by using 16 bits for versioning and 16 bits for the value which can be together updated with a compare-and-swap (CAS) instruction. Using CAS instead of transactions in extenso saves much of the overhead associated with accesses to TM objects.

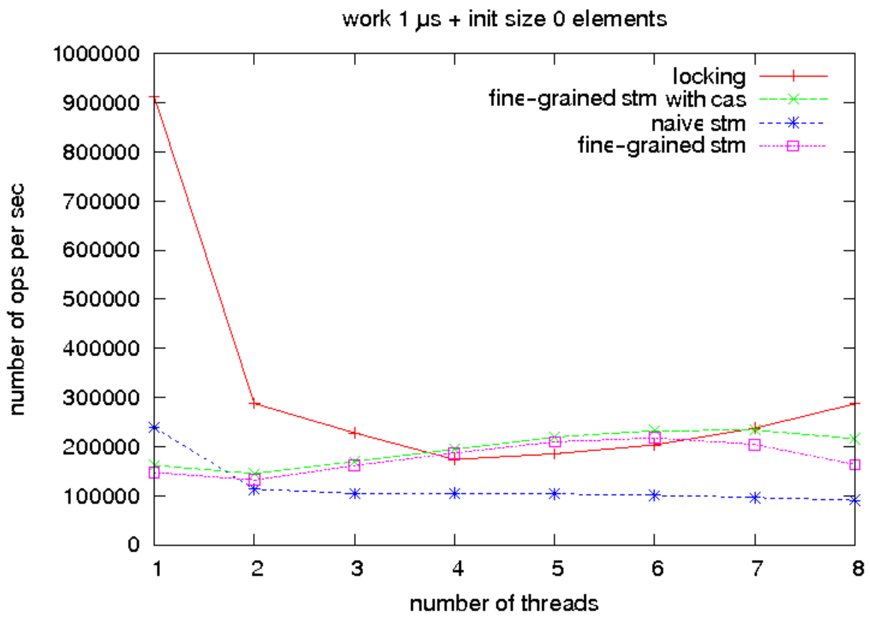
There is also a “fast access” function on TM objects introduced. The idea is that if a node is “FREE”, only insert operations will access this node. An insert operation will therefore tag the node as “DIRTY”, which can be done by CAS. By reading TM objects outside transactions, the administrative overhead associated with a real transaction is avoided. Furthermore, it is not significant which “FREE” slot is used. The “fast access” function is also used during the check of “DENSE” and “LAST”.

The described implementation of an STM-based heap is one of the first complex and optimized implementations using STM. However, there is no currently available automated verifier that could prove that the output produced by this algorithm is certainly valid according to the definition of a priority queue and the linearizability criterion. For achieving a verification of the implementation, the verifier described in the next chapters is able to provide an answer by analyzing the output of the program.

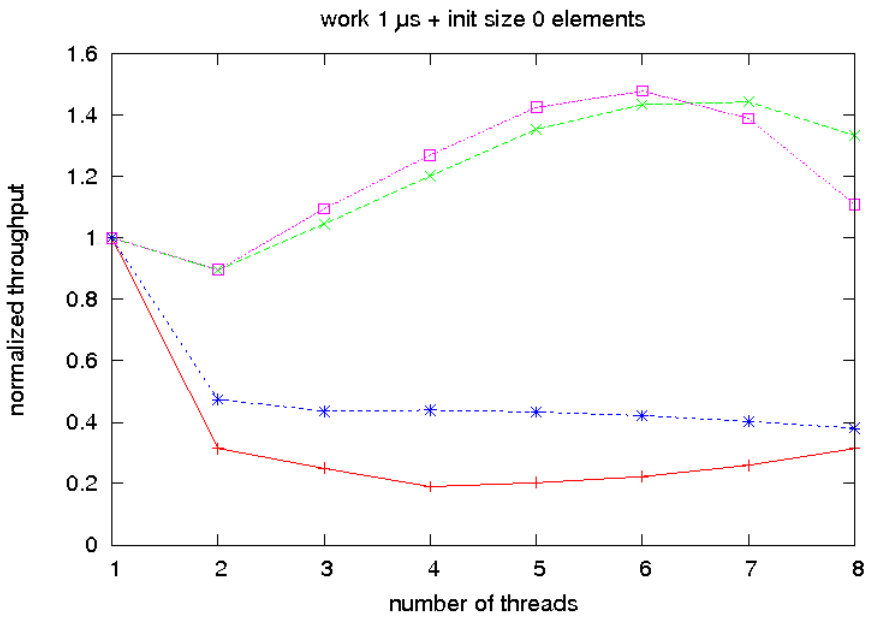
2.8.6 Comparison of the Performance of the Naive and Fine-Grained STM-Based Binary Heap

All algorithms tested are implemented in C using the STM library by Fraser [Fras03]. For the lock-based implementation pthread mutexes are used. The tests were conducted the same way as in section 2.6.

In Figure 2.10 the graphs for the results of a test scenario with an initially empty queue and little local work is shown. For a single thread, the locking



(a) Absolute throughput performance



(b) Graph normalized with respect to the performance of one thread

Figure 2.10: Comparison of the different STM-based binary heaps and a lock-based reference implementation

implementation exhibits, as expected, a much better performance than all STM-based heaps. Because of its simple principle, the naive STM-based heap achieves a superior performance compared with the other STM-based heaps for one thread as shown in Figure 2.10(a). As soon as there is more than one thread, the performance for the locking implementation breaks down, whereas the two fine-grained STM-based implementations show a remarkable degree of scalability if compared to the other two implementations.

The naive STM-based heap scales poorly as presented in Figure 2.10(b). The two fine-grained implementations exhibit comparable scalability, but the optimized implementation has a noticeable advantage. For more than three threads, the fine-grained STM-based heaps catch up with the locking-based one.

Furthermore, the locking-based heap shows an interesting behavior for more than five threads with increasing throughput. This can be explained by the CPU-hopping effect, which is caused by the process scheduler in the Linux kernel because it attempts to perform load-balancing between CPU cores.

2.9 Summary

This chapter has given an overview of the area of concurrent programming and has shown that verification is an aspect that is very important and challenging for parallel programs. It has explained the lock-based approaches as the commonly used solution. Then, it faded into non-blocking approaches which use non-blocking instruction primitives like CAS. After that, it introduced STM as an abstract alternative to achieve correctly synchronized programs.

The main problems for each method has been explained. On the one hand, lock-based implementations often run into problems like deadlocks or performance issues due to sequentialization. On the other hand, non-blocking code is very difficult and complex in development. STM-based implementations currently need non-trivial optimizations to become competitive.

The final statement of this chapter is, no matter which alternative has been chosen, there is always a need for verification. Only when the parallel code is verified, there is a certainty that the code is valid and correct. The next chapter provides basic knowledge about verification.

3. Correctness Reasoning

When reasoning about correctness, there are a lot of different aspects that can be considered. This chapter gives an overview of these aspects and how the work for this thesis is placed in the area of program verification.

First, there is an outline in which respect a program can be verified. Then, the different correctness criteria in the field of verification of parallel programs are introduced. Finally, related work of the different methods for verifying parallel programs on linearizability [HeWi90] is described.

3.1 Correctness Fields

Automated verification of sequential programs has been explored for decades. There are numerous methods for verifying different aspects of code, such as memory safety, detection of memory leaks, and proving compliance with the program invariants [AHMQ+98, BBCL+06, BHJM07, ChSh04, XiCE03]. Without exception, each method lacks in certain factors. None of them can verify every aspect with absolute reliability. Often there are tolerances for false negatives or false positives.

Verifying correctness of parallel programs is even more difficult because there are concurrent operations that overlap and thus, the potential sequences of instructions that are executed is hard to cover. If we consider parallel programs where different threads communicate over shared memory, we may have many points where threads might interfere with each other. It is very hard to cover all possible interferences of different threads of execution. Another problem is that even if the design of a program is correct, the implementation might still have subtle errors. This is often the case because of the complexity of programs of this kind. In this dissertation, mainly data structure implementations shall be verified by the method introduced.

Verification addresses three different aspects. The first aspect is proving that the design of the algorithm is correct. This happens before the program is implemented. The second aspect is determining whether a program is at all executable. This involves issues such as detecting possible deadlocks or livelocks [EnAs03], memory leaks and the like. The third aspect is whether an execution fulfills the specification of the program. Here, the challenge is to discover whether an implementation really implements the specified algorithm correctly. This thesis focuses on this third aspect.

3.2 Correctness Criteria

A programmer who implements a program has a certain image in mind of what the program shall perform. This image has to be formally expressed by a specification for allowing formal treatment. Finally, the program has to be checked whether it produces output according to its specification with respect to consistency.

In the sequential case, this is non-ambiguous because we assume that the program has a definite state at each point in time of the execution. However, if the program is executed in parallel it is not clear how overlapping operations should be treated and considered. When two operations, A and B, overlap, both is possible, that operation A takes effect before operation B or the opposite. Nevertheless, there might be desired constrictions or properties on the order of overlapping operations. Therefore, a correctness criterion has to be chosen before a program can be verified. In the following, it is described which constrictions and properties those might be and why they are useful when reasoning about correctness.

First, the general term *consistency* is introduced. Second, there is an explanation of how self-defined correctness with quality criteria may look like and an outline of how *weak consistency* criteria look like. After that, *casual consistency* is described. Then, the more strict criteria *sequential consistency* and *serializability* are explained. Finally, the practically most strict criterion for concurrent programs called *linearizability* is introduced.

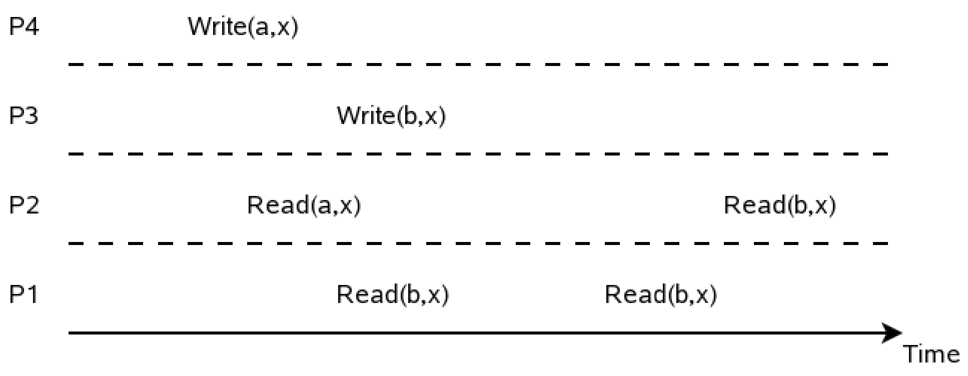
3.2.1 Correctness Reasoning

A consistency model is necessary when there is a set of threads of execution that share the same data. Traditionally, this is regarded in the context of *read* and *write* operations. A *read* operation is expected to read the last written value at a memory address.

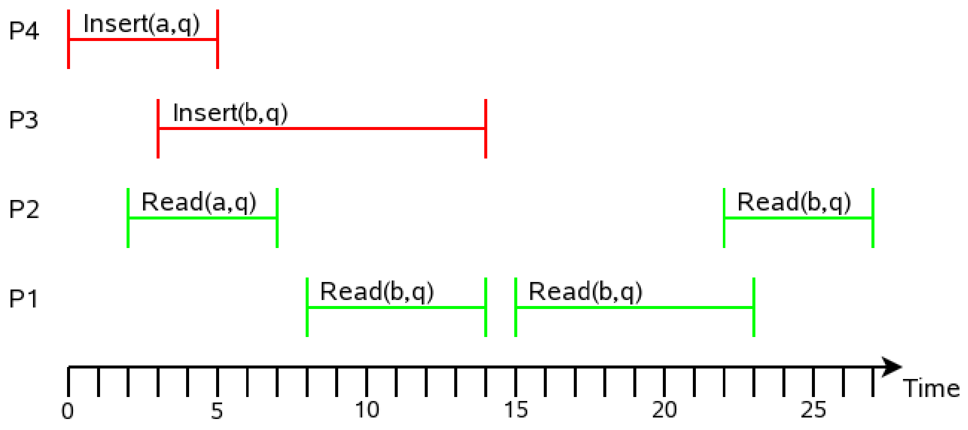
Furthermore, there are different granularity levels, where consistency can be considered. While reading and writing are very basic operations, it is sometimes more interesting to consider a sequence of *read* and *write* operations. Therefore, operations are grouped into critical sections that shall appear to have been executed atomically. Depending on the implementation principle used, the execution of critical sections may overlap. Still, it is important that *read* and *write* operations are executed in a consistent way, but grouping those operations adds more complex semantics to the program. However, additional information is needed because it has to be known when a critical section has actually been entered and left.

Throughout this document the execution of a program is modeled as in the following. The term history refers to a sequence of events that denote that a certain critical section has been entered or left. Since critical sections usually implement complex operations, the term *operation* is used instead of *critical section*. Instead of saying that a critical section has been entered by a thread of execution, it is also possible to state that the thread has started an operation. Instead of saying that a critical section has been left by a thread of execution, it is also possible to state that the thread has ended an operation. Consequently, the event of starting an operation is called a start event, and the event of ending an operation is called an end event. Note, that Herlihy [HeWi90] denoted the start event as an *invocation event* and the end event as a *response event*. Nevertheless, it is the same principle but throughout this document the more intuitive terms start and end event will be used.

Figure 3.1 presents two histories for both granularity levels. In Figure 3.1(a) a history of finest granularity is illustrated. There are only *read* and *write* operations included. $Write(a, x)$ means value a has been written to address x while $Read(a, x)$ means value a has been read from address x. Obviously, only the



(a) Process granularity



(b) Operation granularity

Figure 3.1: Examples for execution histories

order of operations is taken into account, here. Whereas in Figure 3.1(b) information has been given when which critical section or operation has started or ended in relation to other operations. This leads to the necessity of considering overlapping entities.

Correctness reasoning is about whether a program produces always histories that meet a certain criterion or not. Since complex programs with non-trivial operations shall be analyzed, the coarse-granular histories are used as the bases for the verification process.

3.2.2 Arbitrary Correctness

If a programmer perceives that the program or data structure he is implementing is not used in a critical context, he might decide to apply a self-defined arbitrary correctness criterion. This can be done for example by defining a metric for the quality of the output. The concept of arbitrary correctness is also called continuous consistency. The following formalism is a suggestion of how quality reasoning for continuous consistency might be done.

If we consider a program P and its output O_P, we define that the quality function Q of some program output denotes a value that describes how "good" the output is. The programmer has to decide which quality is accepted and thus which values $V_{Q_{good}}(O_P) = \{Q(O_P) | Q(O_P) \text{ is "good"}\}$ are allowed for a program to be considered as correct. This implies a characteristic function $\mathbb{1}_Q$ of the quality as in the following:

$$\mathbb{1}_Q(Q(O_P)) = \begin{cases} 1 & , \text{ if } Q(O_P) \in V_{Q_{good}}(O_P) \\ 0 & , \text{ else} \end{cases} \tag{3.1}$$

Obviously, this characteristic function is 1 if the quality of O_P is "good" enough and 0 otherwise. Now, the program output might consist of a set of output units $O_P = \{o_1, o_2, ...\}$. In this case, the quality would be defined as a function f of the different qualities of all units. Therefore, another quality function Q' is necessary for the output units

$$Q(O_P) = f(Q'(o_1), Q'(o_2), ...) \tag{3.2}$$

Those output units might consist of more output units which again need a description of quality until a component level can be found that is in this sense atomic and not further decomposable.

A concrete example might be an arbitrary concurrent implementation of a FIFO queue. FIFO is an acronym for "First-in, First-out". A FIFO queue in the sequential case is defined as a queue with two operations, *insert* and *remove*. The first item added to the queue is the first item to be removed from the queue. The programmer of the concurrent version of the FIFO queue might define following criteria

1. An item is correctly removed if its order is among the lowest 10% of inserted items
2. The implementation is satisfactory if correct items have been removed 95% of the time

In this example, the program output is a sequence of items returned by *remove* operations. Those items are the output units. The entire reasoning could be also taken into an even higher level, if we had a program implementing a system of x FIFO queues. Then it could be defined that the system works correctly if at least 90% of the FIFO queues work correctly.

If we go away from general data structures to specific applied areas, there might be a criterion used that is applied on numerical values or the staleness of data. Consistency for numerical values [YuVa02] defines the tolerated discrepancy of a value read to the actual value at a certain point in time. For example it could be defined that the temperature read for some place may not differ by more than $1°K$. The tolerance could be also defined relatively. Consistency for staleness would mean, that the data read may not be older than for example 2 minutes. This is useful if for example there is a program that shows which TV show is running at the moment at a certain channel.

For both, the numerical consistency and staleness consistency, the quality reasoning introduced before provides a formal description of the desired criterion.

3.2.3 Weak Consistency

The quality reasoning introduced in the section before is very specific for each application. It is rather a paradigm than a real principle.

Consistency criteria that are considering the order of operations base on abstract formalisms and thus are more general than the validation of quality. There exist a lot of weak consistency criteria that cannot be covered in detail in this document. Further, weak consistency criteria are not of relevance in the scope of this work, since they address larger scale systems like distributed clien-t/server systems. The assumption is that that the probability for simultaneous updates is low and if inconsistent updates occur they can be easily resolved. Consequently, many inconsistencies can be hidden for the client. The idea of resolving inconsistencies for the client is denoted as eventual consistency. For further reading about eventual consistency refer to Tanenbaum/van Steen [TaSt07].

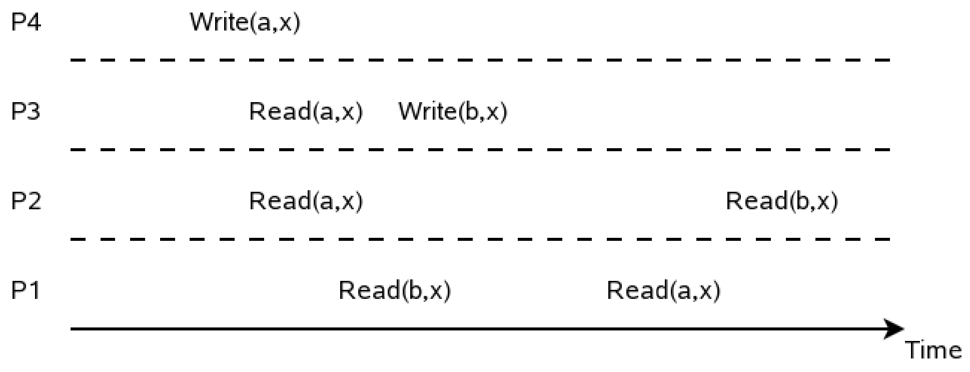
3.2.4 Causal Consistency

Causal consistency [HuAh90] is a criterion that is better considered at a process level than in an operation level because it takes into account whether different operations influence each other from a global point of view. The basic idea is that if an event e_1 causes another event e_2 then everybody has to see the effect of e_1 first, and afterwards the effect of e_2.

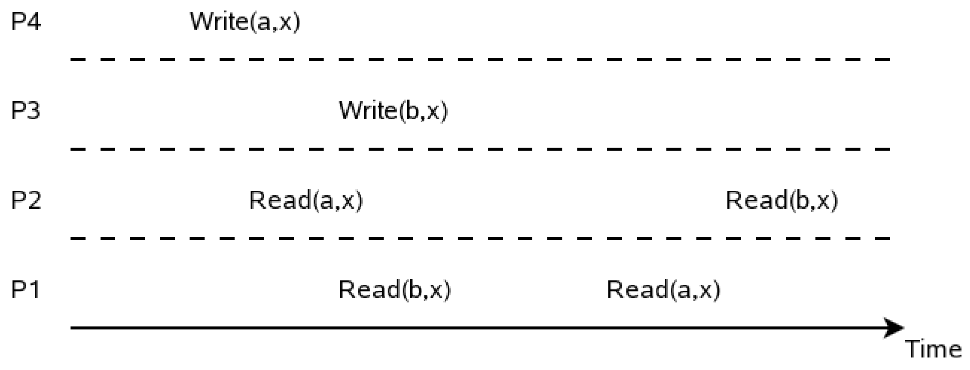
Figure 3.2 shows two histories. Figure 3.2(a) illustrates a history that is not causally consistent because obviously process $P3$ reads the value a at address x before it writes value b to address x. Consequently, writing b potentially depends on the reading of a and thus a must be read before reading a value of b at x. However, process $P1$ reads b at x before it reads a. Hence, the history cannot be causally consistent.

The history represented in Figure 3.2(b) is causally consistent. Even though the processes $P1$ and $P3$ read the values a and b at address x in different order, the writes of a to x and b to x from the processes $P3$ and $P4$ are done independently. Thus, there is no potential for causal relation so both orders are considered consistent and for that reason correct.

For accomplishing a verification on causal consistency, a dependency graph of operations is necessary so that it can be tracked which operations cause others. Since mainly data structures shall be verified by the method introduced in this document there is a potential dependency among all operations occurring in a history. So, for our purposes causal consistency is not adequate and the additional effort for building a dependency graph is redundant.



(a) This history is not causally consistent



(b) This history is causally consistent

Figure 3.2: Examples for causally consistent and causally inconsistent histories

3.2.5 Sequential Consistency

Sequential Consistency [Lamp79] is a criterion that argues that any order of concurrent operations is valid as long as all threads of executions see the same order. Obviously, the set of histories that are sequentially consistent are a subset of the set of histories that are casually consistent because if a history is sequentially consistent it means that all threads of execution see the same order of operations. In particular, this is true if there are dependencies among operations as explained in the previous chapter.

However, there exist causally consistent histories that are not sequentially consistent. Figure 3.2(b) presents such an example. It is causally consistent but it is not sequentially consistent because the processes $P1$ and $P2$ do not have

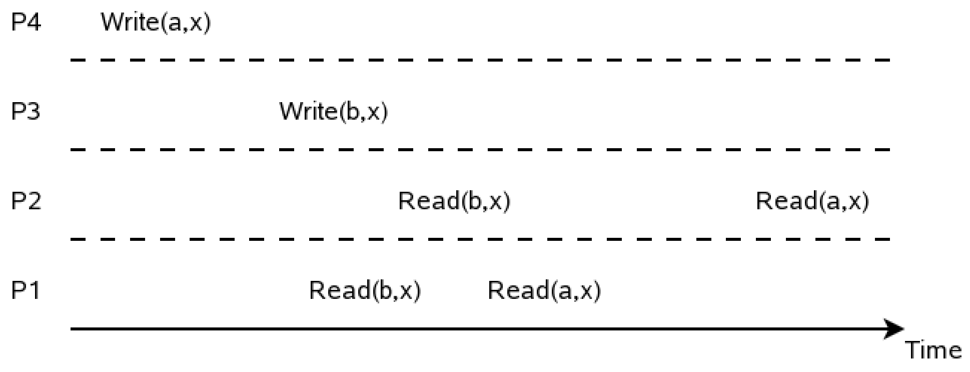


Figure 3.3: A sequentially consistent history

the same view of the order of updates. While for $P1$ $Write(b, x)$ appears to be the first executed operation, $P2$ sees the update $Write(a, x)$ first.

Figure 3.3 demonstrates a sequentially consistent history. Apparently, time is no critical factor for this correctness criterion. It is clear that the operation $Write(a, x)$ took place before the operation $Write(b, x)$ with respect to the time line. However, both processes $P1$ and $P2$ see the updates consistently the same way in opposite order. For many applications this is an adequate behavior and thus it is often used as a correctness criterion. Still, there are applications where such a behavior is not desired.

Let us consider a program with critical sections where higher level semantics are of significance. A priority queue as introduced in section 2.7 implements two complex operations *insert* and *remove*. Both operations consist of sequences of atomic *read* and *write* operations. The semantics of the priority queue in the sequential case is that a *remove* operations removes an element of highest priority from the queue q. $Insert(1, q)$ means the thread has inserted an element with value 1 into the queue q while $Remove(1, q)$ means that an element with value 1 has been removed from q. For simplicity, we identify the value of an element with its priority where higher values imply higher priorities.

Figure 3.4 shows a history with complex operations realized as critical sections with obvious start and end points. It is evident that both *insert* operations are finished before any of the *remove* operations have started. Furthermore, it is also evident that the thread $P2$ ends its operation before $P1$ starts its

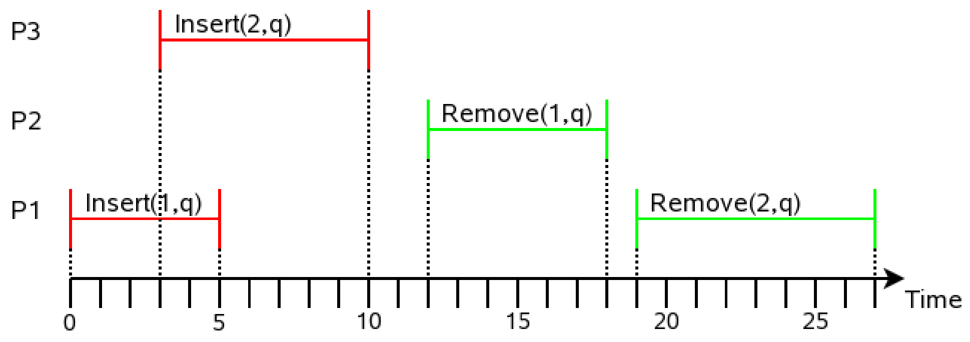


Figure 3.4: A sequentially consistent history with critical sections

operation. Hence, from an intuitive view $P2$ should remove an element of higher priority than $P1$ but it does not. Still, this history is sequentially consistent as we learned before. Now, if there is a time critical application that urgently needs to realize a strict priority schedule, this behavior is not acceptable. Moreover, this behavior is not intuitive with the sequential behavior of a program in mind.

3.2.6 Serializability

If critical sections and their start and end events are considered instead of atomic *read*s and *write*s the term serializability [BeHG86] can also be used. Beyond that, serializability allows multiple objects to be manipulated within a critical section. Usually, when respecting the semantics of objects the term serializability is rather used than sequential consistency.

The validation of a history with complex operations is typically done by ordering the start and end events with respect to the following two properties

1. All start and end events can be reordered to obtain a sequential history as long as the relative order among events of the same thread remain the same

2. The obtained sequential history is correct according to the sequential invariant of the object or objects

In this context, a sequential history is a history in which each end event of an operation follows the corresponding start event of the operation. It is called sequential history because the operations could have been executed in the same manner by a single thread or process.

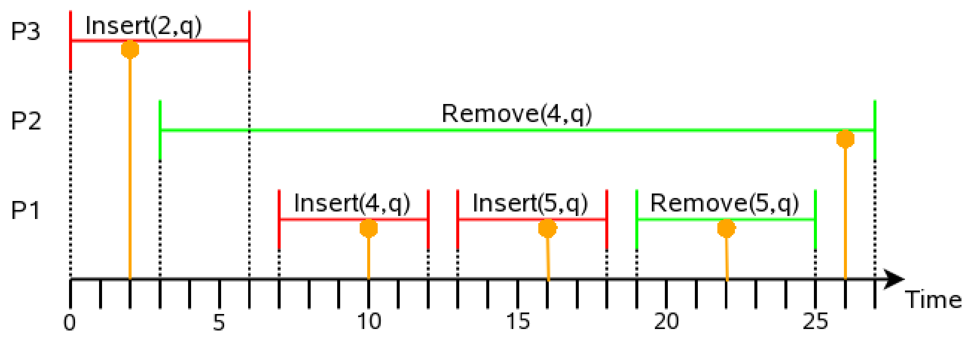
3.2.7 Linearizability

Linearizability has been defined by Herlihy and Wing [HeWi90] and is a stricter criterion than serializability or sequential consistency. It extends the two properties of a serializable history so that the following properties have to be fulfilled:

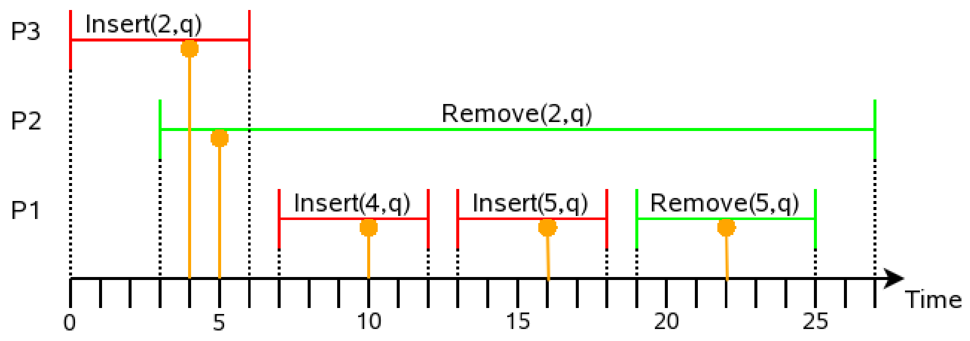
1. All start and end events can be reordered to obtain a sequential history as long as the relative order among events of the same thread remain the same
2. The obtained sequential history is correct according to the sequential invariant of the object or objects
3. If an end event of one operation occurred before a start event of another operation in the original history, then this order has to be retained in the sequential ordering

Getting back to the pictures of histories we had before, the idea of linearizability is that each operation is taking effect at some time between its start and its ending. Since a critical section is assumed to take effect in an atomic manner, an operation realized as a critical section has to take effect at some specific point in time. This point in time is called the linearization point. So, obtaining a sequential history of events with respecting the three properties defined before is eventually the same as assuming that all operations have taken effect at their assigned linearization point. Of course, the sequential history obtained from the assignment of linearization points may not break the object's invariant.

In the example of Figure 3.4 it is impossible to assign linearization points so that the priority queue invariant is kept valid. Irrespective of where we assign the linearization points within the intervals of the operations executed by $P1$ and $P2$, the operation $\mathit{Remove}(1, q)$ will always take effect before $\mathit{Remove}(2, q)$.



(a) $P2$ performs the last operation



(b) $P2$ performs the first operation

Figure 3.5: Examples for linearizable histories with assigned linearization points

Since both *insert* operations have their linearization point before the linearization point of $Remove(1, q)$ the invariant for a priority queue is broken, because 2 is the value of highest priority.

The examples in Figure 3.5 are both linearizable. Note that both histories in Figure 3.5(a) and Figure 3.5(b) contain the same operations with the same timestamps for each operation. The only difference is that the *remove* operation of process $P2$ removes the value 4 in Figure 3.5(a) whereas in Figure 3.5(b) it removes the value 2. Still each history is linearizable because in both cases the linearization points indicated as orange dots in the figures lead to correct sequential histories.

Linearizability matches most exactly the intuition of what is considered as a transfer from a sequential program to a parallel one. This is the reason why the verification method presented in this document deals with the verification of execution histories against linearizability. Another practical reason for examining linearizability is that most implementations that are sequentially consistent are also linearizable. It is hard to imagine why an implementation should behave sequentially consistent but not linearizable and there are hardly any examples for this behavior.

There is yet another correctness criterion which is called strict consistency which is even stricter than linearizability. It says that a *read* operation has to return the result of the latest *write* operation on an object or address. This is only possible with a global clock. Since this is practically not possible to implement, the strict consistency criterion is of no practical relevance.

3.3 Related Work

Verification in general can be done with three basic approaches. First, there is *proof-based* verification as the most difficult but the most complete one. Then, there is *model checking* as a more simplified but therefore computationally efficient approach. Lastly, there is verification by *simulation* and *testing* as the simplest approach and thus is easiest to be realized. In the following all approaches are described closely.

3.3.1 Formal Proof-Based Static Code Analysis

Proof-based verification is done by performing logical reasoning about the possible behavior of a program. Often logical reasoning of programs bases on Hoare's logic [Hoar83] or on some logic based on Hoare's logic. The main element of Hoare's logic is the Hoare triple $\{P\}S\{Q\}$ where P denotes the pre-condition, Q the post-condition, and S a program segment. The idea is to prove that if a program satisfies the condition P at some point in time, it will satisfy Q after the execution of S. Now, for obtaining a proof of a program segment there must be composing rules so that the program can be verified as a whole. The easiest rule as an example is the *rule of composition*

$$\{P\}S\{Q\} \wedge \{Q\}T\{R\} \Longrightarrow \{P\}S;T\{R\} \tag{3.3}$$

The rule applies for a program segment consisting of a segment T following a segment S. It says that the post-condition R of the entire segment for a pre-condition P is the same as the post-condition of T with the pre-condition Q that is the same as the post-condition of S executed for pre-condition P. Hoare used another notation for the same expression

$$\frac{\{P\}S\{Q\}, \{Q\}T\{R\}}{\{P\}S; T\{R\}} \tag{3.4}$$

Now, the Hoare calculus is only applicable for sequential programs. Therefore, extensions are necessary to make it usable for parallel programs. One of the first extensions is the one of Owicki-Gries [OwGr76]. For each thread a sequential proof is executed. The parallel Owicki-Gries rule requires that each thread does not conflict with the proofs of the other threads.

$$\frac{\{P_1\}S_1\{Q_1\}, \{P_2\}S_2\{Q_2\}}{\{P_1 \wedge P_2\}S_1 \parallel S_2\{Q_1 \wedge Q_2\}} \tag{3.5}$$

This means that all intermediate conditions of S_1 and S_2 must be preserved by all atomic commands in the sequences S_2 and S_1, respectively which is a very strong restriction.

Thus, rely/guarantee reasoning [Jone83] is an extension to Owicki-Gries that does not have the same restriction. The specification of rely/guarantee consists of the four components (P, R, G, Q)

P is the pre-condition that describes the assumption of the initial state before a code segment S starts

R is the rely condition that models all interferences of atomic actions from other threads that can be tolerated

G is the guarantee condition that models the impact on other threads by the code segment S

Q is the post-condition that describes the final state for the initial state P after executing the code segment S

If a code segment S satisfies a specification of the four tuple this is denoted as

$$S \text{ sat}_{RG}(P, R, G, Q) \tag{3.6}$$

There also exist proof rules for rely/guarantee reasoning as in the *RG-Weaken* rule example

$$\frac{S\ \mathrm{sat}_{RG}(P,R,G,Q), P' \Rightarrow P, R' \Rightarrow R, G \Rightarrow G', Q \Rightarrow Q'}{S\ \mathrm{sat}_{RG}(P',R',G',Q')} \tag{3.7}$$

This rule says that a specification is weakened by weakening its obligations which are the post-condition and the guarantee condition, and strictening its assumptions which are the pre-condition and the rely condition. There are lot more of rely/guarantee rules that apply for parallel programs.

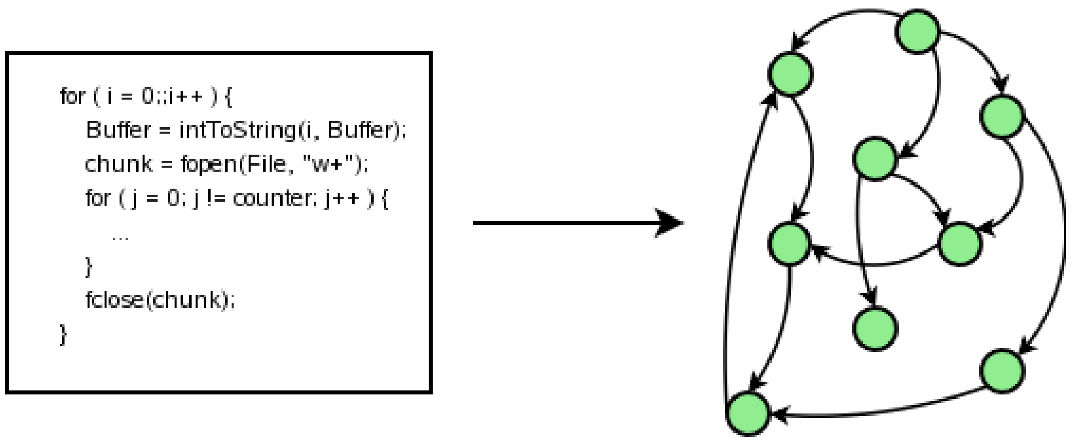
Rely/guarantee models interference as a relation without taking the control flow of the environment into account. Thus, it cannot prove directly properties that have dependencies on the control flow. Nevertheless there are workarounds for this but still there are programs for which a modular proof cannot be found.

Vafeiadis [Vafe07] in his PhD thesis combines elements of the rely/guarantee reasoning and separation logic to obtain a more powerful tool for concurrent verification called RGSep. He proposes modular techniques and makes two theoretical and two practical contributions to reasoning about fine-grained concurrency. RGSep permits ownership-based reasoning and ownership transfer between threads and maintains the expressiveness of binary relations to describe inter-thread interference. Furthermore, it describes a method for proving linearizability which introduces auxiliary single-assignment variables to identify the linearization point.

3.3.2 Model Checking

The principle of model checking [JrGP99] is that a system is described in some description language such that the model checker can check whether a certain property can be reached by the described system from an initial state. The property is described with some temporal logic formula whereas the system description consists of a set of atomic propositions that are associated with some state. The entire description may correspond to a finite state machine which itself might be for example a directed graph as schematized in Figure 3.6.

In the case of software verification, a proof can fail because of the undecidability of the halting problem and thus it is not generally applicable. Model



```
for ( i = 0;;i++ ) {
    Buffer = intToString(i, Buffer);
    chunk = fopen(File, "w+");
    for ( j = 0; j != counter; j++ ) {
        ...
    }
    fclose(chunk);
}
```

Figure 3.6: Schema of a program described as a directed graph

checkers cannot deal with the complexity of software. Therefore, techniques from classical proof-based verification or static analysis are needed like abstract interpretation of code, shape analysis or symbolic execution. Even with those extensions, heuristic searches are necessary for a verification of the entire state space.

Abstraction is the idea of representing a program by a simpler model. Hence, abstraction decreases the state space to a manageable size and allows the model checker to analyze a notation which it could not analyze before. It is also a necessary step because model checkers typically work on finite state models, whereas the state space of a program might be infinite. So the translation of a program to a finite state model is a necessary step and should be done automatically as well, although it might by done by hand which is neither general, nor effective.

Beyond that there exist state-less model checkers like Verisoft [Gode97]. In this approach there is no state storing and hence no state matching. The technology that makes state-less model checking possible is the partial order reduction which summarizes different transition sequences of the same transitions to one transition sequence if the final state is the same. The term *state-less* does not refer that there are no states considered in this approach but it refers to the fact that not all intermediate states a system model might enter are considered. Consequently, the states in the model are not system states but state sets which form a higher level of abstraction.

One approach that makes use of model checking for proving linearizability is to check the property by refinement relations as proposed by Liu *et al.* [LCLS09]. In this approach a model of the specification is defined that is realized as a finite state machine. Then the implementation is modeled by another finite state machine and it is checked whether the transitions of the implementation state machine forms a subset of the transitions of the specification state machine. By this refinement, relations from abstract specifications to concrete implementations are realized. The method avoids the necessity of identifying linearization points in implementations, but can also take advantage of linearization points if they are given. The search space is reduced by the aid of partial order reduction.

Vechev and Yahav [VeYa08] also use a model checker based on Liu Liu *et al.*'s to check their derived linearizable concurrent objects produced by their *Paraglider* tool. The tool systematically constructs algorithms for a concurrent data structure starting from its sequential implementation. The construction process combines manual steps with automatic exploration of implementation details.

3.3.3 Simulation and Testing Techniques

The oldest and most straight forward technique to check an implementation for correctness is to simulate or test it and check the output for correctness. For large outputs and complex invariants this is by no means trivial in the linearizability case. An automation is helpful here.

A method for testing has been first proposed by Wing and Gong [WiGo93]. There, they produce a history H from a concurrent program P_C. They use a sequential implementation P_S as the sequential specification of P_C. Then, they gain sequential histories of H by preserving the real-time order relation that is necessary for linearizable histories. Finally, P_S is run with the input of the histories gained and checked whether P_S will produce the right sequential history. If a correct sequential history is found, H is linearizable, otherwise it is not. Fraser also used a brute-force testing approach to verify the implementation of his STM [Fras03].

3.3.4 Chosen Verification Approach

The most complex data structures that have been verified by known methods are FIFO-queues. The advantage of this approach is, that it really verifies the implementation of a data structure and not an abstract specification. It is also less complex to implement such a verifier. The disadvantage of this approach is, that it does not deliver a definite statement about the linearizability of data structures. If a history has been proven to be linearizable, there still might be a possible history produced by the implementation which would be not. On the other hand, if a history has been proven to be false, the implementation is definitely incorrect and has to be checked for bugs.

Nevertheless, as mentioned before all verification methods lack some aspect. Up to now, none of them can give a 100% reliable answer about linearizability. Hence, it is a trade off between generality of the verification proof and realizability of the verifier implementation. Since the goal of this thesis is to prove very complex data structures for today's applications, this testing approach as a verification methodology is chosen.

3.4 Summary

In this chapter an overview of the area of correctness reasoning on concurrent programs has been given. It has been explained which correctness fields exist in the sense of which aspects of a program can be checked and it has been discussed why it can be useful to check the output of running programs for correctness. Then, the different correctness criteria that are candidates for checking the output of parallel programs were introduced. The decision for this work is to investigate linearizability as a widely accepted correctness criterion. Finally, different methods for verifying concurrent programs against linearizability have been explained. The approach of simulation and testing of programs as the verification technique is used in this work.

The next chapter explains the general technique of the method introduced in this document. It provides the necessary formal treatment for the methodology.

4. Methodology

This chapter gives a formal description of the execution trace verification method against linearizability. The goal is to have a general description of how the execution of arbitrary programs can be tested against this property. The claim is to prove the correctness of the output of a program but not of a program or algorithm itself.

Mainly basic set theory is used for managing states, and logical expressions for describing effects and invariants. The method assumes that the verified implementation produces a complete trace of all relevant operations during an execution. The target is to prove that a total order of the traced operations exists which could have been the order of a sequential execution respecting the program's sequential invariants. Furthermore, the total order shall not violate the partial order of start and end timestamps of the tracked operations in the sense that an operation that has ended before another operation started has to precede also this operation in the new order. As mentioned in the previous chapter, these conditions define the linearizability property of a history.

It is important to understand that if two operations o_1 and o_2 are executed in parallel, there is no definite answer whether actually o_1 or o_2 has taken effect first. Since these are parallel operations, both assumptions are valid. If for some reason the specification does not allow for example o_1 to have taken effect first, we may discard this assumption and simply assume that the other order was executed in fact. However, if both orders are wrong according to the specification, then we have an inconsistency in the history. Finding an answer whether a correct order of all operations exist is the goal of this verification method.

This chapter starts with an outline of the entire methodology for having a first impression and a preview of what will follow later. Then, it explains the basic program specification that is necessary for a verification to be accomplishable. This specification gives a first indication for different possible views to a program's state. After that, the format of a history is described and the term *metastate* is introduced. Then, a proof of the validity of the verification method is given which is the core part of the thesis. After extending the concept of metastates and their representation, it is shown how systems can be specified in an advanced manner by exploiting the knowledge about metastates. The proof and the extension of the metastate concept are the main contributions of this thesis. At last, some final thoughts about special cases during the verification procedure are mentioned.

4.1 Outline of the Execution Trace Verification Method

This section describes the general verification procedure. Figure 4.1 gives an overview of the verification steps. For now, the details of the method are omitted and only the rough approach is explained. The details for every step are explained later in this chapter.

First of all, a specification S of the sequential behavior of the program is needed. Second, there is a parallel program P that produces a history H. The history H is to be verified against the specification S against linearizability. If the verification of H fails, it can be definitely stated that there is an error in P. This forms the first step in Figure 4.1.

The history H consists of a set of logged operations $(l_1, l_2, ..., l_n)$ that have been generated during a test or simulation run of P. Each logged operation in H starts and ends at a certain point in time.

The assumption is that all operations take effect atomically, and that the time of taking effect (also called the linearization point) is between an operation's start time and its end time. Another assumption is that the history is complete, meaning that all operations that have started also have ended and that

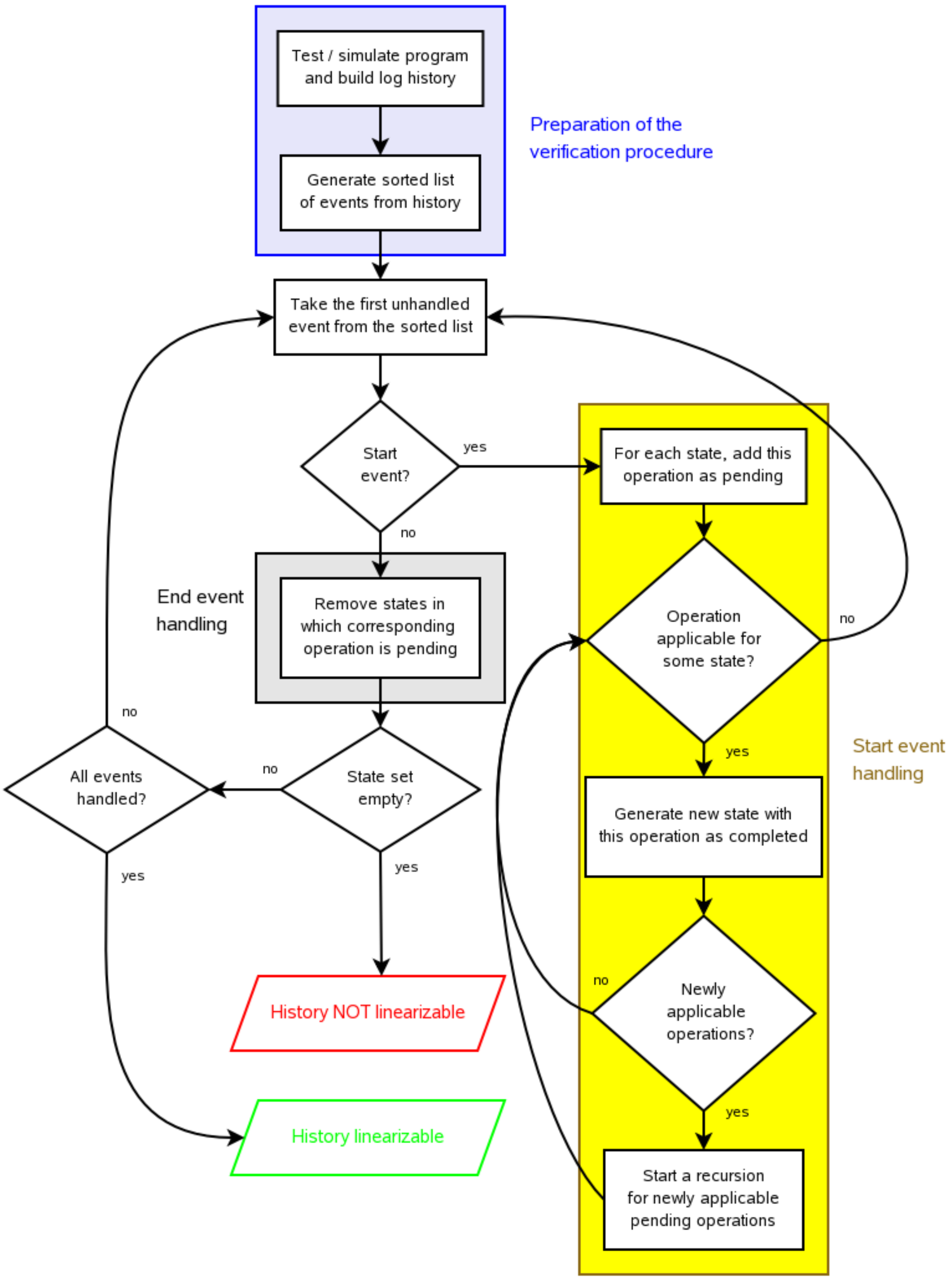

Test / simulate program and build log history
Generate sorted list of events from history
Preparation of the verification procedure
Take the first unhandled event from the sorted list
Start event?
yes
no
For each state, add this operation as pending
Operation applicable for some state?
no
yes
End event handling
Remove states in which corresponding operation is pending
State set empty?
no
yes
All events handled?
no
yes
Start event handling
Generate new state with this operation as completed
Newly applicable operations?
no
yes
Start a recursion for newly applicable pending operations
History NOT linearizable
History linearizable

Figure 4.1: Flow chart of the verification method

all operations of a test run or simulation run have been tracked. These assumptions state that the simulation run has been traced properly. Throughout the document, the term *simulation run* will be omitted and merely the term *test run* is used instead.

As it is not known when the effect of an operation really occurred, all possible orderings of overlapping operations have to be tried and checked on their validity concerning the invariants of the implementation. Therefore, a set of possible system states Σ is managed. A state contains the configuration of all information in a program at a certain point in time. A sequence of events $(e_1, e_2, ..., e_{2n})$, where $n = |H|$, gained from the trace of operations H is generated which is the second step in Figure 4.1. These events are either the start or the end event of an operation. Hence, for a single operation there exist exactly two events in the sequence. The events are sorted in a sequence according to their time of occurrence.

For each event e_i, an event function f_{e_i} is defined that modifies a state σ in the state set Σ according to the specification S of its corresponding operation. For example, one possible modification of a function f_{e_i} might be that if it becomes clear that a state σ can no longer be valid because of the event e_i, the event function of the corresponding operation removes that state from the state set Σ. Collapsing states occurs during the handling of end events.

On the other hand, the expansion of state sets occurs during the handling of start events. Since start events allow for more potential states, the event function possibly increases the managed state set Σ. Starting an event adds one more parallel operation to the currently considered execution. This may cause the necessity of recursions because it increases the number of possible orderings for the currently considered parallel operations. This recursion avoids the later necessity for backtracking because all possible system states are kept at a time.

After applying all event functions $f_{e_1}, ..., f_{e_{2n}}$, there has to be at least one state left in Σ, indicating that there is a valid total order for the operations respecting the previously mentioned time constraints. If there is no state left, the conclusion is that no such total order exists and thus H is not linearizable according to the specification S.

4.2 Basic Program Specification

In this thesis, a program is specified by defining the expected behavior of the operations provided by its interface. Thus, for the verification process no implementation details are of relevance. For example, an implementation of a list might allow two accesses to the data structure, *insert* and *remove*. The specification of the list says that only elements can be removed that have been inserted before. This information is already enough to perform a verification on the behavior of the implementation. It is not necessary to know whether an implementation of a simply locked list or hand-crafted lock-free list or some sorting for access optimization is used. Consequently, it is enough to specify the operations that are provided by the interface of the data structure for a complete description of the sequential invariants.

4.2.1 Abstract Operation Specification

In the following, the priority queue will be used as an example for a specification. An operation specification O_{Spec} consists of a 3-tuple

$$O_{Spec} = (\overrightarrow{P}, C, E). \tag{4.1}$$

where

$\overrightarrow{P}$ the vector of parameters

C the precondition which is a logical expression that has to hold at the point in time when the operation takes effect

E the effect is a state transition that describes the changes made by an operation on a state

The vector $\overrightarrow{P} = (p_1, p_2, ..., p_n)$ is a vector of parameters which are 2-tuples

$$p = (T, V) \tag{4.2}$$

where

T the type of the parameter

V the variable of the parameter

Note that the return value of an operation is treated as a parameter since it also consists of a type and an unknown value before runtime and thus is expressed by a variable.

4.2.2 Difference Between Value and Object Consideration

Note that in the operation specification examples that will be now introduced all operation specifications give merely the values of priorities they insert, remove or change, but not objects. It is important to understand the difference for the principle of the method. If, for example, the parameter vector of the *remove* specification contained the member (*Integer i*) instead of (*int i*), then it would have been exactly known which object is removed. However, in general this is not the case.

A data structure might very well contain only values instead of objects with multiple occurrences of the value like for example a list that contains two values of x which as primitive values cannot be differentiated. However, if for example two concurrent threads are inserting the same value, it is important for the verification process to distinguish the values when a third concurrent operation removes such a value. Hence, for the verification process all inserted values have to be wrapped into objects which is a meta-construct to make a distinction of multiple occurrences of the same value.

This is also a reason, why it makes sense to work on an operation level which is introduced later. Considering operations instead of values is an implicit object meta-construct for values as will be shown.

4.2.3 Example: Specifying an Insertion

The *insert* operation for a list or a priority queue Q might be defined as in the following

$$\begin{aligned} & \mathit{insert}(\mathit{int}\ i, \mathit{Set}\ Q) && && (4.3) \\ & \quad := ((\mathit{int}\ i, \mathit{Set}\ Q), && \text{(parameters)} \\ & \qquad \mathit{TRUE}, && \text{(precondition)} \\ & \qquad Q \to Q \cup \{\mathit{new}\ \mathit{Integer}(i)\}) && \text{(effect)} \end{aligned}$$

This equation shows a formal definition of an *insert* operation according to the formalism introduced before. The operation is named *insert* and takes two parameters, a value i of type *int* and a *Set Q*. *Integer* objects and *int* values are distinguished because a priority queue may contain the same priority multiple times but these priority values still have to be distinguished. Using Java's notion of objects is a well-known convention and useful for this purpose.

The precondition is always *TRUE* because there are no restrictions of when an *insert* operation may take effect. The action or the effect of the operation is that it creates an object - here for simplicity an *Integer* object - with priority i. Further, the *Set Q* is transfered into a new state where Q contains the *Integer* o, now.

It is convenient for the description to use the closed-world assumption [BrLi89]. Generally speaking, this assumption states that everything that is not known to be true is assumed to be false. Consequently, if an operation performs any action that is not defined in its specification, then this behavior is considered to be wrong. The opposite of this assumption is the open-world assumption. It makes sense to use the closed-world assumption because otherwise for the *effect* there would be a description necessary that all elements that have been in Q are still in Q after performing an *insert* operation. Such a detailed specification makes the entire matter less understandable and hence less practicable which is unnecessary for such an intuitive issue.

It is often the case that the implementation of a certain data structure is verified instead of a system consisting of different data structures. Even if a system of different data structures is considered, often, certain variables are used repeatedly during the lifetime of the program. So, if a priority queue

Table 4.1: Types and their initial values

Type	Initial Value
int	0
Integer	null
string	””
String	null
boolean	*TRUE*
Boolean	null
Set	$\emptyset$
Object	null

implementation is specified and only one queue is used, then the queue itself can be defined as a global variable. Hence, the variable Q can be denoted as global and thus the specification of the insert operation becomes simpler because Q is no longer a parameter

$$
\begin{aligned}
&\mathit{insert}(\mathit{int}\ i) \\
&:= ((\mathit{int}\ i), && \text{(parameters)}\\
&\quad \mathit{TRUE}, && \text{(precondition)}\\
&\quad Q \rightarrow Q \cup \{\mathit{new\ Integer}(i)\}) && \text{(effect)}
\end{aligned} \tag{4.4}
$$

4.2.4 Defining an Initial State

Now, since an operation has been specified, there has to be a specification of the initial state of the system. In this example this is a state of Q before any operation has been executed. Here, we say $Q = \emptyset$, initially. For any *Set* an empty set as the initial state is natural.

The convenience about assuming a *Set* to be initially empty is that the state of Q can be entirely inferred from the operations performed on the data structure. If five *insert* operations have been executed, Q will contain exactly five elements. A priority queue might still contain initial elements. If this is the case before the actual test run starts, each produced history is added a prefix of insertions. The timestamps of the insertions have to indicate that they have taken effect before the actual first operation of the test run has started. For all variable types, the default initial values are listed in table 4.1.

4.2.5 Example: Specifying a Removal of an Object

The following specification example of the *remove* operation shows how a real precondition is expressed. Again, Q is assumed to be global and thus is not treated as a parameter. The *remove* operation of a priority queue removes the element of highest priority in the queue. It takes no parameters but it returns the priority of the object removed.

$$\begin{aligned}
&\mathit{remove}() && (4.5)\\
&\quad := ((\mathit{int}\ i), && \text{(parameters)}\\
&\qquad \forall q \in Q : \mathit{IntVal}(q) \leq i \wedge \exists o \in Q : \mathit{IntVal}(o) = i, && \text{(precondition)}\\
&\qquad Q \rightarrow Q \setminus \{o\}) && \text{(effect)}.
\end{aligned}$$

The precondition of the operation specification defines that the *remove* operation only may remove an object that exists in Q and is of highest priority. The effect is that the removed element is no longer in Q.

4.2.6 Example: Specifying a Modification

Many operation do not perform a generation or removal of elements but a modification of existing elements. The *changeKey* operation is such an example that changes the priority of an existing element in the priority queue.

$$\begin{aligned}
&\mathit{changeKey}(\mathit{int}\ i_1, \mathit{int}\ i_2) && (4.6)\\
&\quad := ((\mathit{int}\ i_1, \mathit{int}\ i_2), && \text{(parameters)}\\
&\qquad \exists q \in Q : \mathit{IntVal}(q) = i_1, && \text{(precondition)}\\
&\qquad \mathit{IntVal}(q) \rightarrow i_2) && \text{(effect)}.
\end{aligned}$$

4.2.7 Abstract Program Specification

Specifying an initial state and all operations executed by the program is enough information for the verification method to execute a proof on a given history. So the program specification S of a program P is in general

$$S = (\Pi, \Omega, \Xi) \tag{4.7}$$

where

Π the set of global variables

Ω the set of operation specifications

Ξ the prefix for all generated histories

A complete specification of the priority queue example could have following appearance

$$\begin{aligned} &\mathit{Prio\,Queue}_{\mathit{Spec}} \\ &\quad = (\{\mathit{Set}\ Q\}, \{\mathit{insert}(\mathit{int}\ i), \mathit{remove}(), \mathit{changeKey}(\mathit{int}\ i_1, \mathit{int}\ i_2)\}, \emptyset) \end{aligned} \tag{4.8}$$

with the three operations being specified as before and the history prefix being empty and thus implicitly specifying that the *Set Q* does not contain any elements initially.

4.3 History

As shown in Figure 4.1 the first step of the verification process is to produce a log of executed operations denoted as *history.* A log entry l of an operation in the history is a 4-tuple

$$l = (o, t_{start}, t_{end}, \overrightarrow{v}). \tag{4.9}$$

where

o the identifier of the operation which must have a specification in Ω

t_{start} the time when the operation started

t_{end} the time when the operation ended

$\overrightarrow{v}$ the value vector of the operation that corresponds type-wise to the parameter vector $\overrightarrow{P}$ of the operation specification O_{Spec} of o

The timestamps t_{start} and t_{end} are unique within a history and consequently defining a total order for all events. It is obvious that $t_{start} < t_{end}$. The method's assumption is that operations take effect atomically at some point in time between its two timestamps which is unknown. Now, the task is to assign to each operation a time when it might have taken effect. This assignment has to result in a total order on the operations that is valid for a sequential execution of the program. If such a total order is found, the conclusion is that the execution has been performed in a linearizable manner and thus has produced correct output.

A history H is a set of log entries $(l_1, l_2, ..., l_n)$ that have been produced during a test run. This history is a set of partially ordered elements with respect to their time of occurrence since operations may occur strictly before, strictly after, or overlap with other operations. This history implies a sequence of totally ordered events $(e_1, e_2, ..., e_{2n})$ where e is a pair

$$e = (t, l) \tag{4.10}$$

where

t the time of the event

l the event's log entry in the history

t is either the start or the end time of l. All log entries exist twice in the sorted list of events. The first and second occurrence of an entry is at its start and end time, respectively. Since events are unique in the sequence, we define the set $H_e = \{e_i\}_{i=1}^{2n}$ that contains all elements of the sequence.

Without loss of generality, we can assume for two events

$$(e_i = (t_i, l_x) \in H_e \wedge e_j = (t_j, l_y) \in H_e) \Longrightarrow i < j \Leftrightarrow t_i < t_j. \tag{4.11}$$

With this assumption it also follows that e_1 must always be a start event and that a start event has a smaller index than an end event of the same log entry. The event sequence corresponds to the history definition of Herlihy [HeWi90] with one exception. Whereas in Herlihy's definition of a history it is not known what an operation is going to perform exactly at start time, here, we already know which values are going to be manipulated in which way. If we have in addition

$$\begin{aligned} &\forall i \in \{j \in \mathbb{N} | j \bmod 2 = 1 \wedge j \leq |H_e|\} : \\ &\qquad (e_i = (t_i, l_x) \wedge e_{i+1} = (t_{i+1}, l_y)) \Rightarrow l_x = l_y \end{aligned} \tag{4.12}$$

then H is a sequential history. Equation 4.12 states that in the sequence H_e the successor of a start event is always the end event of the same log entry.

4.4 Metastate

Usually a verification is performed on the actual system's state. However, there are practical problems in the general case which are explained in the following. For avoiding those practical problems the view on the state is raised to a *metastate* level.

4.4.1 Common State

The common term *state* denotes the configuration of all information in a program at a certain point in time. The information is stored as values in registers or memory addresses. From an abstract point of view, all relevant information can be stored in variables.

Table 4.2 shows a history of two *insert* operations. The changes of the system's state during the execution of the operations in Table 4.2 are shown in Table 4.3.

Table 4.2: A history of two *insert* operations

Log ID	Operation	Start	End	Value
0	*insert*	0	2	1
1	*insert*	1	3	2

Table 4.3: State changes during the execution of the operations in Table 4.2

Time	Possible queue states
-1	{[]}
0	{[] , [1]}
1	{[] , [1] , [2] , [1,2]}
2	{[1] , [1,2]}
3	{[1,2]}

At time -1 before the first operation started, the queue is empty. After the start of the first operation at time 0, there are two possible states for the queue. Either the queue might be empty or it might contain one element. An operation that has started but not ended yet is called *pending*. At time 1 when the second operation starts, there are already four possible states. The queue might be empty, contain either the first or the second element, or contain both elements. When operations end, the number of possible states decreases because the inserted elements are definitely in the queue now.

The problem with considering the possible actual states of the system is that the number of possible states increases exponentially with the number of pending operations. With 3 pending operations, there are already 8 possible states. With n pending operations, there are 2^n possible states. This leads to the typical state explosion problem. Therefore, this verification method does not consider the actual possible states of a system but introduces a *metastate* construct.

4.4.2 From States to Metastates and Back

Consider following three statements

- The queue Q contains the elements x, y and z.

- There have been exactly three *insert* operations executed that inserted the elements x, y, and z in Q, respectively.
- There have been exactly four *insert* operations executed that inserted the elements w, x, y, and z in Q, respectively and a *remove* operation that removed the element w.

From the second and third statement, the first statement can be inferred. However, this is not true for the other direction. The first statement describes the actual state of the queue Q whereas the other two statements describe which operations have been executed on the queue. For every state, usually there exists multiple possible operation sequences for reaching this state.

For the set of operations executed at a certain point in time the new term *metastate* is introduced. The *metastate* denotes a higher level state from which the actual state of the system can be inferred. Thus the lower level state which is the common state could be denoted as the *first-level state* whereas the metastate could be denoted as the *second-level state* of a system. For disambiguation throughout this document the term *metastate* will always be used instead of *second-level state* while the term *first-level state* will be used when referring to common states.

There are two reasons why the *metastate* is an important construct for this method. Firstly, it is an object meta-construct as already mentioned in section 4.2. Furthermore, there are possibilities to compress sets of possible states by considering equivalence classes which is of practical usability when tackling the state explosion problem as will be shown later.

4.4.3 Basic Structure

A *metastate* M is a pair consisting of a sequence and a set

$$M = (O_C, O_P) \tag{4.13}$$

where

O_C the sequence of completed operations

Table 4.4: Metastate changes during the execution of the operations in Table 4.2

Time	Metastates
-1	$\{((),\emptyset)\}$
0	$\{((),\{l_0\})\ ,\ ((l_0),\emptyset)\}$
1	$\{((),\{l_0,l_1\})\ ,\ ((l_0),\{l_1\})\ ,\ ((l_1),\{l_0\})\ ,\ ((l_0,l_1),\emptyset)\ ,\ ((l_1,l_0),\emptyset)\}$
2	$\{((l_0),\{l_1\})\ ,\ ((l_0,l_1),\emptyset)\ ,\ ((l_1,l_0),\emptyset)\}$
3	$\{((l_0,l_1),\ \emptyset)\ ,\ ((l_1,l_0),\emptyset)\}$

O_P the set of pending operations

An operation that has started but not yet finished is an element of O_P. This operation is ready to take effect at any point in time if its precondition is true. An operation that is definitely completed is an element of O_C. This operation has already taken place and changed the system's state according to its effect. The order in the sequence of the operations in the O_C component defines the order of the linearization points of the operations.

Going back to the history example all states in Table 4.3 can be inferred by the corresponding metastates listed in Table 4.4. Each set of metastates forms a *metastate description* of the currently possible first-level states. Note that in the table the log ID's of the operations are put into the components O_C and O_P of the metastates as representatives of the operations. When considering metastates all possible orderings of operations have to be taken into account. Therefore, there are more metastates than actual first-level states. However, the number of metastates can be drastically reduced by taking simple properties of operations into account as will be shown in section 4.6.

4.5 Proving Linearizability

The previous section introduced the metastate description of states that are formed by a history. This section leads to the proof of whether a history is linearizable or not and forms the core part and together with the next section the main contributions of this thesis.

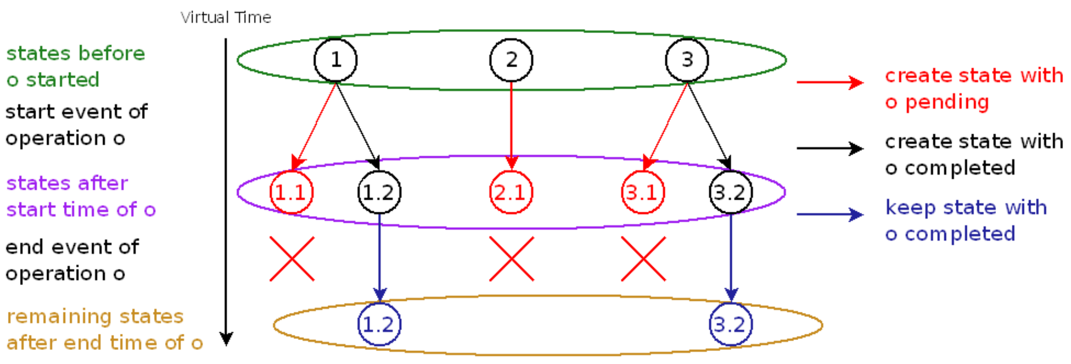

Virtual Time
states before o started
start event of operation o
states after start time of o
end event of operation o
remaining states after end time of o
1
2
3
1.1
1.2
2.1
3.1
3.2
1.2
3.2
create state with o pending
create state with o completed
keep state with o completed

Figure 4.2: General idea of the effect of event functions

4.5.1 Basic Idea

The term *event* has been introduced in section 4.3. As already mentioned, an operation implies two events, one at its start and one at its end.

The basic idea is that when an operation starts, there are potentially more alternative states for the system because the operation might take effect but also might not take effect immediately. We cannot know because we only have a history that tells us when an operation started and definitely ended but not when it took effect. If two operations are overlapping, both orders are valid assumptions if there is no violation of correctness. Our goal is to derive from what we have whether the history is correct according to the linearizability property by finding a valid sequence of operations.

When an operation ends, the information that this operation must have taken effect is available. This means that only states in which this operation has taken effect are still valid. If it has not for a certain state, the state is invalid because this means that the operation has not taken effect during any time since its start time.

Figure 4.2 illustrates the general idea of event function effects. First an operation starts which leads to an expansion of the states due to increasing alternatives. Then the operation ends and all states where this operation is still assumed to be pending are discarded. Only the remaining states are considered for future processing.

4.5.2 Event Functions

Each operation specification has an *effect* component that describes the state transition that the operation applies. The previous section has shown that the state of a system can be represented by a description operating on metastates.

The entire verification procedure operates on events since it is unknown when the actual linearization point of an operation is reached. Therefore, event functions are defined that are applied on metastates. Those functions systematically cover all possible orders of linearization points and prove whether they break the system's invariant or not.

As introduced before, a metastate consists of two components. The first component is the sequence of completed operations O_C. The second component is a set O_P of pending operations that have started but not ended yet. An event function

$$f_e : \Gamma_S \longrightarrow \mathcal{P}(\Gamma_S) \tag{4.14}$$

for an event e is a function from the domain of all metastates Γ_S reachable in a specification S to the powerset of Γ_S. Hence, an event function maps a metastate to a set of metastates. An end event function f_{endX} for an operation X on a metastate (O_C, O_P) can be simply defined as

$$f_{endX}((O_C, O_P)) = \begin{cases} \emptyset & X \in O_P \\ \{(O_C, O_P)\} & X \in O_C \end{cases} \tag{4.15}$$

meaning that a metastate is discarded if operation X could not take effect and kept if it could.

Defining the start event is more complicated. First of all we need a help function $h : \Gamma_S \longrightarrow \mathcal{P}(\Gamma_S)$

$$h((O_C, O_P)) = \{((O_C o), O_P \setminus \{o\}) | o \in O_P \wedge C(o)|_{O_C} = \; true\}, \tag{4.16}$$

where

$C(.)|_{O_C}$ is the condition in the first-level state inferred from the O_C component of the metastate.

From a given metastate M, h provides all metastates where exactly one of the pending operations takes effect if applicable in M. The resulting metastates respect the same operations but with one element shifted from O_P to O_C. Note that this function provides the empty set if there is no shift applicable. With this help function h, a recursive function *Rec*: $\mathcal{P}(\Gamma_S) \longrightarrow \mathcal{P}(\Gamma_S)$ can be defined

$$\mathit{Rec}(\{M_1, M_2, ..., M_n\}) = \bigcup_{i=1}^{n} (\mathit{Rec}(h(M_i)) \cup h(M_i)). \tag{4.17}$$

This function defines a recursion that yields all possible correct metastates by shifting applicable operations from the pending component to the completed component for each metastate. Note that *Rec* is a finite recursion because O_P is a finite set. With the *Rec* function, we can now define the start event function of an operation X:

$$f_{\mathit{startX}}((O_C, O_P)) = \begin{cases} \{(O_C, O_P \cup \{X\})\}, & C(X)|_{O_C} = \mathit{false} \\ \\ \{(O_C, O_P \cup \{X\}), & \\ ((O_C X), O_P)\} & \text{else.} \\ \cup\ \mathit{Rec}(((O_C X), O_P)) & \end{cases} \tag{4.18}$$

The function defines that a starting operation that is not yet applicable remains pending. If it is applicable then it might remain pending or take effect. If it takes effect, it must be checked whether other also pending operations might take effect now since the metastate and consequently the system state have changed.

4.5.3 Extension of Event Functions

Since the verification procedure operates on a metastate description rather than on a single metastate, event functions can be extended to operate on a finite set of metastates $\{M_1, M_2, ..., M_n\}$ instead of on a single one. This can be done by simply overloading the notation of a given event f and defining that

$$f(\{M_1, M_2, ..., M_n\}) := \bigcup_{i=1}^{n} f(M_i) \tag{4.19}$$

This is a trivial extension for an event function to a function $\mathcal{P}(\Gamma_S) \longrightarrow \mathcal{P}(\Gamma_S)$.

4.5.4 The History Verification Theorem

The idea of the verification procedure is that a history of events can be traversed linearly without backtracking such that at some point in time a decision can be made whether this history is linearizable. The following theorem forms the basis of the execution trace verification method introduced in this dissertation.

Theorem: Let S be a specification and $H_e = \{e_i\}_{i=1}^{n}$ a complete history of events of operations specified in S. For each e_i let f_{e_i} be the corresponding event function and i be the position in the history defined by the event's time of occurrence. Then

$$(f_{e_{2n}} \circ f_{e_{2n-1}} \circ ... \circ f_{e_1})(\{((), \emptyset)\}) \neq \emptyset \Longleftrightarrow H_e \text{ is linearizable} \tag{4.20}$$

Proof: First, the direction "$\Longleftarrow$" will be proven. So, assume H_e is linearizable. Let i_H be the index function for an event in the history H. Then, according to the definition of linearizability a permutation of events p exists, such that

1. $H_p = (p(e_1), ..., p(e_{2n}))$ is a sequential history
2. for an end event e_{end} and a start event e_{start} it holds $i_{H_e}(e_{end}) < i_{H_e}(e_{start}) \Longrightarrow i_{H_p}(p(e_{end})) < i_{H_p}(p(e_{start}))$
3. for all operations o corresponding to the events in H_p, $C_{H_p}(o) =$ *true*, where C_{H_p} is the condition of an operation in H_p right before the operation's linearization point

For simplicity, define $e_{i_k} := p(e_i)$, where $i_{H_p}(p(e_i)) = k$. Let $(o_1, ..., o_n)$ be the non-ambiguous sequence of operations corresponding to H_p. Now, it will be proven that the metastate $((o_1, ..., o_n), \emptyset)$ is an element of $(f_{e_{2n}} \circ f_{e_{2n-1}} \circ ... \circ f_{e_1})(\{((), \emptyset)\})$.

The first element in H_p is per definition e_{i_1} which is a start event according to property 1 of permutation p. Due to property 2, there is no end event e_j, where $j < i_1$. Consequently, if $i_1 > 1$, all events that precede e_{i_1} in H_e are start events. If x_j is the operation corresponding to event e_j then there must be a metastate $((), \{x_j | j < i_1\})$ in $(f_{e_{i_1-1}} \circ ... \circ f_{e_1})(\{((), \emptyset)\})$ according to the definition of the start event function. Per definition o_1 is the corresponding operation to e_{i_1}. As this metastate describes the initial state σ_{init} and $C(o_1)|_{\sigma_{init}} = true$ due to property 3, the function $f_{e_{i_1}}$ which is a start event function creates a metastate $((o_1), \{x_j | j < i_1\})$. If e_{i_2} is the next end event in H_e before any other end event, obviously, its function does not erase all metastates from the metastate description because of the definition of end event functions and because o_1 is in the O_C component of at least one of the metastates.

Now, assume that for some m there is a metastate $((o_1, ..., o_m), X)$ for some set of operations X in the metastate description $(f_{e_{i_{2m+1}-1}} \circ ... \circ f_{e_1})(\{((), \emptyset)\})$. Per definition $e_{i_{2m+1}}$ is the start event of the operation o_{m+1}. Due to property 2, the only end event that may occur in H_e before $e_{i_{2m+1}}$ are $e_{i_2}, e_{i_4}, ..., e_{i_{2m}}$. However, according to the assumption, they have not erased the metastate $((o_1, ..., o_m), X)$. Let σ_m be the state inferred from this metastate, then according to property 3 $C(o_{m+1})|_{\sigma_m} = true$. Consequently, there is a metastate $((o_1, ..., o_m, o_{m+1}), X)$ in $(f_{e_{i_{2m+1}}} \circ ... \circ f_{e_1})(\{((), \emptyset)\})$. If any of the end events of the operations $o_1, ..., o_{m+1}$ takes effect after $e_{i_{2m+1}}$ before any other end event, they will not erase this newly created metastate because their corresponding operation is in the O_C component.

$\Longrightarrow ((o_1, ..., o_n), \emptyset) \in (f_{e_{2n}} \circ f_{e_{2n-1}} \circ ... \circ f_{e_1})(\{((), \emptyset)\})$

$\Longrightarrow$ if H_e is linearizable $\Rightarrow (f_{e_{2n}} \circ f_{e_{2n-1}} \circ ... \circ f_{e_1})(\{((), \emptyset)\}) \neq \emptyset$

Now, we prove the direction "$\Longrightarrow$". The goal is to show that a permutation p exists that fulfills the three linearizability properties. Assume $(f_{e_{2n}} \circ f_{e_{2n-1}} \circ ... \circ f_{e_1})(\{((), \emptyset)\}) \neq \emptyset$.

First of all, note that per definition a start event function never creates a metastate with a shorter O_C component than the existent metastates in the description before the application of the function. Second, note that after the application of an end event function only metastates are remaining in the metastate description that contain the corresponding operation in the O_C component. Consequently, an element in $(f_{e_{2n}} \circ f_{e_{2n-1}} \circ ... \circ f_{e_1})(\{((), \emptyset)\})$ must contain all operations that have started and ended in H_e. Let $(o_1, ..., o_n)$ be a sequence of the O_C component of one of the elements. Then, a permutation p can be defined such that $i_{H_p}(p(e_i)) = 2 \cdot k - 1$, if e_i is a start event of o_k, and $i_{H_p}(p(e_i)) = 2 \cdot k$, if e_i is an end event of o_k. Hence, the history H_p defined by p is a sequential history fulfilling property 1.

Let e_{end} be an end event of some operation o and e_{start} the start event of another operation q in H_e. Further, let $i_{H_e}(e_{end}) < i_{H_e}(e_{start})$. Then, the event function $f_{e_{end}}$ is applied before the event function $f_{e_{start}}$. Since the end event function only keeps metastates in which the corresponding operation is in the O_C component already, and start event functions only append operations to existing O_C components, a metastate cannot have an O_C component, where q occurs before $o \Rightarrow i_{H_p}(p(e_{end})) < i_{H_p}(p(e_{start}))$ $\Rightarrow$ property 2 is fulfilled by p.

Due to the definition of a start event function, only operations are put into the O_C component of a metastate if their condition is *true*. Consequently, for all operations in the sequence $(o_1, ..., o_n)$ the condition has been *true* during their insertion into O_C. $\Rightarrow$ property 3 is fulfilled by p.

$\Longrightarrow (f_{e_{2n}} \circ f_{e_{2n-1}} \circ ... \circ f_{e_1})(\{((), \emptyset)\}) \neq \emptyset \Rightarrow$ a permutation p exists fulfilling all linearizability properties. $\Box$

This Theorem forms the basis of the verification methodology introduced in this dissertation. With the event functions defined as introduced, an arbitrary history can be verified against some specification as described in the sections 4.2 and 4.7.

4.6 Metastate Representation

In section 4.4 metastates have been introduced. However, in this form, the representation by metastates is even more space consuming than simply considering first-level states. This section shows how the space requirement can be effectively reduced so that this method becomes practical.

4.6.1 Permutability of Operations

For the further use of the concept of metastates, following terms are introduced

Operation type Two operations o_1 and o_2 that are specified by the same operation specification are called to be of the same *type*.

Configuration A configuration is a pair of a sequence of operations $(o_1, ..., o_n)$ and a first-level state σ. A configuration can be interpreted as a state from which the sequence of operations is executed. If $n = 0$, then the configuration can be identified with the state σ. This is also called a *trivial configuration*. If $n = 1$, then this is called a *primitive configuration*.

Effect The effect of an operation o according to its specification as introduced before is a function of a first-level state to a new state denoted as

$$\varepsilon(o) : x \to y \tag{4.21}$$

where

x is the first-level state before the effect is applied

y is the first-level state after the effect is applied

$((o_1, o_2, ..., o_n), \sigma)$ denotes the configuration of a sequence of operations on an initial first-level state σ. For this operation sequence the new state can be inferred by applying all effects of the operations in the same order for the initial state

$$((o_1, o_2, ..., o_n), \sigma) \longrightarrow ((), \varepsilon(o_n)(...(\varepsilon(o_2)(\varepsilon(o_1)(\sigma)))...)). \tag{4.22}$$

If $X = (o_1, o_2, ..., o_n)$ the short notation is

$$(X, \sigma) \longrightarrow ((), \varepsilon(X)(\sigma)). \tag{4.23}$$

Precondition The precondition of an operation o according to its specification as introduced before is denoted as $C(o)$. Note that in this document there is a semantic difference between $C(o) = \mathit{TRUE}$ and $C(o) = \mathit{true}$. In the first case, the specification of o defines that the precondition of the operation is always *TRUE*. In the second case, the precondition is not necessarily always *TRUE* by definition but the state of the system fulfills the precondition so that it can be executed. Consequently, $C(o) = \mathit{TRUE}$ is a special case of $C(o) = \mathit{true}$.

For a state σ, the condition of an operation with all its variables substituted by their values in S is denoted as $C(o)|_\sigma$

Validity A primitive configuration (o, σ) is called *valid*, if o may be executed in σ. Consequently, we say

$$(o, \sigma) \text{ is } \mathit{valid} \iff C(o)|_\sigma = \mathit{true}. \tag{4.24}$$

This concept can be extended to arbitrary configurations

$$\begin{aligned} &((o_1, o_2, ..., o_n), \sigma) \text{ is } \mathit{valid} \\ \iff &(o_1, \sigma) \text{ is } \mathit{valid} \wedge (o_2, \varepsilon(o_1)(\sigma)) \text{ is } \mathit{valid} \\ &\wedge ... \wedge (o_n, \varepsilon(o_{n-1})(...(\varepsilon(o_2)(\varepsilon(o_1)(\sigma)))...)) \text{ is } \mathit{valid} \end{aligned} \tag{4.25}$$

By definition, operations o can only be added to an O_C component of a metastate that evolved from an initial state σ if $((O_C o), \sigma)$ is *valid.*

Permutability An operation type is called *permutable* within a specification S if for any two operations o_1 and o_2 of the same type but potentially different parameter values, the following property holds

$$\begin{aligned} \forall \sigma \text{ reachable in } S : ((o_1, o_2), \sigma) \text{ is } \textit{valid} \ &\Leftrightarrow ((o_2, o_1), \sigma) \text{ is } \textit{valid} \\ \wedge\ \varepsilon(o_2)(\varepsilon(o_1)(\sigma)) &= \varepsilon(o_1)(\varepsilon(o_2)(\sigma)) \end{aligned} \tag{4.26}$$

In this case it can be said that o_1 is *permutable* with o_2 and vice versa.

The following Proposition can be deduced from the definitions introduced:

Proposition 1: Let σ be the initial state. A metastate $M = (O_C, O_P)$ with $O_C = (XYZ)$, where X and Z are arbitrary operation sequences and $Y = (o_1, ..., o_n)$ a sequence of permutable operations of the same type, represents the same state as any metastate $M' = (O'_C, O_P)$ with $O'_C = (XY'Z)$, where Y' is a permutation of Y.

Proof: $(XY, \sigma) \rightarrow (Y, \varepsilon(X)(\sigma)) \rightarrow ((), \varepsilon(Y)(\varepsilon(X)(\sigma)))$

$= ((), \varepsilon(Y')(\varepsilon(X)(\sigma))) \leftarrow (Y', \varepsilon(X)(\sigma)) \leftarrow (XY', \sigma)$

$\Longrightarrow \varepsilon(XYZ)(\sigma) = \varepsilon(XY'Z)(\sigma)\Box$

Corollary 1: The *insert* operation specified in Equation 4.4 is permutable within $\textit{PrioQueue}_{\textit{Spec}}$ in Equation 4.8.

Proof: Let $\textit{insert}(x)$ and $\textit{insert}(y)$ be operations inserting an element with value x and y, respectively. The validity of $((\textit{insert}(x),\ \textit{insert}(y)), \sigma)$ and $((\textit{insert}(y),\ \textit{insert}(x)), \sigma)$ for any σ is trivial since $C(\textit{insert}(\textit{int}\ i)) = \textit{TRUE}$. The second property for permutability follows from the commutativity of the union operator for sets that defines the effect of *insert* operations. $\Box$

Remark: *Remove* operations of distinct elements in a linked list can be specified in a way that they form another example for permutable operations.

4.6.2 Equivalence Classes

The permutability property of an operation allows reductions of a metastate description. It permits the representation of multiple metastates by one metastate as stated in Proposition 1.

A metastate can be considered as an embodiment of a sequential history that is represented by the O_C component. However, as seen before a set of metastates can be redundant. There are for example two metastate representations for one actual first-level state at time 4 in Table 4.4. Before continuing on the example, further terms have to be introduced for completing the formalism.

Equivalence For a specification S, two metastates M_1 and M_2 are called *equivalent* in S if the same set of first-level states are inferred from both metastates. The notation is

$$M_1 \sim_S M_2 \tag{4.27}$$

For simplicity, the symbol S can be omitted if the usage is clear.

Equivalence class If X_S is the set of all metastates over the specification S and $M \in X_S$ then the set

$$[M]_S = \{N \in X_S | N \sim_S M\} \tag{4.28}$$

forms an *equivalence class.*

Cardinality The *cardinality* $|M|$ of a metastate $M = (O_C, O_P)$ is defined as

$$|M| = |O_C| + |O_P| \tag{4.29}$$

Minimal element An element of an equivalence class is called *minimal* if its cardinality is minimal compared to all other elements in the class. A class may contain multiple *minimal* elements.

Table 4.5: Metastates from Table 4.2 for comparison

Time	Metastates
-1	$\{((),\emptyset)\}$
0	$\{((),\{l_0\})\ ,\ ((l_0),\emptyset)\}$
1	$\{((),\{l_0,l_1\})\ ,\ ((l_0),\{l_1\})\ ,\ ((l_1),\{l_0\})\ ,\ ((l_0,l_1),\emptyset)\ ,\ ((l_1,l_0),\emptyset)\}$
2	$\{((l_0),\{l_1\})\ ,\ ((l_0,l_1),\emptyset)\ ,\ ((l_1,l_0),\emptyset)\}$
3	$\{((l_0,l_1),\ \emptyset)\ ,\ ((l_1,l_0),\emptyset)\}$

Table 4.6: Reduced metastates from Table 4.5 due to permutability

Time	Metastates
-1	$\{((),\emptyset)\}$
0	$\{((),\{l_0\})\ ,\ ((l_0),\emptyset)\}$
1	$\{((),\{l_0,l_1\})\ ,\ ((l_0),\{l_1\})\ ,\ ((l_1),\{l_0\})\ ,\ ((l_0,l_1),\emptyset)\}$
2	$\{((l_0),\{l_1\})\ ,\ ((l_0,\ l_1),\emptyset)\}$
3	$\{((l_0,l_1),\emptyset)\}$

There is no value in knowing whether the one or the other permutable operation has taken effect before, because both possibilities lead to the same first-level state. Since equivalent metastates do not provide any valuable information for the verification process, from now on at most one representative of an equivalence class will be considered per metastate description. This leads to a reduced metastate description as in Table 4.6. Now, each metastate corresponds to exactly one state.

4.6.3 Implied Metastates

The fact that there are operations with their precondition always being *TRUE* offers the possibility of describing multiple first-level states by a single metastate. Therefore the following fact has to be considered.

Implication For two metastates M_1 and M_2, M_2 is called implied by M_1, if M_1 can never exist in a metastate description without M_2 being an element of the same metastate description. Hence, for any metastate description D, we have

$$M_1 \in D \Longrightarrow M_2 \in D \tag{4.30}$$

Following Lemmas and Proposition can be stated

Lemma 1: Let o be an operation with $C(o) = \mathit{TRUE}$ and $M_1 = (X, Y \cup \{o\}), M_2 = ((X, o), Y)$ be metastates with X being an arbitrary operation sequence and Y an arbitrary set of pending operations. Then $M_1 \Rightarrow M_2$.

Proof: Whenever there is a metastate description containing M_1, it must contain M_2 since o is ready to take effect at any time due to its precondition. $\square$

Lemma 2: Let Z be a set of permutable operations of the same type with $\forall z \in Z : C(z) = \mathit{TRUE}$, X being an arbitrary operation sequence, and Y an arbitrary set of pending operations. Then

$$(X, Y \cup Z) \Rightarrow ((X, s(W)), Y \cup (Z \setminus W)), \forall W \subseteq Z \wedge \forall s(W) \tag{4.31}$$

where

$s(W)$ is a sequences of elements of W

Proof: Follows recursively from the permutability of the operations in Z and Lemma 1. $\square$

Lemma 3: With the same setting as in Lemma 1, $M_2 \not\Rightarrow M_1$.

Proof: Follows by the counterexample in Table 4.6.

Proposition 2: Let $Z_1, Z_2, ..., Z_n$ each be a set of permutable operations of the same type within the set but of different type across the sets, $\forall z \in Z = \bigcup_{i=1}^{n} Z_i : C(z) = \mathit{TRUE}$, X being an arbitrary operation sequence, and Y an arbitrary set of pending operations. Then

$$(X, Y \cup Z) \Rightarrow ((X, p(s_1(W_1)...s_n(W_n))), Y \cup (\bigcup_{i=1}^{n} Z_i \setminus \bigcup_{i=1}^{n} W_i)), \tag{4.32}$$
$$\forall W_i \subseteq Z_i \wedge \forall p(.)$$

Table 4.7: Metastates from Table 4.6 for comparison

Time	Metastates
-1	$\{((),\emptyset)\}$
0	$\{((),\{l_0\})$, $((l_0),\emptyset)\}$
1	$\{((),\{l_0,l_1\})$, $((l_0),\{l_1\})$, $((l_1),\{l_0\})$, $((l_0,l_1),\emptyset)\}$
2	$\{((l_0),\{l_1\})$, $((l_0, l_1),\emptyset)\}$
3	$\{((l_0,l_1),\emptyset)\}$

Table 4.8: Further reduced metastates from Table 4.7 due to implied metastates

Time	Metastates
-1	$\{((),\emptyset)\}$
0	$\{((),\{l_0\})\}$
1	$\{((),\{l_0,l_1\})\}$
2	$\{((l_0),\{l_1\})\}$
3	$\{((l_0,l_1),\emptyset)\}$

where

$s_i(W_i)$ is a sequences of elements of W_i

$p(.)$ is a permutation of sequences

Proof: Follows recursively from Lemma 2. □

The consequence of Proposition 2 is that it is unnecessary to treat metastates that are implied by other metastates explicitly because their existence is definite. Hence, it is sufficient to consider only the implying metastates keeping in mind that the other metastates are covered, too.

In the example this means that there is only one metastate left as shown in Table 4.8. From each single metastate at each point in time all first-level states from Table 4.3 can be inferred by taking equivalent and implied metastates into consideration. The advantage from a computational point of view is that there is much less concrete states to be stored for the verification process. However, the restrictions for which operations this is correct must be kept in mind.

A *remove* operation of a queue or a *set* operation of an *int* is not handled that easily because the precondition is not trivial and the operation is not permutable, respectively. Nevertheless, the *insert* operation is the basic operation for most data structures in the sense that a first-level state can be entirely emulated by a metastate containing only *insert* operations in its O_C component. Consequently, a lot of complexity is saved as will be shown later in the case studies in chapter 6.

4.7 Advanced Operation Specification

As shown in the previous section, it is possible to consider the transitions of the system's states on an operation level instead of on the concrete first-level state. Knowing this, it is possible to define an effect on the level of operations and thus, changing the metastate directly instead of by changing the system's variables and hence, the first-level state.

4.7.1 Plain Metastate Transition

Even if permutability and implied metastates are taken into account, sometimes there are still more entities to be considered in a metastate than in a first-level state. The following example shows this for a short priority queue history which could be easily adapted for any collection-based data structure.

Table 4.9 shows a history of four operations of the $\mathit{PrioQueue}_{\mathit{Spec}}$ specification. First, there are two overlapping *insert* operations. After that, two *remove* operations are executed sequentially. This history leads to the first-level states in Table 4.10 and the metastate descriptions in Table 4.11. The descriptions of the times -1 to 3 are omitted since they are identical to the descriptions in Table 4.3 and 4.8.

Since a *remove* operation does not have a trivial precondition, implied metastates are not present. Further, *remove* operations are not permutable because they always have to remove the element of highest priority at a time. Let us consider each step in detail.

At time 3, there is only one metastate containing both *insert* operations in the O_C component. There are no inherent implied metastates and thus, there

Table 4.9: A history of two *insert* and two *remove* operations

Log ID	Operation	Start	End	Value
0	*insert*	0	2	1
1	*insert*	1	3	2
2	*remove*	4	6	2
3	*remove*	5	7	1

Table 4.10: State evolution derived from Table 4.9

Time	States
4	{[1] , [1,2]}
5	{[] , [1] , [1,2]}
6	{[] , [1]}
7	{[]}

Table 4.11: Plain metastate evolution derived from Table 4.9

Time	Metastates
4	$\{((l_0,l_1,l_2),\emptyset)\ ,\ ((l_0,l_1),\{l_2\})\}$
5	$\{((l_0,l_1,l_2,l_3),\emptyset)\ ,\ ((l_0,l_1,l_2),\{l_3\})\}\ ,\ ((l_0,l_1),\{l_2,l_3\})\}$
6	$\{((l_0,l_1,l_2,l_3),\emptyset)\ ,\ ((l_0,l_1,l_2),\{l_3\})\}$
7	$\{((l_0,l_1,l_2,l_3),\emptyset)\}$

is only a single first-level state to be inferred from the metastate description. This first-level state is denoted as $\sigma_{3,1}$ as the first inferred state at time 3.

At time 4, $\mathit{remove}(2)$ with log ID 2, in the following denoted as remove_2, starts. The condition at time 3 of the only existing state $C(\mathit{remove}_2)|_{\sigma_{3,1}} = \mathit{true}$, because the element of highest priority in the queue is the one with value 2. Consequently, both is possible, that remove_2 takes effect or not. This is indicated by once adding the operation to the O_C sequence, and once by putting it into the O_P set. Now, there are two states $\sigma_{4,1}$, where all operations have been executed and $\sigma_{4,2}$ where remove_2 did not take effect.

At time 5, there are three possibilities. Either both *remove* operations have been applied, or only remove_2, or none of the *remove* operations. It is not possible that remove_3 takes effect without remove_2 taking effect before, because $C(\mathit{remove}_3)|_{\sigma_(4,2)} \neq \mathit{true}$.

At time 6, remove_2 is no longer pending and thus only the metastates are left in which it is completed. At time 7, all operations are completed and consequently there is only one metastate left because it is the only metastate where all operations have been executed.

4.7.2 Redundancy

In Table 4.11 at time 7 there is one metastate left containing four operations. However, in the actual inferred first-level state, there is not a single element in the queue. This is due to the redundancy of *insert*/*remove* pairs.

Redundancy Let σ be the initial state, $(o_1, o_2, ..., o_n)$ a n-tuple of operations and $c = ((X_1, o_1, X_2, o_2, ..., X_n, o_n, X_{n+1}), \sigma)$ a valid configuration where X_i is some operation sequence for $1 \leq i \leq n+1$. Then, we say

$$\begin{aligned} &(o_1, o_2, ..., o_n) \text{ is } \textit{redundant} \text{ in } c \\ \Longleftrightarrow &((X_1, X_2, ..., X_n, X_{n+1}), \sigma) \text{ is valid} \\ \wedge &((X_1, o_1, X_2, o_2, ..., X_n, o_n, X_{n+1}), \{\}) \sim_S ((X_1, X_2, ..., X_n, X_{n+1}), \{\}) \end{aligned} \tag{4.33}$$

Corollary 2: In $\mathit{PrioQueue}_{\mathit{Spec}}$ the operation pair $(\mathit{insert}_1(x),\ \mathit{remove}_1(x))$ is redundant in any valid configuration for all $x \in \mathbb{Z}$, if the remove_1 operation has removed the element inserted by the insert_1 operation.

Proof: Let $((X\,\mathit{insert}_1(x)Y\,\mathit{remove}_1(x)Z), \{Q = \emptyset\})$ be a valid configuration, where X, Y, and Z are some operation sequences.

First of all it has to be proven that $((XY), \{Q = \emptyset\})$ is valid. $(X, \{Q = \emptyset\})$ is valid according to the definition of validity. Now, for any *insert* operation in Y, the condition is fulfilled because $C(\mathit{insert}(x)) = \mathit{TRUE}$. For each *changeKey* operation there are three cases.

1. *changeKey* changes the value x of an object to another value inserted by another *insert* operation. This object has not been modified in Q since it has not been touched by insert_1.
2. *changeKey* changes the value x of the object inserted by insert_1 to another value. $\Rightarrow$ remove_1 has not removed the value of insert_1 $\Rightarrow$ $\lightning$ to the initial assumption $\Rightarrow$ it cannot have changed the value inserted by insert_1 but of another *insert* operation and hence, see case 1.
3. *changeKey* changes the value $y \neq x$ of an object to another value $\Rightarrow$ the object has been inserted by another *insert* operation $\Rightarrow$ This object has not been modified since it has not been touched by insert_1.

Consequently, for each *changeKey* operation in Y its precondition is still fulfilled. For each *remove* operation there are two cases.

1. *Remove* removes an object with value y inserted by another *insert* operation. Let Q_σ be the value of Q at state σ inferred from $((X\,\mathit{insert}_1(x)Y),\ \{Q = \emptyset\})$ right before the *remove* operation is taking effect $\Rightarrow C(\mathit{remove}(y))|_\sigma = \mathit{true} \Rightarrow y$ is the highest value in $Q \Rightarrow y$ is the highest value in $Q \setminus \{o_1\}$, where o_1 is the object inserted by $\mathit{insert}_1 \Rightarrow$ the condition of the *remove* operation is fulfilled without insert_1 taking effect before.

2. *Remove* removes an object with value x inserted by $insert_1$. $\Rightarrow$ $remove_1$ has not removed the object inserted by $insert_1 \Rightarrow \lightning$ to the initial assumption $\Rightarrow$ it cannot have removed the object inserted by $insert_1$ but of another *insert* operation and hence, see case 1.

Consequently, for each *remove* operation in Y its precondition is still fulfilled.

$\Longrightarrow ((XY), \{Q = \emptyset\})$ is valid.

Now, it has to be proven that $\varepsilon(X insert_1(x) Y remove_1(x))(\{Q = \emptyset\}) = \varepsilon(XY)(\{Q = \emptyset\})$. Let $Q_\tau = \varepsilon(XY)(\{Q = \emptyset\})$ be the value of Q at state τ right before the first operation in Z takes effect. As shown before in this proof, there cannot be any operation in Y that has modified the object inserted by $insert_1 \Rightarrow \varepsilon(X insert_1(x) Y)(\{Q = \emptyset\}) = \varepsilon(XY)(\{Q = \emptyset\}) \cup \{o\}$, where $IntVal(o) = x \Rightarrow \varepsilon(X insert_1(x) Y remove_1(x))(\{Q = \emptyset\}) = Q_\tau \cup \{o\} \setminus \{o\} = Q_\tau = \varepsilon(XY)(\{Q = \emptyset\})$

$\Longrightarrow \varepsilon(X insert_1(x) Y remove_1(x) Z)(\{Q = \emptyset\}) = \varepsilon(XYZ)(\{Q = \emptyset\})$

$\Longrightarrow ((X insert_1(x) Y remove_1(x) Z), \{\}) \sim_{PrioQueue_{Spec}} ((XYZ), \{\})\square$

4.7.3 Specifying a Removal: Exploiting Redundancies

There are two issues with the metastate description of Table 4.11. Firstly, there is more memory space needed for performing the verification procedure because all executed operations are stored. Secondly, there are computational steps needed for inferring that the queue Q is empty.

The redundancy property just introduced allows for a different handling of *remove* operations. If all elements were put into the queue by an *insert* operation in its final appearance, a *remove* operation would always remove an element that was inserted before by an *insert* operation in $PrioQueue_{Spec}$. Therefore, with this metaknowledge it is possible to specify a *remove* operation by changing the metastate rather than changing the first-level state as in Equation 4.5

$$\begin{aligned}
&\mathit{remove}() && (4.34)\\
&\quad := ((\mathit{int}\ i), && \text{(parameters)}\\
&\qquad \forall\ \mathit{insert}_i(x) \in O_C : x \leq i\\
&\qquad \wedge \exists \mathit{insert}_j(y) \in O_C \cup O_P : y = i, && \text{(precondition)}\\
&\qquad (O_C, O_P) \rightarrow (O_C \setminus \{\mathit{insert}_j(y)\}, O_P \setminus \{\mathit{insert}_j(y)\})) && \text{(effect)}.
\end{aligned}$$

This specification states that a *remove* operation can take effect, if all completed *insert* operations have inserted only elements of lower or equal priority than the element removed by this *remove* operation. The index i indicates that all *insert* operations in the metastate are well-distinguished. Furthermore, there must exist a completed *insert* operation that has inserted an element with the same priority as the element that is removed. The effect is that the corresponding *insert* operation is removed from the metastate.

Note that the assumption of this specification is that there is no *changeKey* operation in the metastate. However, an *insert*/*changeKey* pair is *semi-redundant* as will be shown in the next paragraphs which makes it possible that there still remain only *insert* operations in the O_C component of a metastate even when applying *changeKey* operations.

Furthermore, here O_C is considered to be a set but actually it is a sequence. Since *insert* operations are permutable there is no difference in this example but it is more readable to consider it as a set in the specification. Usually another notation would have to be used for O_C.

The result with this specification is that the example in Table 4.11 reduces to the metastates in Table 4.13. Now, the metastates never contain more elements in the O_C component than the queue of their inferred first-level states.

4.7.4 Semi-Redundancy

In the history example of Table 4.14 first, there is executed an *insert* operation and then, a *changeKey* operation that changes the value of the inserted element. Naively, this leads to the metastate descriptions in Table 4.15. The

Table 4.12: Metastates from Table 4.11 for comparison

Time	Metastates
4	$\{((l_0,l_1,l_2),\emptyset)$, $((l_0,l_1),\{l_2\})\}$
5	$\{((l_0,l_1,l_2,l_3),\emptyset)$, $((l_0,l_1,l_2),\{l_3\})\}$, $((l_0,l_1),\{l_2,l_3\})\}$
6	$\{((l_0,l_1,l_2,l_3),\emptyset)$, $((l_0,l_1,l_2),\{l_3\})\}$
7	$\{((l_0,l_1,l_2,l_3),\emptyset)\}$

Table 4.13: Metastate evolution derived from Table 4.9 modulo redundancies

Time	Metastates
4	$\{((l_0),\emptyset)$, $((l_0,l_1),\{l_2\})\}$
5	$\{((),\emptyset)$, $((l_0),\{l_3\})$, $((l_0,l_1),\{l_2, l_3\})\}$
6	$\{((),\emptyset)$, $((l_0),\{l_3\})\}$
7	$\{((),\emptyset)\}$

Table 4.14: A history of an *insert* and a *changeKey* operation

Log ID	Operation	Start	End	Value 1	Value 2
0	*insert*	0	1	1	
1	*changeKey*	2	3	1	2

Table 4.15: Plain metastates derived from Table 4.14

Time	Metastates
0	$\{(),\{l_0\})\}$
1	$\{((l_0),\emptyset)\}$
2	$\{((l_0),\{l_1\})\ ,\ ((l_0,l_1),\emptyset)\}$
3	$\{((l_0,l_1),\emptyset)\}$

problem in this metastate description is similar to the problem before for *remove* operations. The first-level state is not trivially inferable and there are cases where there are more elements in the O_C component than in the inferred first-level state. The next property helps to reason about a solution for this problem.

Empty operation The empty operation is defined by default for any specification S as

$$\textit{skip}() := ((), \textit{TRUE}, \_) \tag{4.35}$$

The *skip* operation is always applicable, takes no parameters and does not change the first-level state of a system. It is permutable with any other operation and redundant in any metastate.

Semi-redundancy Let σ be the initial state, $(o_1, o_2, ..., o_n)$ an n-tuple of operations and $c = ((X_1, o_1, X_2, o_2, ..., X_n, o_n, X_{n+1}), \sigma)$ a valid configuration where X_i is some operation sequence for $1 \leq i \leq n+1$. Then, we say

$$\begin{aligned}
&(o_1, o_2, ..., o_n) \text{ is } \textit{semi-redundant} \text{ in } c \\
\Longleftrightarrow &\exists \text{ a sequence of operations } (p_1, p_2, ..., p_n), \\
&\text{where at least one of the } p_i, 1 \leq i \leq n \text{ is the empty operation} \\
&\text{with } ((X_1, p_1, X_2, p_2, ..., X_n, p_n, X_{n+1}), \sigma) \text{ is valid} \\
\wedge &((X_1, o_1, X_2, o_2, ..., X_n, o_n, X_{n+1}), \{\}) \\
&\quad \sim_S ((X_1, p_1, X_2, p_2, ..., X_n, p_n, X_{n+1}), \{\})
\end{aligned} \tag{4.36}$$

Semi-redundancy is a generalization of redundancy.

Corollary 3: In $\mathit{PrioQueue}_{\mathit{Spec}}$ a pair $(\mathit{insert}_1(x),\ \mathit{changeKey}_1(x,y))$ is semi-redundant in any valid configuration for all $x, y \in \mathbb{Z}$, if the $\mathit{changeKey}_1$ operation has changed the value of the element inserted by the insert_1 operation.

Proof: Let $((X\,\mathit{insert}_1(x)Y\,\mathit{changeKey}_1(x)Z), \{Q = \emptyset\})$ be a valid configuration, where X, Y, and Z are some operation sequences. Then the pair $(\mathit{skip}(),\ \mathit{insert}_2(y))$ fulfills the necessary properties in the definition for semi-redundancy if $\mathit{insert}_2(y)$ inserts the same object as $\mathit{insert}_1(x))$ but with different value.

$((X\,\mathit{skip}()Y), \{Q = \emptyset\}) = ((XY), \{Q = \emptyset\})$ is valid for similar reasons as in the proof of Corollary 2.

Now, it has to be proven that $\varepsilon(X\,\mathit{insert}_1(x)Y\,\mathit{changeKey}_1(x,y))(\{Q = \emptyset\}) = \varepsilon(XY\,\mathit{insert}_2(y))(\{Q = \emptyset\})$. Let $Q_\tau = \varepsilon(XY)(\{Q = \emptyset\})$ be the value of Q at state τ right after the last operation in Y takes effect. Then as in the proof before, $\varepsilon(X\,\mathit{insert}_1(x)Y)(\{Q = \emptyset\}) = \varepsilon(XY)(\{Q = \emptyset\}) \cup \{o\}$, where $\mathit{IntVal}(o) = x \Rightarrow \varepsilon(X\,\mathit{insert}_1(x)Y\,\mathit{changeKey}_1(x,y))(\{Q = \emptyset\}) = Q_\tau \cup \{o\}$, where $\mathit{IntVal}(o) = y$.

On the other hand, $\varepsilon(XY\,\mathit{insert}_2(y))(\{Q = \emptyset\}) = Q_\tau \cup \{o\}$, where $\mathit{IntVal}(o) = y. \implies \varepsilon(X\,\mathit{insert}_1(x)Y\,\mathit{changeKey}_1(x,y)Z)(\{Q = \emptyset\}) = \varepsilon(XY\,\mathit{insert}_2(y))(\{Q = \emptyset\})$

$\implies$ In $\mathit{PrioQueue}_{\mathit{Spec}}$ $((X\,\mathit{insert}_1(x)Y\,\mathit{changeKey}_1(x,y)Z), \{\})$ is equivalent to $\varepsilon(XY\,\mathit{insert}_2(y))(\{\})\Box$

4.7.5 Exploiting Semi-Redundancies in a Specification

As we have seen, an *insert*/*changeKey* pair is semi-redundant in $\mathit{PrioQueue}_{\mathit{Spec}}$. It can be represented by discarding the first *insert* operation and replacing the *changeKey* operation by another *insert* operation. Consequently, an alternative specification of the *changeKey* operation to the one of Equation 4.6 in $\mathit{PrioQueue}_{\mathit{Spec}}$ would be

Table 4.16: Metastates from Table 4.15 for comparison

Time	Metastates
0	$\{(),\{l_0\})\}$
1	$\{((l_0),\emptyset)\}$
2	$\{((l_0),\{l_1\})\ ,\ ((l_0,l_1),\emptyset)\}$
3	$\{((l_0,l_1),\emptyset)\}$

Table 4.17: Metastates derived from Table 4.14 without semi-redundancies

Time	Metastates
0	$\{(),\{l_0\})\}$
1	$\{((l_0),\emptyset)\}$
2	$\{((l_0),\{l_1\})\ ,\ ((l_2),\emptyset)\}$
3	$\{((l_2),\emptyset)\}$

$$\begin{aligned}
&\mathit{changeKey}(\mathit{int}\ i_1, \mathit{int}\ i_2) && (4.37)\\
&\quad := ((\mathit{int}\ i_1, \mathit{int}\ i_2), && \text{(parameters)}\\
&\qquad \exists \mathit{insert}_{j_1}(y) \in O_C \cup O_P : y = i_1, && \text{(precondition)}\\
&\qquad (Q_C, Q_P) \rightarrow ((O_C \cup \{\mathit{insert}_{j_2}(i_2)\}) \setminus \{\mathit{insert}_{j_1}(y)\}, && \text{(effect)}\\
&\qquad O_P \setminus \{\mathit{insert}_{j_1}(y)\})).
\end{aligned}$$

Now the metastate description in Table 4.15 changes to the one in Table 4.17. The operation ID *2* denotes an artificially created *insert* operation that inserted an element with value 2. Since a *changeKey* operation acts like a sequence of a *remove* and an *insert* operation with a new value, the new insert_2 operation may have the same start and end timestamp as the *changeKey* operation.

4.8 Additional Special Cases

It can happen that during the execution of a program an operation is executed but has no effect for some reason. For example a *remove* operation might be executed on an empty data structure. Nevertheless, this case has to be treated correctly. If such an operation is executed but does not perform its actual task, there must be a description of what happened.

According to the example specification of a priority queue, a *remove* operation that removes a *null* value can only occur if the queue is empty at the time of its occurrence according to the precondition of the operation specification. An operation that does not change the system's state is redundant in itself and thus, does not need to be considered in the O_C component of a metastate at all.

An *insert* operation that inserts a *null* value is invalid per definition. If however a maximum queue size is defined, then an insertion of a *null* value can become valid. In the next chapter there will be a more detailed description of how special cases are treated for example programs.

4.9 Summary

This chapter introduced a new verification scheme for verifying execution traces against linearizability that does not require any backtracking and has proven its validity. First, an outline of the method has been given. Second, the assumed specification format has been described. Third, the format of the history that is necessary has been given. Fourth, an alternative view of the system's state by using the level of metastates has been detailed. Fifth, another specification possibility has been given with the use of metastates. Sixth, it has been proven that the verification methodology is generally applicable. At last, some possible special cases have been mentioned.

The next chapter uses all the theory of this chapter and shows how this methodology is applicable in practice.

5. Implementation

The previous chapter has given all necessary formal background for a verifier implementation and proven that this method is applicable. This chapter describes the implementation that realizes the method. One goal is to have a framework that can be easily adapted to complex concurrent systems for verifying their correctness against linearizability. Another goal is to show example implementations for concrete verification case studies. The implemented verifier is capable of verifying not only common data structures like lists or sets but also more complex ones like the priority queue which has never been proven against linearizability before.

First, this chapter presents the general framework. After that, it outlines the implementation of operation handlers. Finally, it mentions optimization elements. Parts of the implementation described in this document are patented in [DrBGE10].

5.1 Framework

The implementation of the verifier for a specific case consists of two parts. The first part is the generic framework which is adaptable code for special purpose verifications. It provides the basic structures that are necessary for an application of the method introduced in this thesis. It assumes a determined format for the history that is verified. Furthermore, it defines interfaces that have to be provided for a specific verification. This framework can be used in all verification cases applying this method. The second part is the concrete implementation of the provided interfaces that will be described in the next section. An overview of the verification procedure is given in the previous chapter in Figure 4.1. Following three points form the critical parts of the implementation

1. Space efficiency
2. Time efficiency
3. Ensuring completeness of the solution

The first two items are issues that can be tackled by both the general framework and the implementations of the provided interfaces. The third item is an issue that is not a general problem but rather an issue for some special cases like the priority queue. Space efficiency is tackled by gathering common information among different metastates. It is very important to make the implementation as space efficient as possible since the increased performance is gained by paying with higher space requirements. Although, the implemented method is faster than currently available methods, it is always an issue to optimize the performance as much as possible. Ensuring the completeness of the solution is very challenging since this requires the realization of a complete recursion so that all possible sequences are covered.

5.1.1 Building a History

The verifier runs on a test-case. So before the verifier can run, the system under test (SUT) needs to produce a history by logging all operations and their effects. Hence, the three basic steps of the verification procedure are:

1. Devise a test scenario
2. Instruct the SUT to build a history
3. Verify the history

The overview of the use cycle for the verification is schematized in Figure 5.1. After the design of the code the three basic steps of the verification are executed. If the verification issues that the histories generated by the SUT are

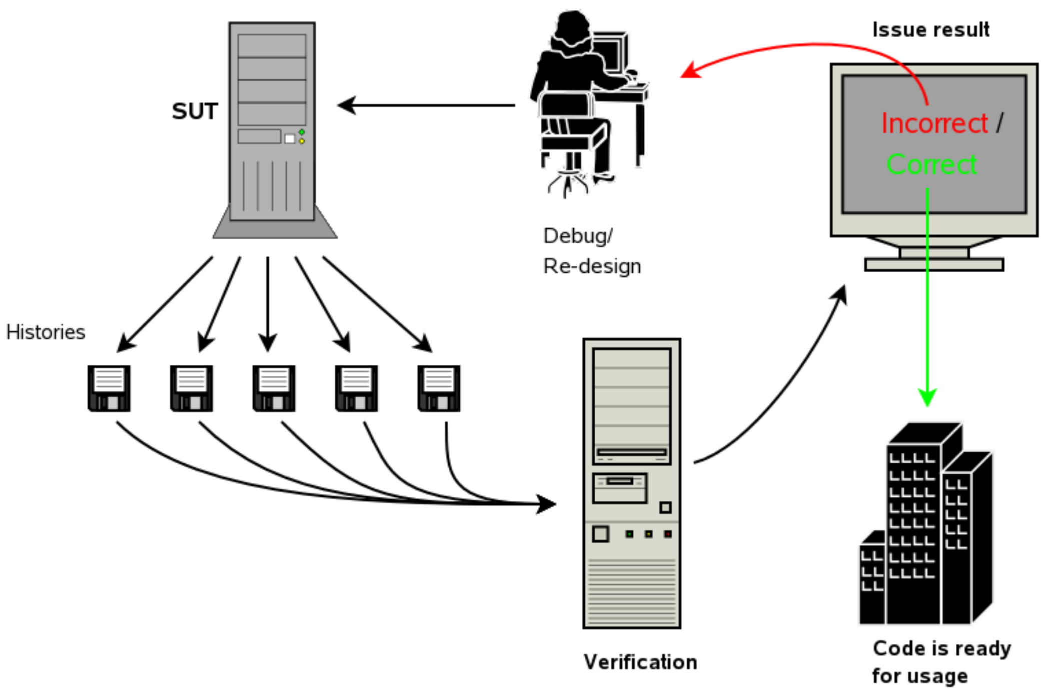


Figure 5.1: Schema of the design/verification cycle for the use of the introduced method

correct, the code can be used. Otherwise, more design or debug cycles are necessary and the changed code has to be verified once more. It is necessary to define sensible test cases for the SUT to obtain expressive verification results.

The memory space for logging operations has to be allocated as a whole before the beginning of a test run. By this, no time is wasted for memory allocation during the execution of the actual program code. If time is wasted unnecessarily during a test of a concurrent program, this means that the test scenario has lesser characteristics of a stress test which makes the occurrence of potential errors less likely. A concurrent program is much more likely to produce potential errors if as many parallel accesses are executed as possible within a time period.

The number of operations executed during a test run is limited to a fixed number so that it is assured that the allocated memory space is sufficient for all logged operations. During the test run, a global pointer points to the first free memory location for a log entry. This memory space is reserved at the

Table 5.1: The format of log entries in a history

Log ID	Operation	Start	End	Thread	Value
0	*remove*	0	2	1	null
1	*insert*	1	5	2	5
2	*insert*	3	7	1	4
3	*remove*	4	6	3	4

Table 5.2: A history of operations sorted according to event times

Time	Log ID
0	0
1	1
2	0
3	2
4	3
5	1
6	3
7	2

beginning of an operation by the compare-and-swap instruction (CAS) which atomically sets the pointer to the next free memory location. When an operation is ending, it fills the reserved memory with all necessary information for the verifier. This information is the ID of the operation, the operation name, the start time, the end time and the values manipulated by the operation. Often the ID can be identified with the start or end time because these values are unique by definition. In addition, a thread ID can help to analyze the structure of the history but it is not essential for the actual verification procedure.

After finishing the test run, the entire log history is stored in a file as binary data. Now, the history is ready to be analyzed.

5.1.2 Preprocessing the History

The test run of the program produces an unsorted history in a format similar to the one shown in Table 5.1. This format contains operations in some arbitrary order. However, in this raw appearance the necessary information for the verification process is not conveniently extractable.

Therefore, a new list of sorted entries is produced as in Table 5.2. Here, each operation occurs twice, once at its start time and once at its end time (compare section 4.3).

5.1.3 Verifier Harness

The verifier harness performs the actual verification. Figure 5.2 shows the UML schema of the most important data structures of the harness.

For all SUT's, the verification procedure is basically the same. The first log entry in the sorted list is removed and checked whether the current virtual time is equal to the start time or the end time of the entry. The operation of the log entry is extracted and the proper operation handler is called. An operation handler for a specific operation has to implement an interface that provides two functions. One that handles the start and one the end event of an operation. After handling all events faultlessly, the success of the verification is issued.

The state set is managed mainly by two data structures which are the *MetaState* and the *MetaStateUniverse*. The *MetaState* contains a list for pending operations, one for finished operations and one for neutralized operations. Finished operations are part of the completed component introduced in the previous chapter. If operations are permutable, it is not necessary to store their precise sequence. Neutralized operations are those which form a redundant operation sequence in the specification. Hence, all finished and neutralized operations with the proper pending operations that are ready to take effect at any time imply the actually managed first-level state.

The *MetaStateUniverse* forms the entire metastate description. It contains all *MetaState*s and a *DefiniteState* that stores a sequence of finished operations modulo redundancies that is common for all *MetaState*s in the *MetaStateUniverse*. Figure 5.3 shows the logical structure of *MetaState*s which are composed by their three components completed by the *DefiniteState*.

The *DefiniteState* decreases significantly the required memory because if for example a collection contains definitely x elements at some point in time and if there are y different possible metastates, then we save $(x - 1) \cdot y$ references

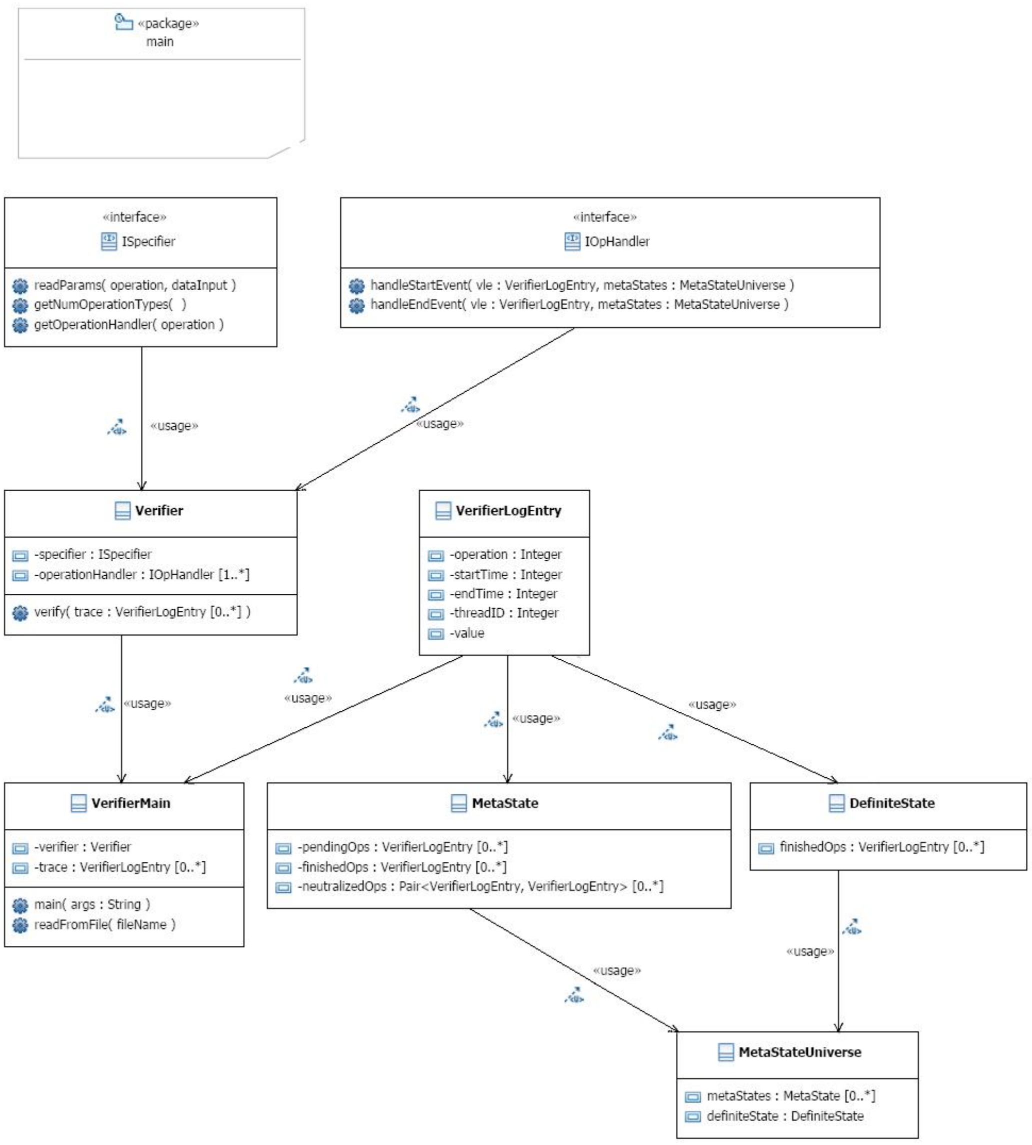

«package»
main
«interface»
ISpecifier
readParams(operation, dataInput)
getNumOperationTypes( )
getOperationHandler(operation)
«interface»
IOpHandler
handleStartEvent(vle : VerifierLogEntry, metaStates : MetaStateUniverse)
handleEndEvent(vle : VerifierLogEntry, metaStates : MetaStateUniverse)
«usage»
«usage»
Verifier
-specifier : ISpecifier
-operationHandler : IOpHandler [1..*]
verify(trace : VerifierLogEntry [0..*])
VerifierLogEntry
-operation : Integer
-startTime : Integer
-endTime : Integer
-threadID : Integer
-value
«usage»
«usage»
«usage»
«usage»
VerifierMain
-verifier : Verifier
-trace : VerifierLogEntry [0..*]
main(args : String)
readFromFile(fileName)
MetaState
-pendingOps : VerifierLogEntry [0..*]
-finishedOps : VerifierLogEntry [0..*]
-neutralizedOps : Pair<VerifierLogEntry, VerifierLogEntry> [0..*]
DefiniteState
finishedOps : VerifierLogEntry [0..*]
«usage»
«usage»
MetaStateUniverse
metaStates : MetaState [0..*]
definiteState : DefiniteState

Figure 5.2: Verifier harness schema

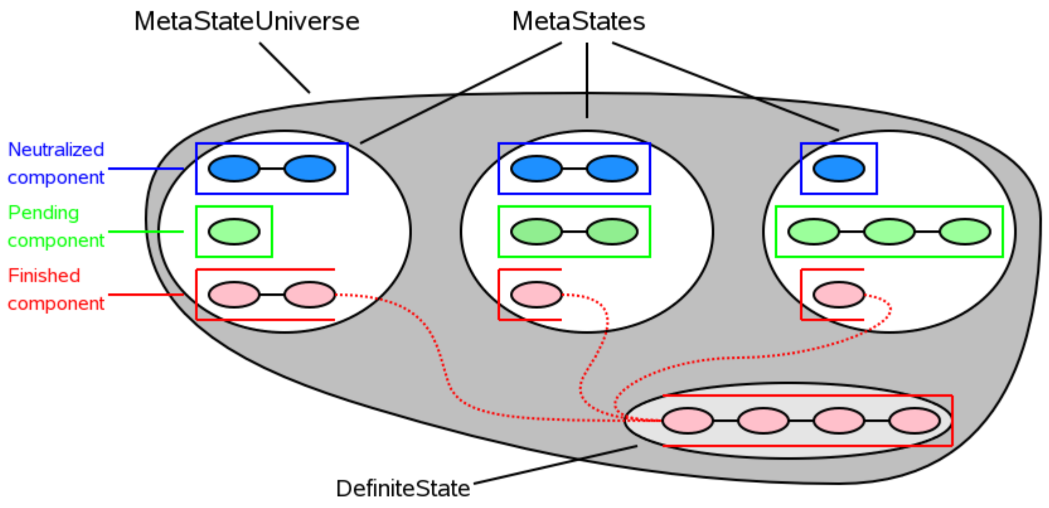


Figure 5.3: Logical scheme of *MetaStates* in the implementation

to operations. In practice, the *DefiniteState* often contains more finished operations than the currently managed *MetaStates* altogether. Note, that the concept of a shared state for all state alternatives has to be treated carefully if operations are not permutable.

Another advantage is that if the operations in the *DefiniteState* are stored in a sophisticated manner, it is very fast to find the relevant operation entries. For example in the case of a priority queue, the *DefiniteState* itself can be a priority queue so that the important operations are obtained quickly. Hence, the *DefiniteState* saves time and space.

5.1.4 Specifier Interface

The *specifier interface* provides all functions that are necessary for an operable verification. It defines how the parameters for each operations are read since they can vary in number and types. Furthermore, it delivers the *OperationHandlers* for each operation which are described in the next section. It also stores the internal appearance of the *DefiniteState* that is used for optimization purposes. Lastly, it also defines print functions for all kinds of data structures that are important especially for the debugging procedure.

5.2 Operation Handlers

For each data structure or system to be verified there are operation handlers for each operation necessary. The *OperationHandler* is declared by an interface and implements the start and the end event handling for an operation.

5.2.1 End Event Handler

The end event handler is implemented for most operations in a very similar manner. Following steps are executed for an operation O

```
1: for each MetaState in MetaStateUniverse do
2:   if $O \in$ MetaState.pending then
3:     if $O$ is trivially applicable in MetaState then
4:       shift $O$ from MetaState.pending to MetaState.finished
5:     else
6:       discard this MetaState
7:     end if
8:   end if
9:   if $O$ is in a tuple in MetaState.neutralized then
10:     remove the tuple if all its operations occurred in the past
11:   end if
12: end for
13: if MetaStateUniverse is empty then
14:   issue “History not linearizable”
15: end if
16: if $O \in$ MetaState.finished $\forall$ MetaState $\in$ MetaStateUniverse then
17:   remove $O$ from all MetaStates and put it into the DefiniteState
18: end if
19: check for equivalent states
```

So first, the end event handler checks for all *MetaState*s in the *MetaStateUniverse* whether the currently handled log entry is in the pending operations component of the *MetaState*. In the case of *insert* operations for lists or priority queues, the log entry is merely shifted from the pending component to the finished component since such an operation can never fail. For other operations, the *MetaState* is discarded if an operation is located in the pending component.

Afterwards, the *MetaStateUniverse* is checked on emptiness. If there is no *MetaState* left, the failure of the verification is issued.

Furthermore, elements in the neutralized component which are mostly pairs but sometimes higher tuples of operations are removed if all operations in the tuple are in the past relative to the current event time. That is, all log entries have an equal or earlier end time than the currently handled entry.

5.2.2 Start Event Handler

The complexity of the start event handler depends on the properties of the handled operation. If an operation O is handled that is permutable and ready to take effect at any time, the handler is very simple.

```
1: for each MetaState in MetaStateUniverse do
2:   if there exists fitting neutralizing operations in MetaState.pending then
3:     create new states with this neutralized tuple
4:   end if
5:   add $O$ to MetaState.pending
6: end for
7: check for equivalent states
```

Here, for each *MetaState* the current log entry is put into the pending list. If there are other log entries of operations that are in the pending component which might neutralize the currently handled operation, all possible neutralizations are considered and respected in a new state each. Examples for operations of this kind are *insert* operations for lists and priority queues. If an operations is permutable but not trivially applicable, than it has to be checked whether the invariant is fulfilled.

```
1: for each MetaState in MetaStateUniverse do
2:   if $O$ is applicable in MetaState then
```

```
3:          apply $O$ in all ways and create new states with $O$ being neutralized or
            finished
4:      **end if**
5:      add $O$ to *MetaState.pending*
6:  **end for**
7:  check for equivalent states
```

Since this operation is permutable there is no recursion necessary for further investigation of the *MetaState*. One example for such an operation is the *remove* operation of lists. The most complex case is the handling of operations that are not permutable and consequently not trivially applicable.

```
1:  for each MetaState in MetaStateUniverse do
2:      if $O$ is applicable in MetaState then
3:          apply $O$ in all ways and create new states with $O$ being neutralized or
            finished
4:      end if
5:      add $O$ to MetaState.pending
6:  end for
7:  for all newly created MetaStates do
8:      perform a recursive check for all operations in MetaState.pending for
        applicability
9:  end for
10: check for equivalent states
```

Elaborate recursions are necessary to ensure that all possibilities of orderings are covered during the verification procedure. For each existing state, it is checked whether the currently handled operation can be applied. Furthermore, it is added as pending in each state. If it can be applied in a given state, a new state is created with the necessary modifications. Then for each new state, a recursive check for potential applications of pending operations is performed. This recursion produces new states with each iteration. In the case of a *remove* operation it is possible that in some states multiple possibilities are available for an operation application. If the value x shall be removed but there are multiple x's inserted, all possibilities have to be considered in a combinatorial way which is a non-trivial effort.

In all three cases, redundancies might occur in the sense that after applying all recursions and applications of operations that there are multiple equivalent states that have to be eliminated. Consequently, in a last step the *MetaStateUniverse* is checked for equivalent states which are discarded. This step is executed by the verifier framework rather than the operation handler and can be optimized as described in the next section.

5.3 Optimization

As described before, after each operation handling there is a consolidation step for gathering identical states. This step can be optimized by computing a hash value for each state depending on the operations finished and pending in a *MetaState*. By this, *MetaState*s with different hash values are certainly not identical and consequently do not need an entire operation-by-operation comparison for all sets. However, if there is a hash collision, all operations must be compared because it is not certain whether the two *MetaState*s are really identical. Nevertheless, comparing hash values will save a detailed operation-by-operation comparison most of the time and hence makes it a useful optimization.

In the case of our implementation, the hash value is a 64-bit long integer. The first 32 bit encode the pending operations whereas the second 32 bit encode the finished operations. The hash value for the empty *MetaState* is 0. Neutralized operations are considered as finished.

In Java each object has a 32-bit hash value. Hence, if h_e is the 32-bit hash value of a *VerifierLogEntry* e and h_m is the 64-bit hash value of a *MetaState* m and e is added as pending to m then the new hash value h'_m of m is

$$h'_m = h_m \oplus h_e \tag{5.1}$$

The advantage of the bitwise *XOR* operator is that if the operation is removed from the pending component, the new value can be computed by simply using the same function again. The obtained hash value will be the same as before adding the operation. If e is added to the finished or neutralized component of m then

$$h'_m = h_m \oplus (h_e \cdot 2^{32}) \tag{5.2}$$

Again, the removal of an entry from the finished component induces the same computation of the new *MetaState* hash value. If an entry is shifted from the pending to the finished or neutralized component, first Equation 5.1 and then Equation 5.2 is applied. The effectiveness of this optimization is discussed in the next chapter.

There is still space left for optimizations, but they are often connected to special use cases. One possibility is to perform the verification in a multi threaded manner. However, deeper thoughts on how the results of parallel verification procedures can be attached to each other properly are necessary. The parallelizability of a problem is also strongly attached to the considered use case. Moreover, there is the question of how redundancies can be avoided from occurring at all instead of consolidating them after they have appeared already. Furthermore, a general question is how the test cases could be generated such that errors occur as likely as possible if existent so that less test cases are needed for a practical result. Those are still open questions that could be dealt with in future work.

5.4 Applicability

The framework has been adapted for a number of different data structures. The simplest case study data structure is the list. Furthermore, this method is implemented also for sets. It is of no relevance, in which way the checked data structures have been implemented. For the verifier it is not necessary to know whether the list is implemented by a linked list, an array list or a vector. Both, the list and the set, have already been verified by other tools before.

However, this method has also been implemented for priority queues which up to now have never been verified by any tool known to the author. The next chapter shows an evaluation and test results that demonstrate the efficiency and capability of the tool and the method. It is compared to a brute force approach described in the next section. In addition to that, there is a theoretical analysis of the expressiveness of the outcome of the verification procedure and how it can be used and interpreted.

5.5 Summary

This chapter described in detail the implementation of the verifier method and highlighted the main programming issues. It started with an overview of the generic framework of the implementation. Then, it elucidated the functionality of the operation handlers which form the core of each verification application. Finally, it touched the realized scenarios for this verification method. The next chapter analyzes the practicability and behavior of the verifier implementation and shows performance numbers.

6. Evaluation

The previous chapter described the realization of the verification methodology. This chapter provides an evaluation of the introduced method and its implementation. First, the experiments are described. Second, the brute-force approach is detailed that is used as a reference. Third, the performance results are presented. At last, other issues are mentioned like space consumption and effectiveness of the introduced method.

6.1 Description of the Experiments

The experiments in this evaluation focus mainly on the performance comparison between the method introduced in this thesis and the brute-force approach which until now has been the only approach used in the literature for verifying execution histories to the best of the author's knowledge. To some extent the optimized method mentioned in section 5.3 that uses hashes for *MetaState*s is also included in the analysis. In the current literature there is no verification code available for histories generated by priority queue implementations that could be used for a comparison with the introduced method. Therefore, the code described in the next section has been used as a reference for brute-force approaches.

Two implementations of priority queues have been used to produce histories for different settings. The first implementation is a coarse-grained locking implementation in which in each operation a thread is acquiring a global lock for the execution. The second implementation is the fine-grained STM-based implementation described in section 2.7. During its development phase, this implementation has also produced incorrect histories due to coding errors, which

Table 6.1: Test cases for the verified histories

Operations	200K	300K	400K	500K	600K
2 threads	×	×	×	×	×
3 threads	×	×	×	×	×
4 threads	×	×	×	-	-
5 threads	×	×	-	×	-
6 threads	×	×	-	-	×

have been used as examples. By the aid of the verifier, those coding errors could be identified and fixed. However, it is not possible to guarantee the correctness of the implementation for all possible executions due to the nature of the method as will be discussed later. An element of risk remains.

The histories have been generated on a Linux SMP-system running on an Intel Xeon multi processor with 8 (2×4) cores. Due to the limited availability of this system, the histories have been verified in a Java 5.0 runtime environment on an Intel Pentium 4 dual core machine. The verification has been performed single-threaded, though. A multi-threaded execution of the verification procedure could be an issue for future work.

The covered test cases are marked in Table 6.1 by x's. These test cases are adequate for showing the behavior of the verifiers in different dimensions. Five lines can be identified - two horizontal, two vertical and a diagonal line. The numbers of total operations increase from left to right whereas the number of threads increases from top to bottom. Each thread executes $\frac{\textit{Total operations}}{\textit{Number of threads}}$ operations.

For each test case three histories per implementation are generated. The verification of a history produces two numbers of relevance which are

- the preprocessing time
- the verification time

The preprocessing time includes the reading of the history file and the preparation of the data structures that are used for the actual verification like the sorted list of log entries in our method. The verification time measures the time from the first step of the actual verification algorithm to the moment the result is issued. For both values the average of the three times measured are taken as the performance result for a test case.

6.2 Brute-Force

The brute-force approach is used as a reference for comparison with the method introduced in this dissertation. It uses the input described before and systematically checks all possible orderings of operations one by one respecting the time constraint of linearizable histories. Fraser used a brute-force verification approach in [Fras03] as well. However, he verified implementations of sets that contain elements with a key and a value whereas in the test scenarios considered here, there are elements with values only. Not all implementations of data structures use keys but primitive values only and hence it is desirable to have a verification method that is more generic than that. If keys are absent, this is a higher challenge for the verification tool because finding the correct pair of inserting and removing operations is much harder whereas with keys it is clear which removing operation has removed the element of which inserting operation. This is the reason why the brute-force approach used here had to be adapted to become more generic and will be detailed in the next subsections.

6.2.1 Preprocessing the History

For a higher efficiency during the verification procedure, preprocessing is very important for the brute-force approach. Following data structures contain necessary information during the verification.

sortedEntr A tree of log entries sorted according to their end time

parEntr A tree map consisting of log entries as keys sorted according to their start time and lists of parallel log entries as values.

nextEntr A tree map consisting of log entries as keys sorted according to their start time and lists of next log entry candidates as values.

The idea is that the *parEntr* tree map stores all parallel operations to the key operation whereas the *nextEntr* tree map stores all operations that might follow the key operation as next. An operation a is a *next operation candidate* of b if the following conditions are fulfilled.

- The start time of a is later than the end time of b
- There exists no operation c which has a start time later than b's end time and an end time earlier than a's start time

It is obvious that the first condition must hold, otherwise a would be parallel to or earlier than b. If the second condition does not hold, than c must occur after b but before a. Thus a could never follow b directly. An example of a history is shown in Figure 6.1. It leads to the mappings in *parEntr* and *nextEntr* as shown in Table 6.2.

6.2.2 Algorithm

Operations that are in the *parEntr* value of a key operation o may follow o or can be followed by o in a sequential history according to the time constraint of the linearizability definition. On the other hand, operations that are in the *nextEntr* value of a key operation o may follow o but cannot be followed by o in a sequential history. Consequently, the Figure 6.2 can be inferred from Table 6.2. Now, the goal is to find a traversing of the graph starting from *Start* that covers each node exactly once and respects the arrows' directions.

As the name indicates, the brute-force algorithm checks this by counting up all possibilities. The sequences are checked on the fly on a violation of the object invariant. A number of different variables are needed throughout the verification procedure.

current The log entry that is currently checked on applicability

before The log entry that has been previously checked on applicability but failed

execPath The entire currently assumed execution path

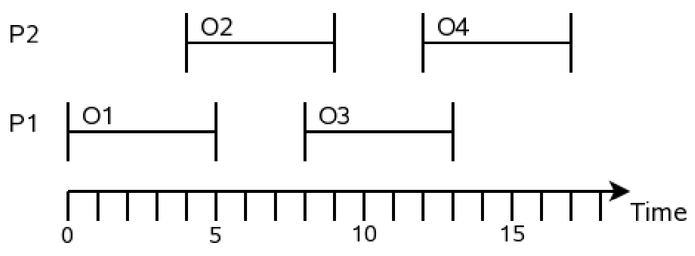


Figure 6.1: History example for the brute-force illustration

Table 6.2: The format of log entries in a history

Key operation	Parallel entries	Next entries
Start	-	O1, O2
O1	O2	O3, O4
O2	O1, O3	O4
O3	O2, O4	-
O4	O3	-

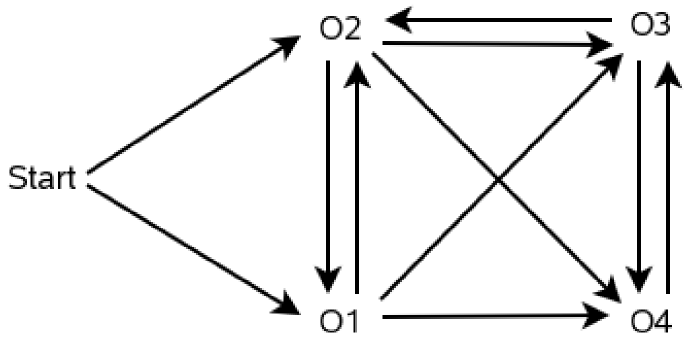


Figure 6.2: Resulting graph from Table 6.2

The algorithm starts by picking the first element in the list which is the value of the first key in the *next* map and sets *current* to that element. The first key of the *next* map is an artificially generated key that indicates that no operation has been executed and thus, the value contains all candidates of operations that might be the first operation to take effect according to the time constraint.

This first operation is applied by a sequential implementation of the checked system or data structure. If it can be applied, its entry is removed from the sorted tree and used as the key value for the search of the next operation candidates that can be applied. Furthermore, the operation is added to the *execPath* and *before* is set to *null* indicating that the previous operation has been successfully applied. If it cannot be applied, *before* is set to *current.* The pseudo code of each of the next iterations of the brute-force algorithm is shown below.

```
if $execPath$ is empty then
    $current$ := $nextEntr$.getFirstValue().getNext($before$)
    if $current$ = $null$ then
        all possibilities have been checked and the verification failed
    end if
else
    $lastExecuted$ := $execPath$.getLast()
    if $parEntr$.getValuesOf($lastExecuted$) is empty then
        $current$ := $null$
    else
        if $before$ = $null$ then
            $current$ := $parEntr$.getValuesOf($lastExecuted$).getFirst()
        else
            $current$ := $parEntr$.getValuesOf($lastExecuted$).getNext($before$)
        end if
    end if
    if $current$ = $null$ then
        if $before$ = $null$ then
            $current$ := $nextEntr$.getValuesOf($lastExecuted$).getFirst()
```

```
20:       else
21:          $current := nextEntr$.getValuesOf($lastExecuted$).getNext($before$)
22:       end if
23:    end if
24: end if
25: if ($current = null$)$\vee$($current$.startTime $>$ $sortedEntr$.getFirst().startTime)
    then
26:    $before = execPath$.pollLast()
27:    $sortedEntr$.add($before$)
28: else
29:    if $current$ is applicable in $execPath$ then
30:       $before = null$
31:       $execPath$.add($current$)
32:       $sortedEntr$.remove($current$)
33:       if $sortedEntr$ is empty then
34:          a valid sequential order has been found and the verification suc-
             ceeded
35:       end if
36:    else
37:       $before = current$
38:    end if
39: end if
```

Note that each operation may occur only once in the *execPath* in the end. However, due to overlaps of operations the above iteration might select already handled operations as *current*. This problem is avoided by marking entries that are put into the *execPath* and skipping them in the selection process for the *current* entry.

By using the data structures build up in the preprocessing phase, the way how this algorithm counts up all possibilities can be illustrated by a tree of checked execution paths. The tree in Figure 6.3 shows the example obtained from the graph in Figure 6.2. A path from the root to a leaf is a complete execution path respecting the arrows' directions without traversing any node more than

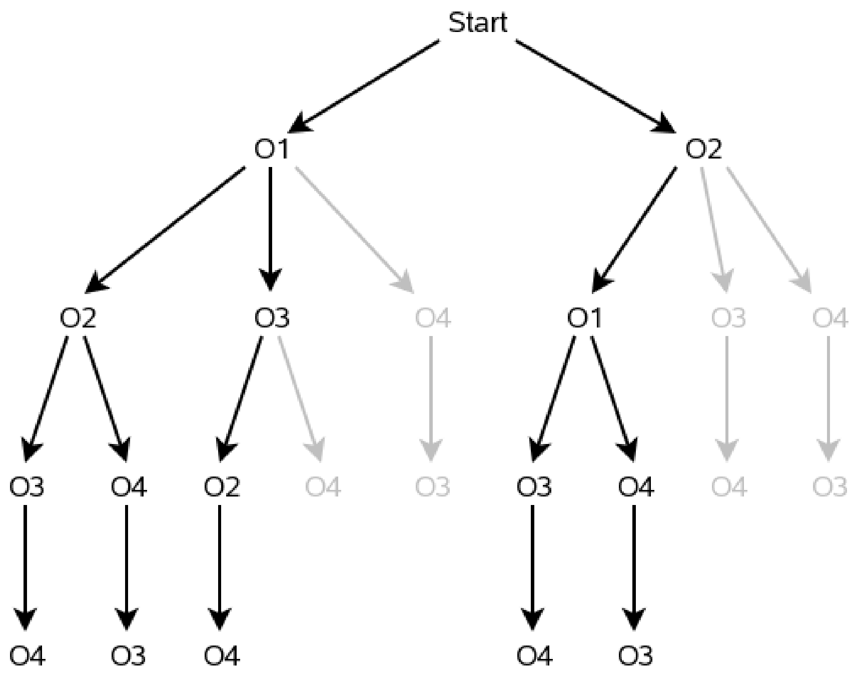


Figure 6.3: Execution tree resulting by applying the brute-force approach for Table 6.2

once. The gray paths are shorter than the number of operations handled. These paths break the time constraint and hence are not further checked by the verifier. The tree is traversed from left to right by the brute-force verifier and each potentially valid (black) path is checked for the validity of the systems invariant. As soon as a correct path has been found, the algorithm terminates.

6.3 Performance Results

This section summarizes the obtained performance results. The results will be discussed in the three dimensions - horizontal, vertical, diagonal - as mentioned in section 6.1. In the following, *systematic* refers to the verification method introduced in this thesis and *optimized* denotes the systematic method including the optimizations described in section 5.3. Although, the brute-force method follows a system as well, we will stick to this denotation as it is not as refined as in our approach.

6.3.1 Horizontal Analysis

The first part of the evaluation of the performance results deals with the behavior of the verification methods for an increasing amount of operations in histories which is referred to as *horizontal analysis*, here. The chart in Figure 6.4 shows the horizontal picture for verifying histories generated by the STM-based implementation executed by two threads. The brute-force approach performs better than the systematic approach for an amount of up to 500 thousand operations in total. However, for 600 thousand operations the introduced method overtakes brute-force. There is no significant difference between the optimized and the standard systematic method.

The graph shows a quite exponential growth of time consumption with increasing numbers of operations for the brute-force method. The factor per 100k operations is about 1.8. The systematic methods show a fairly linear growth.

The picture looks even worse for brute-force if the preprocessing time is added to the graphs as in Figure 6.5. Here, only the preprocessing takes more time then the total time of the systematic methods.

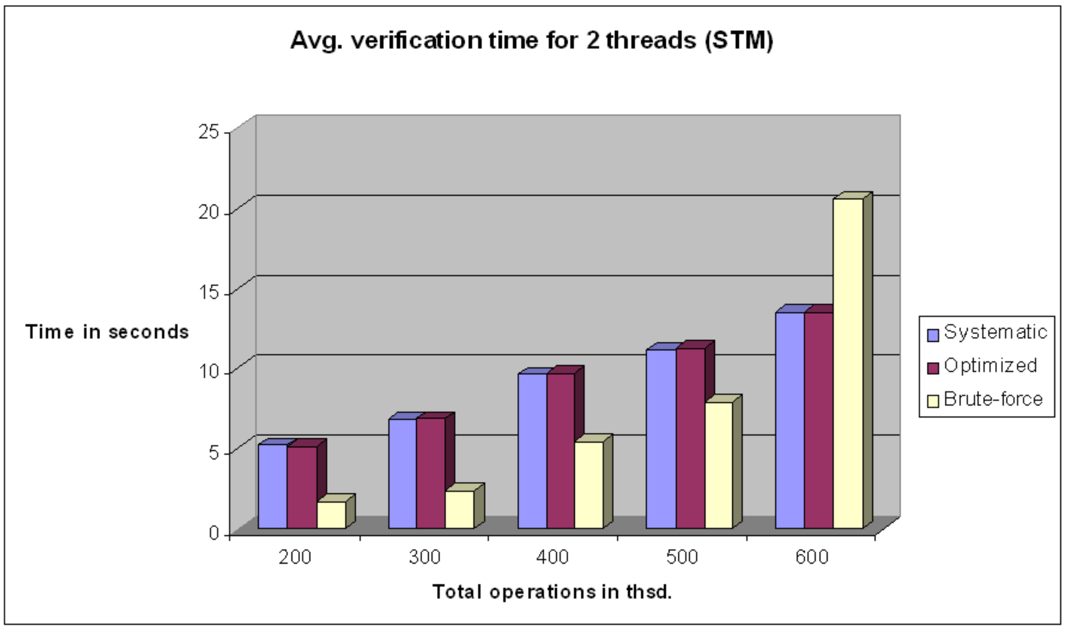


Figure 6.4: Horizontal chart for 2 threads of the STM implementation (verification time)

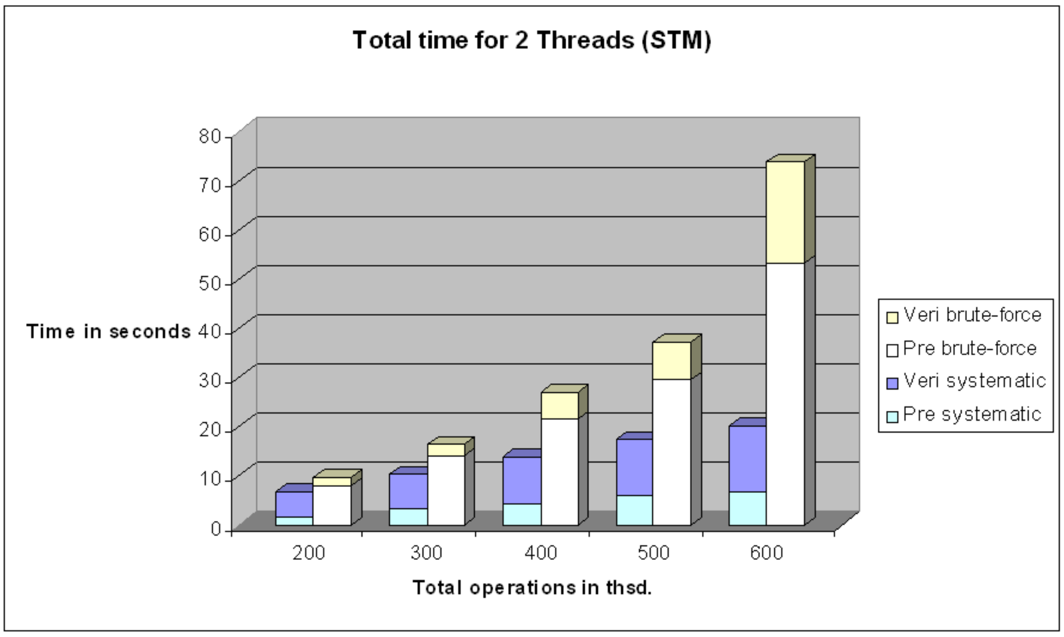


Figure 6.5: Horizontal chart for 2 threads of the STM implementation (total time)

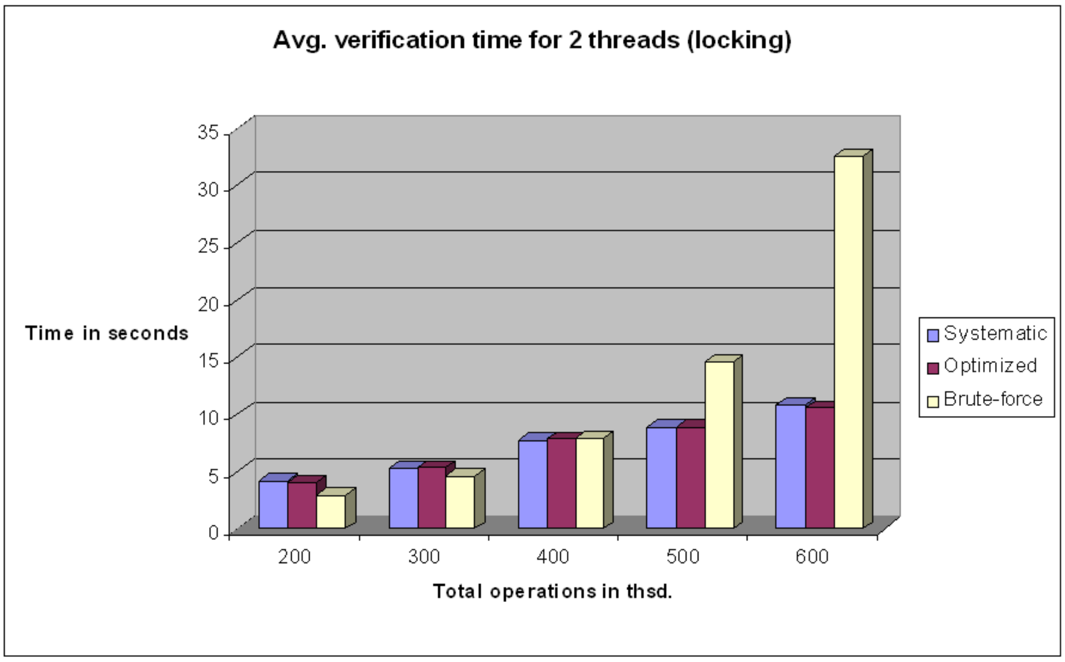


Figure 6.6: Horizontal chart for 2 threads of the locking implementation (verification time)

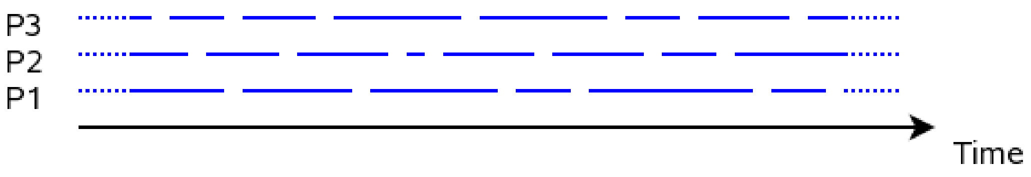


Figure 6.7: Schema of the structure of histories produces by the STM-based implementation

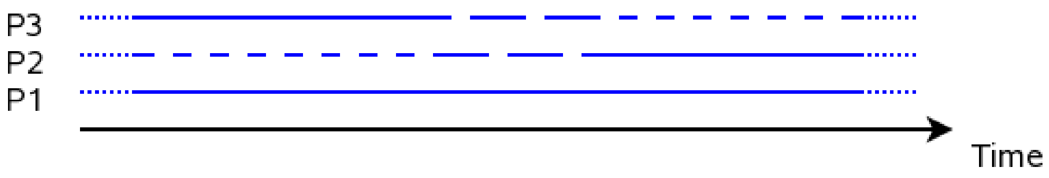


Figure 6.8: Schema of the structure of histories produces by the lock-based implementation

Another interesting result is the verification time in dependence on the history structure. Figure 6.6 shows the verification times for similar test scenarios as before but now for the locking implementation. The observation is that here, the systematic approach performs even better than before whereas the brute-force approach performs worse.

This can be explained by having a closer look at the structure of histories generated by STM- and locking-based implementations as illustrated in Figure 6.7 and 6.8, respectively. The lines represent the time intervals of operations when they are executed. While in the STM-based implementation all intervals are roughly of the same length, the intervals in the locking-based implementation can be very short but also very long. There are many sequences of short intervals. In a locking-based history it is very likely, that the correct order of operations is the one where the sequences occur logically before the long running parallel operations. Hence, this is almost the worst case for the brute-force approach because it will try most of the other possibilities first. However, for the systematic approach, it is even advantageous if there are sequential parts in the history because they are processed pretty easily by the method.

Although it is an interesting observation that the verification time depends on the history structure, it does not change the fact, that horizontally, there is an exponential growth for the brute-force approach whereas the systematic approach grows linearly concerning time consumption. The intuitive reason for this exponential behavior can be given by having a look at the following simple example.

Let us consider a history $H_1 = \{o_3, o_4, o_5\}$ where o_3 and o_4 are parallel and the right order of those two operations is resolved by considering o_5 which follows strictly sequentially after the two parallel operations. In the worst case, the brute-force verifier will have to traverse two orders which are

1. o_3, o_4, o_5

2. o_4, o_3, o_5

If the history is extended by another two parallel operations o_1 and o_2 which strictly occur before o_3 and o_4, but now o_5 deciding the order of the two new operations, then we already have four orders in the worst case. For further two operations we then have 8 orders etc. Of course, the worst case is not always the actual case. Moreover, the probability that an operation has a critical impact on the order of an operation pair that occurred much earlier is lower for higher distances between the deciding operation and the pair. Nevertheless, with increasing numbers of operations in a history these cases occur more often and this is the reason for the exponential growth that can be observed. The systematic approach, however, keeps all possible orderings in memory and hence there is no big difference for when the right order is actually decided.

6.3.2 Vertical Analysis

The results in the previous subsection have already given an impression of the advantages of the systematic approach for increasing sizes of histories that are verified. The most interesting question is how the complexity grows for increasing numbers of threads.

The Figures 6.9 and 6.10 show the graphs for increasing numbers of threads on a logarithmic scale. There is one graph for 200k operations executed by the locking implementation and another for 300k operations by the STM implementation. The brute-force exhibits an even worse behavior than in the horizontal analysis. The time consumption increases in a factorial manner which is even worse than an exponential increase. The values for 6 threads were not obtained because a verification has been aborted after a time span of more than 72 hours for both depicted scenarios.

The systematic approaches also show a factorial increase of time consumption. However, the practicality is obvious compared to the brute-force approach. In Figure 6.9 the systematic approaches are faster than the brute-force approach by a factor of 10 for 4 threads, of 100 for 5 threads, and of a higher factor than $10,000$ for 6 threads.

The optimized approach also shows a slight advantage over the standard version by $5 - 20\%$. However, for lower numbers of threads the overhead produced by the hash value computation leads to slightly worse results.

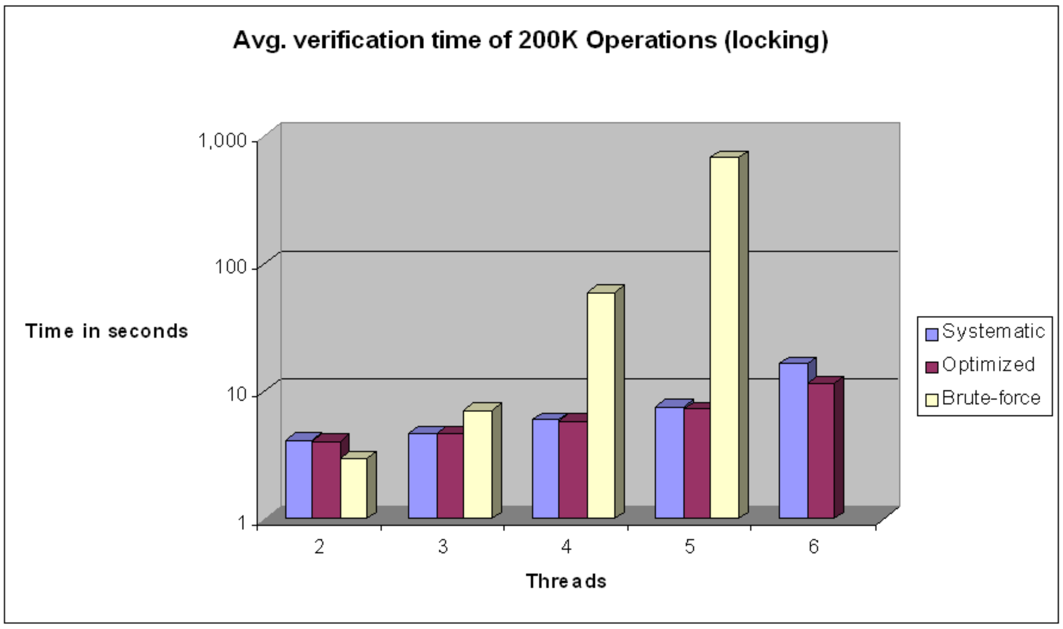


Figure 6.9: Vertical chart for 200k operations of the locking implementation (verification time)

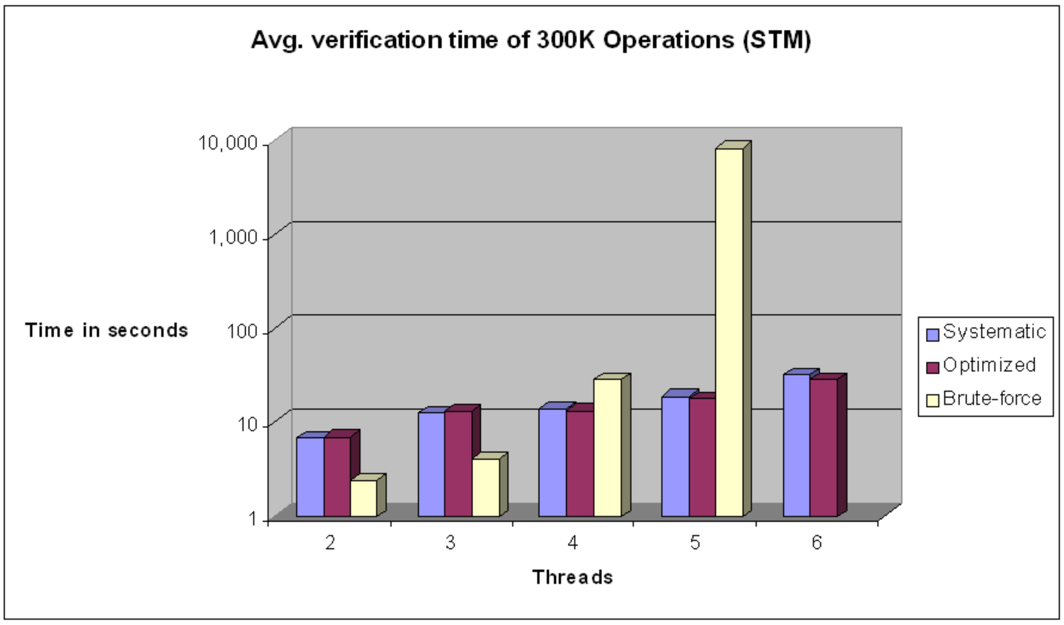


Figure 6.10: Vertical chart for 300k operations of the STM implementation (verification time)

6.3.3 Diagonal Analysis

The diagonal analysis is a summarization of the vertical and horizontal analysis. After going through the first two analyses the result of the diagonal consideration is foreseeable.

Figure 6.11 shows the graph for increasing numbers of threads but constant numbers of operations per thread. Again, the case for the brute-force approach for 6 threads is not included for the same reason as in the vertical analysis.

Figure 6.12 summarizes the preprocessing times. There are small deviations observable for the standard and optimized version of the systematic approach. These deviations can be explained by the influences of the operating system since the time for loading the file that contains the target history is included in the results.

6.3.4 Verification of Incorrect Histories

A structured approach for measuring the verification time of incorrect histories has not been used. The reason for this is that the results for such an investigation deviate strongly depending on when the error actually occurs. However, it is obvious that the systematic approach finds errors in a history quicker than it proves the correctness of a history, because if there is an error, not the entire history is traversed.

The brute-force approach works the opposite way. An incorrect history forces the brute-force approach to try all possible operation orderings, whereas a correct order can be found before traversing them all. Consequently, this is an issue that favors our approach even more. This issue was confirmed by randomly chosen incorrect histories which have been generated by the STM-based implementation during its debugging time.

6.4 Other Issues

The following two issues also have to be considered for a complete evaluation of the method. Space requirements have to be taken into account as well as the actual expressiveness of a test run.

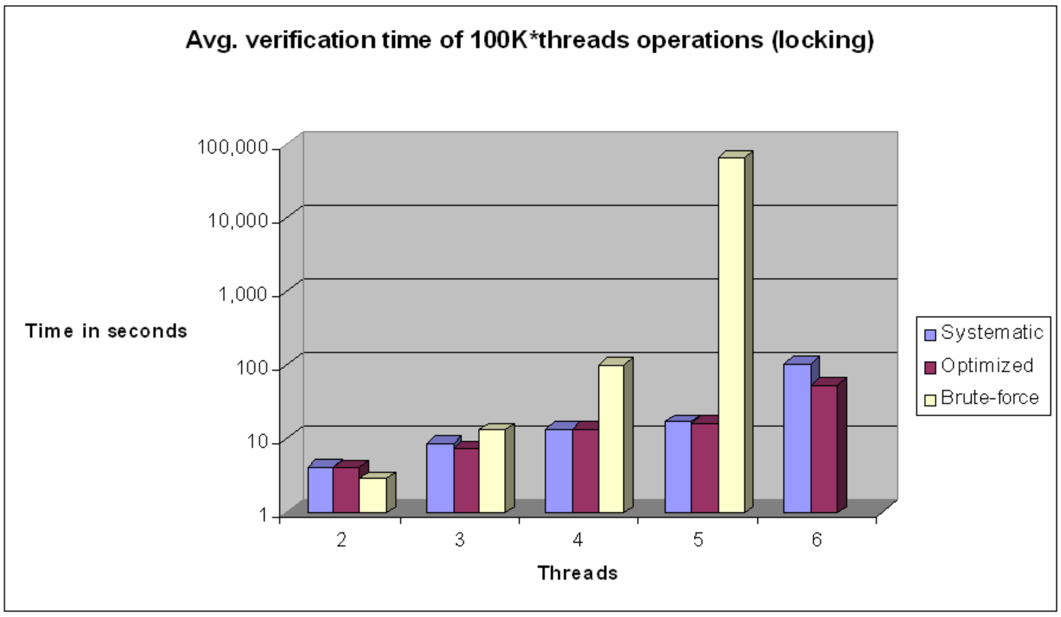


Figure 6.11: Diagonal chart for the locking implementation (verification time)

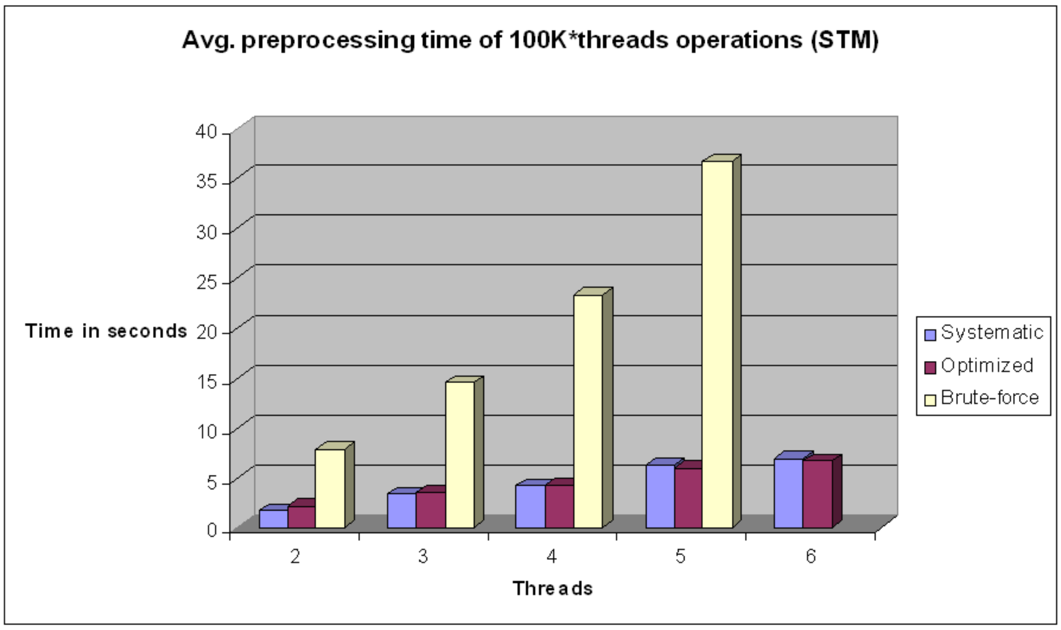


Figure 6.12: Diagonal chart for the STM implementation (preprocessing time)

6.4.1 Space Requirements

As we have seen in the previous section, the performance of the introduced verification method is for some cases better than the brute-force approach by multiple orders of magnitude. However, this gain of performance is not obtained for free.

While in principle the brute-force method only stores the currently tested execution path, the introduced method has to store all currently valid alternatives of execution paths. The space requirement increases exponentially but can be reduced drastically by the considerations of equivalent states and the like as discussed in chapter 4 and implementation optimizations like the *Definite-State* in chapter 5. The observation here is that the space requirements increase by some factor around 3 per added thread. This is one of the biggest problems to solve and optimize in future work because currently often the limits are reached at 11 or 12 threads. The following itemization gives a very rough impression of how many metastates are kept in memory at maximum for different numbers of threads.

2 threads $\rightarrow \approx$ 10 states

3 threads $\rightarrow \approx$ 30 states

4 threads $\rightarrow \approx$ 100 states

5 threads $\rightarrow \approx$ 300 states

6 threads $\rightarrow \approx$ 1000 states

Still, many errors can be detected for test runs of up to 9 threads. Despite the space problem, there can be obtained many more practical results than in the brute-force approach. Histories generated by 8 threads where still verified in feasible time.

6.4.2 Error Detection Effectiveness

Since this method does not prove the correctness of an SUT but only of executions of it, there is never a real certainty of whether the implementation of the SUT is really correct. However, under certain assumptions it is possible to give a probability analysis of how likely an error is to occur if it exists.

While the probability for an error to occur increases linearly with the number of operations executed, it is, however, very difficult to estimate how it increases for the number of threads without any knowledge of the concrete implementation. The overlaps of operations do not necessarily follow a regular pattern as seen before. It is also important that the SUT produces sane histories that cover as much of the parameter and state space as possible. This is part of other research that deals with the generation of complete test scenarios.

Therefore, a simple assumption of the likeliness of errors occurring is used. Figure 6.13 shows the probabilities for finding an error with increasing numbers of verified operations for the following scenarios

- an error occurs once every million operations
- an error occurs once every three million operations
- an error occurs once every ten million operations
- an error occurs once every thirty million operations
- an error occurs once every a hundred million operations

The first value has been chosen out of own practical experience during the development time of parallel code. Based on the verification, an error that occurred once every million operations was a common case. Hence, this value and higher ones are considered for the following analysis. For a probability of an error occurring of $P_{error} = \frac{1}{x}\%$ and a number of verified operations y, the probability P_{detect} of detecting an error is

$$P_{detect} = 1 - (1 - \frac{1}{x})^y \tag{6.1}$$

For big x and $y = a \cdot x$, we have

$$P_{detect} \approx 1 - \lim_{x\to\infty}(1 - \frac{1}{x})^{ax} = 1 - e^{-a} \tag{6.2}$$

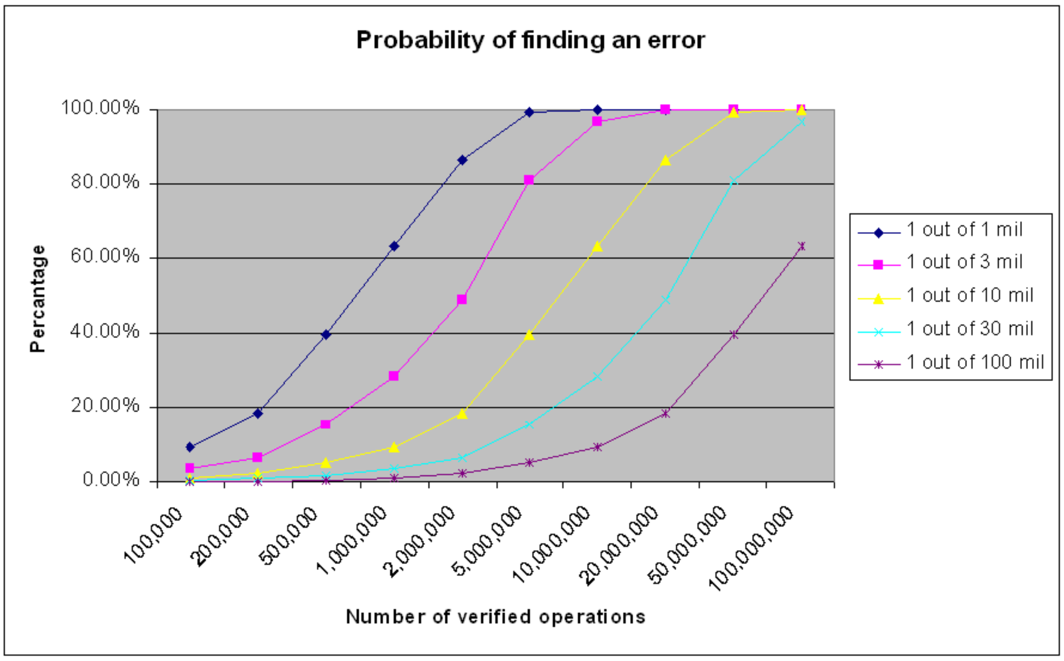


Figure 6.13: Probability for finding errors with different assumptions

Table 6.3: Values of P_{detect} for different a's

a	1	2	3	4	5	6
$1 - e^{-a}$	63.21%	86.47%	95.02%	98.17%	99.33%	99.75%

Table 6.3 shows how P_{detect} changes for different a. With this information it is also possible to interpret a positive verification for y operations. If y operations have been verified to be correct, we can say that the probability that the implementation has an error probability of at most $\frac{1}{y}$ is $P(P_{error} \leq \frac{1}{y}) \approx \frac{1}{e} \approx 36.79\%$. In the general case, we have

$$P(P_{error} \leq \frac{a}{y}) \approx e^{-a} \tag{6.3}$$

It strongly depends on the use case, which error detection probability is necessary or which error probability is acceptable. This requires an analysis for each scenario and is also an issue in other areas for example observation of memory errors in hardware [ScPW09, ScGi06, LiSH07].

Note, that the error detection effectiveness can be increased by defining sensible test cases. However, this is not in the scope of this thesis. There exists already work in the area of test generation [EFNR+01, Eyta06, MMPK+07].

6.4.3 History Shape

In a test run it is virtually impossible to start all threads at exactly the same time. Consequently, it takes n_1 operations for a test run with m logged operations to evolve an execution pattern as defined by the test scenario. The same is true for the end of a test run. Since all threads execute the same number of operations, some threads will be finished earlier than other threads. Let assume n_2 is the number of operations that are executed after the first thread has finished its last operation. Then $n = n_1 + n_2$ is the number of operations that are not executed in the way that was defined by the test scenario. n is bigger for shorter operations and higher numbers of threads. Thus, a history should be as long as possible so that n becomes negligible. Since this section shall only give a feeling for the expressiveness of a verification, in the previous detection effectiveness we assumed for simplicity $n = 0$. Nevertheless, the issue described here should be taken into account in practice.

6.5 Summary

This chapter has given an evaluation of different aspects of the introduced method. First, it showed performance numbers of the new method compared to a brute-force approach. The result is that the use of the new method is much more practicable than brute-force. This gain of performance is bought through higher space consumption. This chapter closed by showing the expressiveness of using a post-execution verification. It is necessary to verify an adequate number of operations assuming that the error probability is equally distributed.

7. Summary & Outlook

This dissertation has introduced a new post-execution verification method for parallel programs against linearizability. The motivation of this dissertation was to have a tool that gives an indication of the correctness of complex parallel implementations which are difficult or impossible to be treated with known methods. This tool is valuable to complex concurrent implementations because it is the only practical solution that is currently existing for these kind of programs.

For understanding the area this work is placed in, chapter 2 explained the difficulties in developing concurrent code. Therefore, it detailed a refined concurrent STM-based implementation that has been developed in the scope of this dissertation. Then, after giving a detailed introduction to correctness reasoning and introducing the term linearizability and its known verification methods, the core work of this thesis was treated in chapter 4 which is the theoretical base for the introduced method. All contributions of that and the following chapters are enumerated in the next section. It picks up the claims that were defined in section 1.4. In the last section, future research that could be done for continuing the work in this dissertation will be proposed.

7.1 Contributions

In the first chapter the general contributions of this dissertation has been announced. This section picks up on them and summarizes how those contributions have been realized.

1. New state representation

The brute-force verifier for parallel programs of today usually operate on objects. This lowers the flexibility of these kind of applications because there exist numerous data structures that are based on primitive values instead of objects. Therefore, another view on the state of a system is introduced by the metastate *construct which also allows for operating on primitive values.*

The construct *metastate* has been introduced that considers pending and completed operations at a certain point of verification time instead of all concrete states. For handling primitive values during a verification it is necessary to wrap those values into objects as a meta-construct for being able to distinguish them. If multiple occurrences of primitive values in a data structure are not distinguished properly, it is not possible to reproduce the behavior of a parallel program. *Metastates* provide such an implicit meta-construct. Furthermore, *metastates* can be compressed in such a way that it becomes more practical to consider them instead of concrete system states.

2. *Metastate* compression

In state exploration algorithms, one of the biggest problems is the state space explosion for parallel programs. Therefore, the newly introduced metastate *construct is elaborated for potential compression of its representation such that there is a huge gain for the verification performance in using it.*

The theory on *metastates* has been studied such that a compression of the realization of the *metastates* could be accomplished. Potential for compression has been worked out by regarding permutable operations, equivalent and redundant states. The gains of this part of the work are that the big issue of space requirements can be reduced to a feasible level. These observations have been worked into the theory of the method, so that implementations can take advantage of this knowledge for a more practical operability.

3. New verification methodology

The core work is a formal description of the applied verification methodology. It proves the validity of the method. It is the first non-brute-force method that can be used for linearizability checks which does not need any backtracking within *the verification procedure whereas brute-force approaches go back and forth in the execution stream.*

Basic terms have been given and defined like the *metastate* and the *event function* which later on have been used as theoretic fundamentals for the introduced method. With the aid of these definitions, the validity of the method has been proven. The method stores all possible states at a certain point of virtual time in the memory such that there is no backtracking necessary. The advantage of having no backtracking is that the history can be processed in a linear manner and program errors that occurred can be identified easier. We learnt that there is a practical possibility to find failures in an output and to locate them exactly instead of merely saying that a program is non-linearizable.

4. Generic verifier framework

The implementation of a generic verifier framework provides interfaces that allow a wide range of use cases to be implemented and execute a verification against it. It also gives a generic base implementation that can be used by verifiers using this method. This generic verifier framework is the first step towards a realization of the introduced methodology.

A generic verifier framework has been described that delivers interfaces for many different use cases to execute a verification against it. The base implementation of the verifier contains mostly the data structures that store all state alternatives, whereas the interfaces enable the opportunity to provide specifications and operation handling. The results indicate that this method is already applicable for a large space of programs. Furthermore, the results show that this method can provide outstanding performances compared to a brute force approach which has been in the only used approach up to now.

5. Foundation for systematic state pruning

There is given concrete implementations for two different use cases that have already been verified before which are the list and the set. However, the verification tools implemented here are more generic than the known ones because it can operate merely on keys of elements whereas other approaches needed keys and unique values of elements in collections. Furthermore, this method *does not require any backtracking which increases the chance for finding coding errors.*

The set and list use cases which already have been verified were also implemented according to the introduced methodology. This shows that with little effort this tool is applicable for different algorithms whereas other implementations of verifiers could only be used for only one certain specification. While in the literature each key was assigned to a value, the introduced method can do just by knowing the keys but not the value. Having a pair of a key and a unique values reduces the verification procedure to just finding the right operation pairs like *remove/insert* of a certain element and checking their timestamps. If only keys are present, there are more possible candidates of *insert* operations for a *remove* operation which increases the complexity. Hence, the introduced method allows for a more generic approach of the already known use cases and thus extends the already known verifiers of those data structures in this respect. It also does not use any kind of backtracking which makes error finding easier.

6. High-performance verification of the priority queue use case

Implementations for the priority queue use case have never been verified before. The implemented tool is the first verifier applied for priority queues and shows a high performance compared to a brute-force approach.

The priority queue use case has been verified which has never been done before. The implemented tool is the first that was capable of achieving a verification for this use case. This use case has been analyzed thoroughly and has been compared to a brute-force approach. The outcome of the analysis is that the introduced method has huge advantages compared to brute-force in most dimensions. Although, space can be an issue, the limits of the performance of the brute-force approach are reached much faster than the limits of space for the introduced method. This verification tool can provide a satisfactory confidence on implementations. It has also been used for debugging purposes during the development of priority queue implementations.

7. Fine-grained STM-based priority queue

A case study of a sophisticated software transactional memory-based data structure is given. This case study shows an example for a priority queue which made verification necessary for the development of a parallel implementation especially in the case of STM.

A sophisticated software transactional memory-based priority queue has been described in detail. The algorithm has many features that were not used before in an STM environment and thus is a very complex example of a parallel implementation which was treated by this verification method. This STM-implementation contains features that can be considered by expert programmers for their implementation purposes when trying to produce optimized parallel code based on STM. Despite the complexity of realizing those features, this implementation can still be treated by this verification method since the method does not consider implementation details but only its output.

7.2 Outlook

The work in this thesis is the initial work for a history-based verification tool. The ultimate goal is a tool that is capable of accepting a wide range of specifications and of performing a verification against them. For reaching this goal, there are some suggestions that could be picked up.

Optimizing space and time requirements

There are multiple optimizations that could be applied. Avoiding redundancies that occur from recursions is essential for optimizing the space requirements. More efficient storing strategies of the data structures used during the verification could also improve this aspect. A way of subdividing histories in a way that they can be verified in a multi-threaded manner would improve the time consumption especially for verifications dealing with histories generated by executions of a high number of threads.

Further analysis of the theory base

Furthermore, there could be more research done for the theoretical fundamentals such that more facts can be learned from relations among operations.

There are still many different characteristics of operations that can be investigated and treated in a general way. Especially arithmetic operations have not been dealt with intensively in this dissertation but rather collections. The difficulty with arithmetic operations is that operations altering the value of a variable often cannot be permuted generally. There are still open questions how for example a simple increment could be treated in an efficient way.

Generating histories with high state space coverage

Another issue that has to be further dealt with is how histories can be generated from a system under test such that the probability of an error to occur increases if existent. A history should contain as many different execution orderings as possible and the space of global states of the system should be ideally covered completely. There has been already done a lot of work in the area of test case generation [EFNR$^+$01, Eyta06, MMPK$^+$07]. In this work values have been generated randomly. Another possibility is to produce a execution simulator of the verified code that produces artificially steered histories such that the possible state space is covered in a more reliable way.

Implementing generic specifications

Ultimately, it is very motivating to have a generic verifier that reads a specification file for example in XML and provides verifier code that can be used. With such an interpreter it would be no longer necessary to implement the operation handlers for each use case which makes it very convenient for any kind of user.

If the issues enumerated here are tackled properly, there is a potential for very powerful verification tools that provide accurate and practical results for almost any arbitrary piece of software. Software development could be well supported by this kind of tools until source code verification tools become advanced enough for complex software. The work in this thesis is a first step towards a generic verification methodology against linearizability and shall encourage more research in this area such that parallel code can be developed more efficiently and with higher quality.

References

[ABCG$^+$06] Krste Asanovic, Ras Bodik, Bryan C. Catanzaro, Joseph J. Gebis, Parry Husbands, Kurt Keutzer, David A. Patterson, William L. Plishker, John Shalf, Samuel W. Williams und Katherine A. Yelick. The landscape of parallel computing research: a view from Berkeley. Technical Report UCB/EECS-2006-183, Electrical Engineering and Computer Sciences, University of California at Berkeley, December 2006.

[ABDD$^+$07] Alex Aiken, Suhabe Bugrara, Isil Dillig, Thomas Dillig, Brian Hackett und Peter Hawkins. An overview of the saturn project. In *PASTE '07: Proceedings of the 7th ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering*, New York, NY, USA, 2007. ACM, S. 43–48.

[AHMQ$^+$98] Rajeev Alur, Thomas A. Henzinger, Freddy Y. C. Mang, Shaz Qadeer, Sriram K. Rajamani und Serdar Tasiran. MOCHA: Modularity in Model Checking. In *CAV'98: Tenth International Conference on Computer-aided Verification*. Springer-Verlag, 1998, S. 521–525.

[Alph96] Alpha Architecture Handbook, Version 3, October 1996.

[Amda67] G. Amdahl. The Validity of the Single Processor Approach to Achieving Large Scale Computing Capabilities. *AFIPS Conference Proceedings* Band 30, 1967, S. 483–485.

[Ayan90] Rassul Ayani. LR-Algorithm: Concurrent Operations on Priority Queues. In *Proceedings of the Second IEEE Symposium on Parallel and Distributed Processing*. IEEE, 1990, S. 22–25.

[BBCL$^+$06] Thomas Ball, Ella Bounimova, Byron Cook, Vladimir Levin, Jakob Lichtenberg, Con McGarvey, Bohus Ondrusek, Sriram K. Rajamani und Abdullah Ustuner. Thorough static analysis of device drivers. *SIGOPS Oper. Syst. Rev.* 40(4), 2006, S. 73–85.

[BeHG86] Philip A. Bernstein, Vassos Hadzilacos und Nathan Goodman. *Concurrency control and recovery in database systems*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA. 1986.

[BHJM07] Dirk Beyer, Thomas A. Henzinger, Ranjit Jhala und Rupak Majumdar. The software model checker Blast: Applications to software engineering. *Int. J. Softw. Tools Technol. Transf.* 9(5), 2007, S. 505–525.

[BrLi89] Stefan Brass und Udo W. Lipeck. Specifying Closed World Assumptions for Logic Databases. In *MFDBS '89: Proceedings of the 2nd Symposium on Mathematical Fundamentals of Database Systems*, London, UK, 1989. Springer-Verlag, S. 68–84.

[ChSh04] Hao Chen und Jonathan S. Shapiro. Using build-integrated static checking to preserve correctness invariants. In *CCS '04: Proceedings of the 11th ACM conference on Computer and communications security*, New York, NY, USA, 2004. ACM, S. 288–297.

[Cous07] Patrick Cousot. Proving the absence of run-time errors in safety-critical avionics code. In *EMSOFT '07: Proceedings of the 7th ACM & IEEE international conference on Embedded software*, New York, NY, USA, 2007. ACM, S. 7–9.

[Dijk68] Edsger W. Dijkstra. Cooperating sequential processes. 1968.

[Dijknd] Edsger W. Dijkstra. Over seinpalen. circulated privately, n.d.

[DiSS06] Dave Dice, Ori Shalev und Nir Shavit. Transactional Locking II. In *DISC'06: Proceedings of the 20th International Symposium on Distributed Computing*, 2006, S. 194–208.

[DrBa08] Kristijan Dragičević und Daniel Bauer. A survey of concurrent priority queue algorithms. In *IPDPS'08: 22nd IEEE International Symposium on Parallel and Distributed Processing.* IEEE, 2008, S. 1–6.

[DrBa09] Kristijan Dragičević und Daniel Bauer. Optimization Techniques for Concurrent STM-Based Implementations: A Concurrent Binary Heap as a Case Study. In *IPDPS'09: 23nd IEEE International Symposium on Parallel and Distributed Processing.* IEEE, 2009, S. 1–8.

[DrBGE10] Kristijan Dragičević, Daniel Bauer und Luis Garcés-Erice. System and method for demonstrating the correctness of an execution trace in concurrent processing environments. Patent application number: 20100205484, December 2010.

[EFNR+01] Orit Edelstein, Eitan Farchi, Yarden Nir, Gil Ratsaby und Shmuel Ur. Multithreaded Java program test generation. In *JGI '01: Proceedings of the 2001 joint ACM-ISCOPE conference on Java Grande*, New York, NY, USA, 2001. ACM, S. 181.

[EnAs03] Dawson Engler und Ken Ashcraft. RacerX: effective, static detection of race conditions and deadlocks. *SIGOPS Oper. Syst. Rev.* 37(5), 2003, S. 237–252.

[Enna06] Robert Ennals. Software transactional memory should not be obstruction-free. Technical Report, Intel Research, 2006.

[Eyta06] Yaniv Eytani. Concurrent Java Test Generation as a Search Problem. *Electron. Notes Theor. Comput. Sci.* 144(4), 2006, S. 57–72.

[FiLP85] Michael J. Fischer, Nancy A. Lynch und Michael S. Paterson. Impossibility of distributed consensus with one faulty process. *Journal of the ACM* 32(2), 1985, S. 374–382.

[FLLN$^+$02] Cormac Flanagan, K. Rustan M. Leino, Mark Lillibridge, Greg Nelson, James B. Saxe und Raymie Stata. Extended static checking for Java. In *PLDI '02: Proceedings of the ACM SIGPLAN 2002 Conference on Programming language design and implementation*, New York, NY, USA, 2002. ACM, S. 234–245.

[FLMS05] Faith Ellen Fich, Victor Luchangco, Mark Moir und Nir Shavit. Obstruction-Free Algorithms Can Be Practically Wait-Free. In *DISC'05: Distributed Computing, 19th International Conference*, Band 3724. Springer, 2005, S. 78–92.

[Flyn72] Michael J. Flynn. Some Computer Organizations and Their Effectiveness. *IEEE Transactions on Computers* 21(9), September 1972, S. 948–960.

[Fras03] Keir Fraser. *Practical lock freedom.* Dissertation, Cambridge University Computer Laboratory, 2003.

[Gode97] Patrice Godefroid. Model checking for programming languages using VeriSoft. In *POPL '97: Proceedings of the 24th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, New York, NY, USA, 1997. ACM, S. 174–186.

[GoJS96] James Gosling, Bill Joy und Guy L. Steele. *The Java Language Specification.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA. 1996.

[GoYS04] Ganesh Gopalakrishnan, Yue Yang und Hemanthkumar Sivaraj. QB or not QB: An efficient execution verification tool for memory orderings. In *In Computer-Aided Verification (CAV), LNCS 3114*, 2004, S. 401–413.

[GSVW$^+$09] Justin E. Gottschlich, Jeremy G. Siek, Manish Vachharajani, Dwight Y. Winkler und Daniel A. Connors. An efficient lock-aware transactional memory implementation. In *ICOOOLPS '09: Proceedings of the 4th workshop on the Implementation, Compilation, Optimization of Object-Oriented Languages and Programming Systems*, New York, NY, USA, 2009. ACM, S. 10–17.

[Gust88] John L. Gustafson. Reevaluating Amdahl's law. 31(5), 1988, S. 532–533.

[HaFr03] Tim Harris und Keir Fraser. Language support for lightweight transactions. *SIGPLAN Notifications* 38(11), 2003, S. 388–402.

[Hans75] Per Brinch Hansen. The Programming Language Concurrent Pascal. *IEEE Trans. Software Eng.* 1(2), 1975, S. 199–207.

[HeMo93] Maurice Herlihy und J. Eliot B. Moss. Transactional Memory: Architectural Support for Lock-Free Data Structures. In *Proceedings of the 20th Annual International Symposium on Computer Architecture*. ACM Press, Mai 1993, S. 289–300.

[Herl88] Maurice P. Herlihy. Impossibility and universality results for wait-free synchronization. In *PODC '88: Proceedings of the seventh annual ACM Symposium on Principles of distributed computing*, New York, NY, USA, 1988. ACM, S. 276–290.

[Herl03] Maurice Herlihy. Obstruction-free synchronization: Double-ended queues as an example. In *ICDCS'03: In Proceedings of the 23rd International Conference on Distributed Computing Systems*. IEEE Computer Society, 2003, S. 522–529.

[HeWi90] Maurice P. Herlihy und Jeannette M. Wing. Linearizability: a correctness condition for concurrent objects. *ACM Trans. Program. Lang. Syst.* 12(3), 1990, S. 463–492.

[HLMI03] Maurice Herlihy, Victor Luchangco, Mark Moir und William N. Scherer III. Software transactional memory for dynamic-sized data structures. In *PODC '03: Proceedings of the twenty-second annual symposium on Principles of distributed computing*, New York, NY, USA, 2003. ACM, S. 92–101.

[HMPS96] Galen Hunt, Maged M. Michael, Srinivasan Parthasarathy und Michael L. Scott. An Efficient Algorithm for Concurrent Priority Queue Heaps. *Inf. Proc. Letters* Band 60, 1996, S. 151–157.

[Hoar74] Charles Antony Richard Hoare. Monitors: an operating system structuring concept. *Commun. ACM* 17(10), 1974, S. 549–557.

[Hoar83] Charles Antony Richard Hoare. An axiomatic basis for computer programming. *Communications of the ACM* 26(1), 1983, S. 53–56.

[HuAh90] Phillip Hutto und Mustaque Ahamad. Slow Memory : Weakening Consistency to Enhance Concurrency in Distributed Shared Memories. *Proceedings of Tenth International Conference on Distributed Computing Systems*, 1990.

[Jone83] Cliff B. Jones. Specification and Design of (Parallel) Programs. In *Proceedings of IFIP'83*. North-Holland, 1983, S. 321–332.

[JrGP99] Edmund M. Clarke Jr., Orna Grumberg und Doron A. Peled. *Model Checking*. MIT Press. 1999.

[Lamp79] L. Lamport. How to Make a Multiprocessor Computer That Correctly Executes Multiprocess Progranm. *IEEE Trans. Comput.* 28(9), 1979, S. 690–691.

[LCLS09] Yang Liu, Wei Chen, Yanhong A. Liu und Jun Sun. Model Checking Linearizability via Refinement. In *FM 2009: Formal Methods, Second World Congress, Eindhoven, The Netherlands, November 2-6, 2009. Proceedings*, 2009, S. 321–337.

[LiSH07] Xin Li, Kai Shen und Michael C. Huang. A memory soft error measurement on production systems. In *In USENIX Annual Technical Conference*, 2007.

[Mich04] Maged M. Michael. Hazard Pointers: Safe Memory Reclamation for Lock-Free Objects. *IEEE Transactions on Parallel and Distributed Systems* 15(6), 2004, S. 491–504.

[MMPK$^+$07] Saša Misailović, Aleksandar Milićević, Nemanja Petrović, Sarfraz Khurshid und Darko Marinov. Parallel test generation and execution with Korat. In *ESEC-FSE '07: Proceedings of the the 6th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering*, New York, NY, USA, 2007. ACM, S. 135–144.

[OwGr76] Susan S. Owicki und David Gries. An axiomatic proof technique for parallel programs. *Acta Informatica* Band 6, 1976, S. 319–340.

[Pugh90] William Pugh. Skip lists: a probabilistic alternative to balanced trees. *Commun. ACM* 33(6), 1990, S. 668–676.

[SATHM$^+$06] Bratin Saha, Ali-Reza Adl-Tabatabai, Richard L. Hudson, Chi Cao Minh und Benjamin Hertzberg. McRT-STM: a high performance software transactional memory system for a multicore runtime. In *PPoPP '06: Proceedings of the eleventh ACM SIGPLAN symposium on Principles and practice of parallel programming*, New York, NY, USA, 2006. ACM, S. 187–197.

[ScGi06] Bianca Schroeder und Garth A. Gibson. A Large-Scale Study of Failures in High-Performance Computing Systems. In *In Proceedings of the 2006 International Conference on Dependable Systems and Networks (DSN'06)*, 2006, S. 249–258.

[ScPW09] Bianca Schroeder, Eduardo Pinheiro und Wolf-Dietrich Weber. DRAM errors in the wild: a large-scale field study. In *SIGMETRICS '09: Proceedings of the eleventh international joint conference on Measurement and modeling of computer systems*, New York, NY, USA, 2009. ACM, S. 193–204.

[ShTo95] Nir Shavit und Dan Touitou. Software Transactional Memory. In *PODC'95: Proceedings of the 14th annual ACM symposium on Principles of distributed computing.* ACM Press, August 1995, S. 204–213.

[SMATB$^+$07] Tatiana Shpeisman, Vijay Menon, Ali-Reza Adl-Tabatabai, Steven Balensiefer, Dan Grossman, Richard L. Hudson, Katherine F. Moore und Bratin Saha. Enforcing isolation and ordering in STM. *SIGPLAN Notices* 42(6), 2007, S. 78–88.

[SuTs05] Håkan Sundell und Philippas Tsigas. Fast and lock-free concurrent priority queues for multi-thread systems. *J. Parallel Distrib. Comput.* 65(5), 2005, S. 609–627.

[TaSt07] Andrew S. Tanenbaum und Maarten Van Steen. *Distributed Systems: Principles and Paradigms.* Prentice Hall. 2007.

[TMGH$^+$09] Fuad Tabba, Mark Moir, James R. Goodman, Andrew W. Hay und Cong Wang. NZTM: nonblocking zero-indirection transactional memory. In *SPAA '09: Proceedings of the twenty-first annual symposium on Parallelism in algorithms and architectures*, New York, NY, USA, 2009. ACM, S. 204–213.

[Vafe07] Viktor Vafeiadis. *Modular fine-grained concurrency verification.* Dissertation, Cambridge University Computer Laboratory, 2007.

[VeYa08] Martin Vechev und Eran Yahav. Deriving linearizable fine-grained concurrent objects. *SIGPLAN Notices* 43(6), 2008, S. 125–135.

[ViIS05] J. Marathe Virendra, William N. Scherer III und Michael L. Scott. Adaptive Software Transactional Memory. In *In Proceedings of the 19th International Symposium on Distributed Computing*, 2005, S. 354–368.

[Vola06] Nic Volanschi. A Portable Compiler-Integrated Approach to Permanent Checking. In *ASE '06: Proceedings of the 21st IEEE/ACM International Conference on Automated Software Engineering*, Washington, DC, USA, 2006. IEEE Computer Society, S. 103–112.

[WiGo93] Jeannette M. Wing und Chun Gong. Testing and verifying concurrent objects. *J. Parallel Distrib. Comput.* 17(1-2), 1993, S. 164–182.

[XiCE03] Yichen Xie, Andy Chou und Dawson Engler. ARCHER: using symbolic, path-sensitive analysis to detect memory access errors. In *ESEC/FSE-11: Proceedings of the 9th European software engineering conference held jointly with 11th ACM SIGSOFT international symposium on Foundations of software engineering*, New York, NY, USA, 2003. ACM, S. 327–336.

[YuVa02] Haifeng Yu und Amin Vahdat. Design and evaluation of a conit-based continuous consistency model for replicated services. *ACM Trans. Comput. Syst.* 20(3), 2002, S. 239–282.

[Zö83] Dieter Zöbel. The Deadlock problem: a classifying bibliography. *SIGOPS Oper. Syst. Rev.* Band 17, October 1983, S. 6–15.

Index