Specification and Verification of Synchronisation Classes in Java

A Practical Approach

Afshin Amighi

Specification ★ Verification of Synchronisation Classes in Java

A Practical Approach

Afshin Amighi

SPECIFICATION AND VERIFICATION OF SYNCHRONISATION CLASSES IN JAVA

A PRACTICAL APPROACH

DISSERTATION

to obtain
the degree of doctor at the University of Twente,
on the authority of the rector magnificus,
prof.dr. T.T.M. Palstra,
on account of the decision of the graduation committee,
to be publicly defended
on Wednesday 17 January 2018 at 16:45 hrs.

by

Afshin Amighi
born on 22 June 1976
in Tabriz, Iran

This dissertation has been approved by:

Supervisor: Prof.dr. M. Huisman

CTIT

CTIT Ph.D. Thesis Series No. 17-451
Centre for Telematics and Information Technology
University of Twente, The Netherlands
P.O. Box 217 – 7500 AE Enschede, The Netherlands

IPA Dissertation Series No. 2018-01
The work in this thesis has been carried out under the auspices of the research school IPA (Institute for Programming research and Algorithmics).

European Research Council
Established by the European Commission

European Research Council
The work in this thesis was supported by the VerCors project (Verification of Concurrent Data Structures), funded by ERC grant 258405.

ISBN: 978-90-365-4439-9
ISSN: 1381-3617 (CTIT Ph.D. Thesis Series No. 17-451)
DOI: 10.3990/1.9789036544399
Available online at https://doi.org/10.3990/1.9789036544399

Typeset with LaTeX.
Cover design and printed by: Gildeprint.
Original image of the cover under license from Shutterstock.com.
Copyright © 2018 Afshin Amighi, The Netherlands.

Graduation Committee:

Chairman:	Prof.dr. P.M.G. Apers	University of Twente
Supervisor:	Prof.dr. M. Huisman	University of Twente

Members:

Prof.dr.ir. J.C. van de Pol	University of Twente
Prof.dr.ir. H. Wehrheim	Paderborn University
Prof.dr. A. van Deursen	Technical University of Delft
Prof.dr.ir. A. Pras	University of Twente
Dr.ir. V. Vafeiadis	Max Planck Institute for Software Systems

تقدیم به آموزگار

زندگی، رادین؛ عشق، الهام؛

فداکاری و تلاش، سیمین و رضا؛ مهربانی، نوشین و فرین

Acknowledgements

On March 2011, when I started my journey of “ *PhD candidate* ”, to me, it seemed like a magic to finish it. Now the journey is about to finish and the magic is going to happen. I owe this to my kind and supportive supervisors, colleagues, friends and family members. This work would not have been possible without the valuable assistance of them. Here, I would like to devote this space to thanking those who have walked alongside me during this journey and made the magic happen.

To *Jaco van de Pol* and *Marieke Huisman*: thanks for providing me this wonderful opportunity of being a member of the FMT family. I learned, grew and enjoyed every moment of working there.

To my excellent supervisor, *Marieke Huisman*: I certainly would not accomplish this without your patient guidance and useful critiques of my research. I have been always impressed by the amount of the papers, reviews, writings, meetings, trips and other work that you always manage to handle and still within the deadlines you are able to be critical about the work with helpful comments. You were not only my supervisor, but also a kind friend. You have been always helpful, caring, encouraging and understanding to me and my family.

I am also grateful to European Research Council (ERC), who funded this work via the VerCors project.

To my committee members, *Jaco van de Pol*, *Viktor Vafeiadis*, *Heike Wehrheim*, *Arie van Deursen*, and *Aiko Pras*: I am appreciative of you for evaluating my thesis and be a member of my graduation committee.

To *Viktor Vafieadis*: thanks for providing me the chance to visit Max Plank Institute-Software Systems and for all the valuable discussions and comments.

To my project teammates, *Stefan Blom*, *Wojciech Mostowski*, *Marina Zaharieva-Stojanovski* and *Saeed Darabi*: I am so grateful for the support

and assistance provided by you. Thank you for all the generous advises. Discussing with you about the challenges has been always constructive and valuable to me. This thesis truly is the result of all those daily discussions.

To my creative and smart FMT colleagues, *Arend Rensik*, *Gijs Kant*, *Mariëlle Stoelinga*, *Amirhossein Ghamarian*, *Maarten de Mol*, *Eduardo Zambon*, *Tom van Dijk*, *Ngo Minh Tri*, *Ketema Jeroen*, *Axel Belinfante*, *Lesley Wevers*, *Stefano Schivo*, *Waheed Ahmad*: I am indebted to you all. I enjoyed every moment that we spent together. I will never forget all those small chats, coffee breaks, Friday afternoon talks, lunch colloquiums, campus runnings.

To my new FMT colleague, *Sebastiaan Joosten*: I had a great time discussing with you about the challenges provided in ICT with Industry Workshop-University of Leiden. Thanks for all the detailed and helpful feedback on my thesis.

To *Ida*, *Jeanette* and *Joke*: I am truly appreciated by the assistance provided by you. You have been always there to help me with all the steps that could accelerate required official procedures.

To my truly friends, *Mohammad* and *Shahin*: you have been always like a brother to me. You and your lovely families; Leila, Soude; have been always there to lend me and my family a hand without hesitate. Me and my family will never forget your assistance and support.

To my kind friends, *Hassan*, *Hamed*, *Meisam*, *Majid*, *Gerrie*, *Alireza*, *Saeed*, *Sara*, *Zahra*: thank you for all the warm and friendly gatherings.

To my amiable colleagues in RUAS, *Ahmad*, *Mohammed*, *Tony*, *Andrea*, *Anne*, *Hans*, *Marjon*, *Francesco*, *Giuseppe*, *Giulia*, *Hossein*, *Tanja*, *Albert*, *Jan*: thank you for providing me such a nice working environment.

To my kind in-laws, *Jalil*, *Farah* and *Behnam*: I owe a deep gratitude and thanks for your positive attitude and encouragement in all the steps. Thank you *Farah* for your support; the last steps were very difficult. You made it easy for me.

To my incredible parents, *Simin* and *Reza*: I cannot express anything that can show my gratefulness for your love and encouragement. You are always ready to provide me hope and motivation for my next step. I would never have enjoyed so many wonderful opportunities without your endless sacrifice.

To my lovely sisters, *Noushin* and *Farrin*: I am more than grateful to you. You were always kindly supporting my decisions. I hope I can spend

more time later with you and your beloved families: Kasra, Kia, Koosha and Reza.

I save the last for the best. To my **lifelong friend**, *Elham*: this would not be achieved without your support and patience. During this journey you have been always passionate, empathetic, persistent and considerate. We started the journey together, we both will finish it. I will practice my patience when you start writing your thesis in the coming months.

Abstract

Nowadays, concurrent programming is becoming a mainstream technique to achieve high-performance computing. Implementing a correct program using shared memory concurrency is challenging. This is because of the unpredictable interleavings of threads that may cause unaccepted access to the shared memory, known as *data race*. Programmers use *synchronisers* to control thread interleaving and prevent data races. Coarse-grained synchronisation constructs block threads for large parts of execution tasks, while fine-grained synchronisation primitives employ atomic operations.

Using testing techniques to catch possible errors in concurrent programs is not feasible, because the appearance of errors depends on the execution order of the participating threads, which varies in different runs. And moreover, testing techniques are able only to show the presence of errors. Formal techniques are needed to guarantee the absence of errors independently from execution environments. Axiomatic based approaches are one of the common techniques that can verify the correctness of a given program with respect to its specifications without executing the program. Axiomatic based program verification employs a collection of formal rules that expresses the correct behaviour of the program using a program logic. Program logic shows how to formally derive correctness of the implementation w.r.t. its specification.

Permission-based Separation Logic is a program logic that has been developed to reason about concurrent programs. In permission-based Separation Logic one can express and reason about the *ownership* of memory locations. This is an important and powerful feature of this logic because any *data race* becomes detectable. Verification techniques of concurrent programs based on permission-based Separation Logic exploit this feature of the logic to verify that a concurrent program is free from data races.

In this thesis, we propose an approach *to specify and verify synchronisation constructs* which are at the heart of any concurrent program. The class of synchronisers that we study here includes both *coarse-grained* and *fine-grained* synchronisers. In our approach we lift formalised rules of permission-based Separation Logic to the *specification* level. Our method offers a generalised, high-level and extendable approach to specify and verify arbitrary synchronisation constructs. Our approach is *a practical technique.* Using the VerCors tool set we demonstrate the practicality of our approach by verifying a variety of examples implemented in Java. The VerCors tool set and all the tool verified examples are available online.

In this thesis, first, we start with the basics of threads communication and synchronisation primitives in Java. As a result, we verify the correctness of a concurrent Java program where threads have multiple join points. Then, we study the general behaviour of commonly used synchronisation classes in Java. We propose a unified approach to specify the correct behaviour of synchronisers. Concretely, we discuss the formal specification of the synchronisation classes implemented in the java.util.concurrent package. This enables us to verify the correctness of the concurrent Java programs that are using these synchronisation classes. Next, in order to verify that the implementation of synchronisers satisfies our specifications we develop techniques to reason about the verification of synchronisers. A common way to implement such sychronisers is by using atomic operations. We identify different synchronisation patterns that can be implemented by using atomic operations. Moreover, we propose a specification of these operations in Java's AtomicInteger class, which is an essential class in the implementation of Java's synchronisers. We use our specification of the atomic operations to verify the implementation of both exclusive access and shared-reading synchronisation classes developed in java.util.concurrent.

Further to the results of reasoning about atomic operations, we propose a verification stack where several concurrent program verification techniques are combined. In each layer of the stack a particular property of a concurrent program is verified. In our three layer verification stack, the bottom layer, reasons about data race freedom of programs. The second layer reasons about properties that ties properties of both thread-local and shared variables. The top layer adds a notion of histories to reason about functional properties of concurrent data structures. We illustrate our technique on the verification of a lock-free queue and reentrant lock both implemented in Java.

Finally, we propose a specification and verification technique to reason about data race freedom and functional correctness of GPU kernels that use *atomic operations* as synchronisation mechanism.

Altogether, the techniques proposed in this thesis are a step forward towards the practical verification of non-trivial Java(-like) concurrent programs that are using either fine-grained or coarse-grained synchronisers. The verified programs are guaranteed to be free from data races and the verification is done with a rich specification language that relies on the rules developed by permission-based Separation Logic.

Contents

Abstract xiii

Contents xvii

Listings xix

1 Introduction 1
1.1 Concurrency 4
1.2 Synchronisation 7
1.3 Verification 10
1.4 A Practical Approach 11
1.5 Thesis 12

2 Technical Background 17
2.1 Atomic Variables in Java 19
2.2 Permission-based Separation Logic 20
2.3 VerCors Specification Language 25

3 Reasoning about Thread Creation and Termination 27
3.1 Reasoning about Dynamic Threads 29
3.2 Contract of Class Thread 32
3.3 Example: Multi-threaded Data Processing 33
3.4 Conclusion and Related Work 36

4 Synchronisers Specifications 41
4.1 Locks in Java 44
4.2 Semaphore Specification 51
4.3 CountDownLatch Specification 52

4.4 CyclicBarrier Specification . . . 57
4.5 Examples . . . 58
4.6 Conclusion and Related Work . . . 63

5 Verification of Synchronisers: Exclusive Access 67
5.1 Synchronisation Patterns . . . 70
5.2 Ownership Exchange via Atomics . . . 73
5.3 Specifications of Atomics . . . 78
5.4 Contracts of **AtomicInteger** . . . 86
5.5 Conclusion and Related Work . . . 99

6 Verification of Synchronisers: Shared Reading 101
6.1 Synchronisation Classes . . . 104
6.2 Reasoning about Atomics . . . 105
6.3 Contract of AtomicInteger . . . 109
6.4 Verification . . . 111
6.5 Conclusion and Related Work . . . 115

7 Multi-layer Verification based on Concurrent Separation Logic 117
7.1 Layer 1: Permissions and Resource Invariants . . . 121
7.2 Layer 2: Relating Thread-Local and Global Variables . . . 124
7.3 Layer 3: Functional Properties using Histories . . . 129
7.4 Conclusion and Related Work . . . 136

8 Specification and Verification of Atomic Operations in GPGPU Programs 139
8.1 Background . . . 142
8.2 Specification . . . 144
8.3 Formalization . . . 149
8.4 Conclusion and Related Work . . . 156

9 Conclusions 159

List of Publications 165

Bibliography 167

Listings

1	SampleThread class.	5
2	An implementation for a concurrent counter.	6
3	Counter class.	7
4	Intrinsic locking.	8
5	Explicit locking.	8
6	High-level data-race.	9
7	Atomic Increment.	10
8	The AtomicInteger class.	20
9	Specification of class Thread.	32
10	Class Buffer.	34
11	Class Sampler.	35
12	Class AFilter.	38
13	Class Plotter.	39
14	Verification of the main program.	40
15	Specification of the Lock interface.	47
16	Specification of the ReadWriteLock interface.	49
17	Specifications for lock initialization.	50
18	Specification of the Semaphore class.	51
19	Simplified specification of the CountDownLatch class.	54
20	Improved specification of the CountDownLatch class.	55
21	Specification of the CyclicBarrier class.	58
22	The producer.	59
23	The consumer.	60
24	Implementation of SProdMCons.	61
25	Client code using CountDownLatch	64
26	Client code using the CyclicBarrier	65
27	The processing code for the example in Listing 26.	66
28	ProducerConsumer: cooperation.	71

29 SpinLock: competition. 72
30 SingleCell: hybrid. 74
31 Contracts for AtomicInteger 90
32 Verification of SingelCell: constructor 94
33 Verification of SingelCell::findOrPut() 95
34 Veirication of ProducerConsumer 97
35 Verification of SpinLock. 98
36 Implementation of a Semaphore. 105
37 Implementation of a CountDownLatch. 106
38 Contracts for AtomicInteger: Exclusive and Shared 110
39 Verification of Semaphore: constructor. 113
40 Verification of Semaphore::acquire(). 114
41 Verification of Semaphore::release(). 115
42 CSL resource invariant of the lock-free queue. 123
43 Dequeue attempt. 125
44 Interface Lock. 127
45 The definition of the lock invariant in the Lock class. 128
46 The definition of the lockset predicate in the Thread class. . . . 129
47 The method that encodes creating a history 131
48 Fragment of History Specification for Queues. 133
49 History specification of the get method. 134
50 Specified run method of the receiver 135
51 An example of a kernel with specifications 143
52 Specification of parallel add in a work group. 145
53 Specification of global parallel add. 147

CHAPTER 1

Introduction

We are quickly moving towards a fully digitalized society. We cannot imagine anymore what life would be like without all the ICT-based services that we depend upon. We expect these services to be available 24/7, that they respond to all our needs immediately, and moreover that they do this in a reliably and trustworthy manner.

These expectations on availability and responsiveness lead to ever increasing demands on the performance of the software behind these services. For a long time, these increasing demands have been answered by increasing the speed of single processing units: according to Moore's famous law [64], the number of elements on a chip (and thus its processing speed) doubles approximately every two years. However, due to physical limitations, we are quickly approaching the maximal processor speed for single computing units, and Moore's law no longer applies. Instead, modern computing devices incorporate multiple processor units, *i.e.*, multi-core computing is becoming the new standard.

To make optimal use of a multi-core device, the software that runs on it needs to support parallelism, *i.e.*, it needs to provide support to divide the computational job in multiple tasks which can be performed simultaneously[1]. Moreover, it also often happens that the intended computational tasks are inherently parallel, and parallel (or concurrent) programming is the natural and efficient way to implement it. Therefore, many modern high-level programming languages provide built-in support, as well as extensive libraries, to support *concurrent programming*.

Despite the programming language support, in the end the division of the computational job into multiple parallel tasks is still the job of the programmer. Unfortunately, this task is error-prone, as it requires the programmer to make a mental model of all the different interactions that can exist between the different parallel threads of execution. As users do not only want high performance, but also that computations are reliable, non-crashing, and function correctly, we need techniques that help the programmer to avoid such errors. This is precisely the focus of this thesis.

In the remainder of this introduction, we first briefly discuss the main characteristics of shared memory concurrent programs, and then we de-

[1] Logically simultaneous processing is defined as concurrency and physically simultaneous processing is known as parallelism. In this thesis, concurrency and parallelism are used interchangeably.

scribe our approach to reason about the behaviour of concurrent programs. In shared memory concurrency, typically multiple parallel computations (threads of execution) are created that all make use of a single shared memory. If access to the shared memory is not properly controlled, a program has data races, and its behaviour can become unpredictable. In this thesis, we study *synchronisation* techniques to control access to shared memory, in order to avoid this unpredictability. Standard techniques for synchronisation are often based on locks. Their drawback is that they can negatively impact the performance of a program, because threads might have to wait for a long time before they are able to proceed. Therefore, more efficient concurrent programs use fine-grained synchronisation using atomic operations. We show how to reason about programs using atomics for synchronisation directly. We also discuss how to reason about other high-level synchronisation mechanisms that can be built on top of atomic synchronisation primitives. Our approach is based on static verification, *i.e.*, it works at compile-time. In this thesis, we focus mostly on Java(-like) programs, however, our techniques are also applicable for other programming languages with similar concepts.

1.1 Concurrency

In a concurrent program, a number of parallel processes operate concurrently to achieve a common task. In Java, a programmer can create such a parallel process, called thread, by creating an object that is an instance of a class that extends the `Thread` class[2]. Calling the `start()` method on this object causes the Java virtual machine to start the execution of a new thread, which executes the body of the `run()` method of the thread object. Listing 1 contains a very simple thread creation example.

A thread can wait for another thread to terminate by calling the `join()` method on this other thread. This blocks the caller (provided the callee thread has already been started) until the joined thread completes its execution.

A common way for threads to communicate with each other is via shared memory, which they can access and update. If a thread updates a variable in the shared memory, its new value is eventually made available to all other

[2]Another way of creating threads in Java is to implement the `Runnable` interface.

```
public class SampleThread extends Thread {
  public void run() { /* Task of the thread is implemented here. */ }
  public static void main(String[] args){
    (new SampleThread()).start();
  }
}
```

Listing 1: SampleThread class.

threads. The advantage of shared memory concurrency is that it is simple and intuitive. However, its downside is that the order in which different threads access the shared memory is not deterministic. This can cause a bug. If two threads access the same memory location, and at least one of these two accesses is a write operation, then we say that the accesses are *conflicting*. If the program does not have sufficient synchronisation to ensure that conflicting accesses cannot happen simultaneously, then we say that the program has a data race [68]. Note that if the program is sufficiently synchronised, the actual value stored in a variable might still depend on the execution speed of another thread. This is called a race condition (or high-level data race) [31] and might also indicate a bug in the program. However, in this thesis, we concentrate on how to prove absence of data races. Data races are considered to be bugs, and must be avoided. In particular, if a program with data races is executed on a relaxed memory model, compiler reorderings might be allowed which result in unexpected program executions.

As an example, Listing 2 shows a program[3] that creates a number of threads that concurrently increment the shared variable Counter::count, which is initialized to 0. If n threads are created[4], after termination of the count method, we expect that the final value of Counter::count is n. Listing 3 shows a candidate implementation of the incr() method. Clearly, in this implementation there is a conflicting access to the variable count, which causes a data race[5]: threads can interfere with each other while executing the instructions in lines 7 and 8.

For example, assume two threads t and t' are trying to increment count

[3] For simplicity of the example, exception handling is omitted.

[4] The number of threads in the example can be any positive number.

[5] This example shows both data races and race conditions.

```java
public class MainConcCounter{

  public static void main(String[] args){
    int num = 50; // number of threads; can be any number
    Thread[] counterThreads;
    Counter counter = new Counter();

    for(int i = 0; i<num; i++)
      counterThreads[i] = new ConcCounter(counter);

    for(int i = 0; i<num; i++){ counterThreads[i].start(); }
    for(int i = 0; i<num; i++){ counterThreads[i].join(); }
  }
}

public class ConcCounter extends Thread{
  private Counter counter;

  public ConcCounter(Counter c){ counter = c; }
  protected void run(){ counter.incr(); }
}

public class Counter{
  private int count;

  public Counter(){ count = 0; }
  public void incr(){ /* increment count by one */ }
}
```

Listing 2: An implementation for a concurrent counter.

in Listing 3. The processor picks t to run while t' is sleeping. The following execution scenario results in a data race:

- Thread t reads the current value of count, say k.
- Thread t sleeps.
- Thread t is resumed.

```
public class Counter{
  private int count;

  public Counter(){ count=0; }

  public void incr(){
    int tmp = count; /* reading the shared variable */
    count = tmp+1; /* writing to the shared variable */
  }
}
```

Listing 3: Counter class.

- Thread t' reads the current value of count.
- Thread t' updates count to $k + 1$.
- Thread t' is suspended.
- Thread t wakes up.
- Thread t updates count to $k + 1$.

As can be seen here, the contribution of t to the variable count is destroyed. The implementation for the method incr() is correct in a sequential program, but is not thread safe.

1.2 Synchronisation

Thus, to prevent data races, we need to have ways to control the possible interferences between the different threads. Monitors are a classical mechanism to protect and control access to shared variables: only a thread that holds the monitor can be executing the code block, protected by the monitor. In Java, every object has a built-in monitor. A code block B is protected by the monitor associated to object obj by wrapping B with a **synchronized**: *i.e.*, **synchronized**(obj){B } guarantees that at most one thread at the time is executing code block B. If another thread tries to acquire the monitor, it will be blocked until the monitor is released by the first thread.

```
public class Counter{
  private int count;


  public Counter(){ count=0; }

  public void incr(){
   synchronized(this){
     int tmp = count;
     count = tmp+1;
   }
  }
}
```

Listing 4: Intrinsic locking.

```
public class Counter{
  private int count;
  private Lock sync =
    new ReentrantLock();
  public Counter(){ count=0; }

  public void incr(){
    sync.lock();
    int tmp = count;
    count = tmp+1;
    sync.unlock();
  }
}
```

Listing 5: Explicit locking.

Listing 4 shows a variant of the counter, which uses a **synchronized** statement to prevent threads from interfering with each other. Any thread entering the protected region will temporarily own the shared variable count exclusively until it leaves the protected region.

The synchronisation mechanism as discussed above is built-in to Java, which is often called intrinsic locking. In addition, the util.concurrent.lock package also provides an alternative, often called explicit locking, by providing a Lock interface with methods lock() and unlock(). Usage of this interface is illustrated in Listing 5 (using the ReentrantLock implementation of the Lock interface). In Chapter 4: Synchronisers Specifications we will specify the behaviour of the Lock interface in more detail.

If a programmer makes suitable use of locks, the data race problem can be avoided. However, a program might still have race conditions (or high level data races). For example, Listing 6 shows an implementation for the incr() method which is free of data races, but suffers from race conditions.

However, the use of monitors can have a negative impact on the performance of a program, because threads might be blocked, waiting for the monitor to be released. To avoid this performance loss, atomic operations could be used instead. But, in order for this to work, we need to guarantee that an update will immediately be visible to other threads. To support this, Java provides a notion of atomic variable, where all threads are guaranteed

```
public void incr(){
  sync1.lock();
    int tmp = count;
  sync1.unlock();
  sync2.lock();
    count = tmp+1;
  sync2.unlock();
}
```

Listing 6: High-level data-race.

to see the latest value written to the variable. To make a lock-free implementation of the counter, we need to ensure that if a thread reads a value k, before writing $k+1$, k is still the value of the counter. For this purpose, Java provides a compare-and-set (CAS) instruction. A CAS operation takes three parameters as its arguments: 1. a memory location to update, 2. a value expected to be seen in the location, and 3. a new value. The operation inspects the value of the location. If the value is equal to the expected value, it updates it to the new value. Otherwise, it terminates without changing the variable. All these steps are done atomically, meaning that during the operation the location is inaccessible to other threads.

In Listing 7 we implemented our counter example using atomic operations. Later, in Chapter 2 we will discuss atomic variables and the atomic package of Java in more detail. For the moment, let's define the AtomicInteger class (see Listing 7, line 2) as a wrapper for atomic (volatile) variables in Java with three main atomic operations: `get()` for atomic read, `set(int v)` for atomic write and, finally, `compareAndSet(int x, int v)` as the CAS operation. Lines 8 - 10 of Listing 7 shows how a thread repeatedly reads the current value and attempts to update. Essentially, the writing thread calls the `count.compareAndSet(current, next)` method to first check if `count` was updated in the mean time, *i.e.* between lines 8 and 10. If not, the new value is written (in one atomic instruction, together with the comparison). If yes, the thread will repeat the reading of the value of the counter and the update process[6].

[6] In fact, the `AtomicInteger::incrementAndGet()` method is implemented in this way.

```
public class Counter{
  private AtomicInteger count;

  public Counter(){ count=new AtomicInteger(0); }

  public void incr(){
    while(true) {
      int current = count.get();
      int next = current + 1;
      if(count.compareAndSet(current, next))
        return;
    }
  }
}
```

Listing 7: Atomic Increment.

1.3 Verification

In order to make sure a concurrent program always behaves as intended, testing is not sufficient. In particular, because errors might only show up if threads are interleaved in a particular order, and thus, even if the pre-deployment testing phase does not reveal errors, errors might still pop-up later after deployment. Therefore, more advanced techniques are needed, which can analyse all possible behaviours of a program.

Therefore, this thesis advocates the use of static verification to analyse the behaviour of concurrent programs. We use program logics to reason about correct behaviour of concurrent programs. In program logic based verification, first, a formal semantics is defined to describe the possible behaviour of a program. Second, a specification language is proposed (and formalised), which allows one to express the intended properties of a program. Third, a program logic is proposed that enables one (without executing the program) to derive whether the program indeed has the desired properties.

Hoare Logic [43] is the common ancestor of many of these logics: it was one of the first program logics that enabled reasoning about the behaviour of a program purely syntactically. Using ideas from Hoare Logic, the Design-

by-Contract paradigm [62] proposed a systematic development approach to produce more reliable object-oriented software components. The main idea of Design-by-Contract is that for each element of the component its contract should be specified. This contract makes the assumptions and guarantees between the components and its user explicit. In particular, the contract of a given method consists of a pre-condition, which specifies the state in which the method may be called, and a post-condition, which describes the properties on the state that the method guarantees after its execution. To guarantee this, we need to ensure that there exists a relation between the contract and the implementation. This is where the use of (an extension of) Hoare logic comes into play. The proof rules allows one to verify that the implementation indeed fulfills the specification, as expressed by the contract.

For a concurrent program, one of the critical properties that should be verified is whether the program is free of data races. In this thesis, we develop a verification approach using existing program logics to reason about data race freedom of concurrent programs. This means that a verified concurrent program is proven to lack data race in every possible run. Our main focus is on three basic atomic operations, *i.e.* atomic read, atomic write and compare-and-set. We propose specifications of the basic atomic operations to verify the implementation of the synchronisation constructs. Moreover, we also show how other, more involved, properties can be proven, once data race freedom has been established.

1.4 A Practical Approach

Our ultimate target is to develop a practical technique for concurrent programs implemented in Java(-like) programming languages. However, what we see is that even though formal verification is a very promising approach, its practical applicability in industrial applications still is challenging. Typical multi-threaded industrial applications are large and complex. Therefore a verification technique for concurrent software needs to be scalable, easy to use, and automatic in order to be usable in such a setting.

A modular approach to verification is a well-known technique to improve scalability. In modular program analysis [65], components are verified separately, using contracts for the other components. For the verification of sequential programs, procedure-modularity is enough: whenever a method call is encountered, it is verified that the pre-condition of this method is ful-

filled, and the post-condition of the called method is then simply assumed. However, for multi-threaded programs, we also need a thread-modular verification technique, where the behaviour of the other (interferring) threads is abstracted into environment actions. For the verification of a single thread, it is then assumed that all possible interferences of other treads are captured by these environment actions. This approach is used in well-known techniques such as Assumptions-Commitments of Francez and Pnueli [37] and Rely-Guarantee of Jones [51]. During the last decade, Separation Logic [76] and its extensions for concurrent programs [68] opened more opportunities for modular verification of non-trivial concurrent programs implemented in real programming languages. More details about Separation Logic and its role in our work will be discussed later in this thesis.

Certainly, full automation and tool support, to some extent can help an approach to be accepted by the practitioners. Verification of a given concurrent program using a *fully automatic* tool is still an active research topic. In this thesis, we advocate an approach that can reason in a relatively simple way about common synchronisation primitives. To achieve this, first, we need to understand the basic machinery of how the specification and verification should be done, and then once this is there, efforts are made to reduce the number of auxiliary annotations that are needed. This effect is clearly visible in the different chapters of the thesis.

1.5 Thesis

As explained above, synchronisation is a key component in concurrent programming. The programmer either has to define his/her own synchronisation mechanism, using atomic operations, or use one of the available constructs from a standard library, such as the java.util.concurrent package, which provides synchronisation mechanisms for all common thread interaction patterns.

In all cases, ensuring the correctness of the synchronisation mechanism is essential, and this thesis provides the means to do this. The results described in this thesis are part of a larger project, named VerCors [6], on the static verification of concurrent software. Within this VerCors project, we use permission-based Separation Logic [24], an extension of Hoare Logic that is especially suited to reason about concurrent programs (written in different programming languages). The verification techniques developed in

the project are implemented as part of the VerCors tool set [87, 19].

This thesis develops a uniform specification and verification technique for different synchronisation mechanisms. The main goal of the work is to develop a simple specification language that does not require a lot of user interactions for verification. To achieve this goal, instead of proposing a new and powerful program logic, the focus is on reusing well-established logics and to adapt them to our concrete verification problems. In this work we consider atomic operations as the core building block of synchronisation in Java. In addition to the specification and verification of various synchronisation classes, we also show how reasoning about atomic operations can help to reason about non-blocking data-structures where atomic operations are the only shared-variable interactions.

The remainder of this thesis is structured as follows:

- Chapter 1 (Introduction).

- Chapter 2 (Technical Background): presents the necessary technical background for the rest of the dissertation. It covers the basics of the atomic operations in Java, permission-based Separation Logic and it introduces our VerCors specification language that we employ to specify Java concurrent programs.

- Chapter 3 (Reasoning about Thread Creation and Termination): describes the basic approach that we use to reason about multi-threaded Java programs using thread `start` and `join` as the only synchronisation mechanism. Our contribution in this chapter is to use this approach to verify a concurrent pipeline processing program with multiple joins. This chapter is published as part of the following paper:

  A. Amighi, C. Haack, M. Huisman, and C. Hurlin. Permission-based Separation Logic for multithreaded Java programs. *Logical Methods in Computer Science*, 11(1), 2015.

- Chapter 4 (Synchronisers Specifications): proposes contracts for synchronisation constructs. Our contribution in this chapter is to propose a unified approach to specify the behaviour of various synchronisation mechanisms, which are part of the Java concurrency package, *i.e.* `java.util.concurrent`. The approach is illustrated through the verific-

ation of several client programs using synchronisation classes. This chapter is based on the following paper:

A. Amighi, S. Blom, M. Huisman, W. Mostowski, and M. Zaharieva-Stojanovski. Formal Specifications for Java's Synchronisation Classes. In *22nd Euromicro International Conference on Parallel, Distributed, and Network-Based Processing, PDP 2014, Torino, Italy, February 12-14, 2014*, pages 725–733, 2014..

- Chapter 5 (Verification of Synchronisers: Exclusive Access): proposes a technique to reason about the implementation of exclusive access synchronisation constructs. The main contributions of this chapter are: 1. an overview of typical synchronisation patterns using the basic atomic operations as synchronisation primitives; 2. a general specification for these basic atomic operations; 3. a simple, practical and thread-modular contract for AtomicInteger as an *exclusive access* synchroniser; and 4. verification of several examples implementing the synchronisation patterns using our VerCors tool set. This chapter is based on the following paper:

  A. Amighi, S. Blom, and M. Huisman. Resource Protection Using Atomics - Patterns and Verification. In *Programming Languages and Systems - 12th Asian Symposium, APLAS 2014, Singapore, November 17-19, 2014, Proceedings*, pages 255–274, 2014..

- Chapter 6 (Verification of Synchronisers: Shared Reading): is an extension of Chapter 5 to reason about the implementation of *shared-reading* synchronisation constructs. Our contribution in this chapter is to extend the contract of the AtomicInteger class from Chapter 5 in order to verify both exclusive and shared-reading synchronisation constructs. We used the unified contract to verify the implementation of Semaphore, CountDownLatch and SpinLock with our VerCors tool set. This chapter is based on the following paper:

  A. Amighi, M. Huisman, and S. Blom. Verification of Shared-Reading Synchronisers. *SAC*, 2018. Submitted.

- Chapter 7 (Multi-layer Verification based on Concurrent Separation Logic): illustrates how the techniques developed in this thesis can be integrated in a layered approach, to make verification of more involved properties feasible. My contributions in this chapter are: 1. a

layered verification architecture for concurrent programs where each layer verifies one aspect of the program, and 2. verification of the implementation of a lock-free pointer manipulating data structure. This chapter is based on the following paper:

A. Amighi, S. Blom, and M. Huisman. VerCors: A Layered Approach to Practical Verification of Concurrent Software. In *24th Euromicro International Conference on Parallel, Distributed, and Network-Based Processing, PDP 2016, Heraklion, Crete, Greece, February 17-19, 2016*, pages 495–503. IEEE Computer Society, 2016.

- Chapter 8 (Specification and Verification of Atomic Operations in GPGPU Programs): discusses how our technique to reason about atomic operations in concurrent programs can be extended to GPGPU programs. The contributions of this chapter are: 1. a specification and verification technique that adapts the notion of Concurrent Separation Logic resource invariants to the GPU memory model and enables us to reason about data race freedom and functional correctness of GPGPU kernels containing atomic operations; 2. a soundness proof of our approach; and 3. a demonstration of the usability of our approach by developing automated tool support for it. This chapter is based on the following paper:

  A. Amighi, S. Darabi, S. Blom, and M. Huisman. Specification and Verification of Atomic Operations in GPGPU Programs. In *SEFM 2015*, pages 69–83, 2015..

- Chapter 9 (Conclusions): Concludes the dissertation and discuss future directions.

CHAPTER 2

Technical Background

In this chapter we provide the background that is required for the following chapters. This chapter briefly explains how a set of threads can be coordinated using atomic operations in Java. This helps to understand how a synchronisation mechanism behaves, which is an essential step for reasoning. Then, the chapter provides a short overview of the program logic that we employ to reason about correctness of concurrent programs with atomic operations. Finally, a specification language will be introduced that is used to capture the intended behaviour of concurrent programs.

2.1 Atomic Variables in Java

In Java, volatile fields are special fields which are used for communication between threads. Writing to a volatile field has the same memory effect as if a monitor is released and reading from a volatile variable has the same memory effect as locking a monitor. This behaviour of volatile fields guarantees that before reading from a volatile field or after writing to a volatile field, its value is not cached locally. Therefore, when a thread reads a volatile field it never sees a "stale" value. This makes volatile variables suitable for the implementation of a synchronisation mechanism, where it is essential that all threads have a *consistent view* of the state of the synchroniser.

Java provides a concurrency package called java.util.concurrent that supplies library implementation required for concurrent programming [57]. To support thread-safe access to single variables, java.util.concurrent provides the atomic package. The atomic package provides a set of atomic classes as wrappers for volatile variables with different types. Each atomic class exports appropriate atomic operations for read, write, and compare-and-set:

- get(): returns the value that was last written to the field;
- set(T v): atomically assigns the value v of type T to the field; and
- compareAndSet(T x,T n): atomically checks the current value of the field and updates it to n, if it is equal to the expected value x, otherwise leaves the state unchanged, and returns a boolean to indicate whether the update succeeded.

To give an example of an atomic class, Listing 8 shows the AtomicInteger class with its three basic operations, *i.e.* get, set and compareAndSet. This

```
public class AtomicInteger{
  private volatile int value;

  public AtomicInteger(int v){ ... }

  public int get(){ ... }
  public void set(int v){ ... }
  public boolean compareAndSet(int x, int n){ ... }
}
```

Listing 8: The AtomicInteger class.

AtomicInteger class is the basis for almost all the implementations of the synchronisation constructs provided in java.util.concurrent; like ReentrantLock and other classes that are implementing the interface Lock, Semaphore, CyclicBarrier and CountDownLatch.

In java.util.concurrent, the methods of each atomic class are not limited to these three basic operations. For example, AtomicInteger also defines a method for atomic increment: the method incrementAndGet() atomically increments the value by one and returns the new value. However, incrementAndGet() and other similar methods are implemented on top of these three basic operations.

In addition to the synchronisation constructs, atomic operations are also employed directly to implement concurrent data-structures. The implementations of non-blocking and lock-free pointer-based data-structures in Java are extensively using AtomicReference. Similar to AtomicInteger, AtomicReference provides three basic atomic operations to manipulate volatile references in Java. As a well-known example, we refer to a concurrent queue, *i.e.* ConcurrentLinkedQueue implemented in java.util.concurrent, which is the Java implementation of a lock-free queue proposed by Michael and Scott [63].

2.2 Permission-based Separation Logic

Separation Logic [76] is a Hoare-style program logic which was originally introduced to reason about imperative pointer-manipulating programs. In order to reason about memory locations, which are as *resources*, the logic

extends the program state with the *heap*. A key characteristic of this logic is that it allows to reason about disjointness of heaps. In Separation Logic, in addition to the predicates and operators from first order logic, there are two new constructs: *points-to* predicate and $*$-*conjunction* operator.

The *points-to* predicate $e \mapsto v$ describes that:

1. the location of the heap addressed by e is pointing to a location that contains the value v, and

2. the program expression executing $e \mapsto v$ is the *owner* of e.

The $*$-*conjunction* operator in $\phi * \psi$ expresses that predicates ϕ and ψ hold for two disjoint parts of the heap.

Below, we use $[e]$ to denote the contents of the heap at location e and we use $e \mapsto -$ to indicate that the precise contents stored at location $[e]$ is not important. Moreover, the function fv returns the set of free variables of the given commands and assertions.

In addition to classical read and write axioms which ignore the heap, Separation Logic introduces axioms to specify pointer-based operations. Predicates P and Q of a Hoare-triple $\{P\}\ C\ \{Q\}$ in Separation Logic are predicates on the state where the state is a pair of the store and the heap. If the command C is a read/write operation on a store, then classical read/write Hoare-triples are applied. To read and write an address of the heap in Separation Logic, the following rules are used:

$$\frac{}{\{e \mapsto -\}\ [e] := v\ \{e \mapsto v\}} \quad [\textsc{Write}]$$

$$\frac{x \notin \mathsf{fv}(e, e')}{\{e \mapsto e'\}\ x := [e]\ \{e \mapsto e' \wedge x = e'\}} \quad [\textsc{Read}]$$

One of the key advantages of Separation Logic is its power in local reasoning that is achieved by its frame rule:

$$\frac{\{P\}\ C\ \{Q\}}{\{P * R\}\ C\ \{R * Q\}} \quad [\textsc{Frame}]$$

where the command C does not modify the free variables in R. This rule intuitively states that the correctness of $\{P\}\ C\ \{Q\}$ still is valid in a heap extended by R. P is an assertion that specifies only the memory that is affected by the execution of C and after the execution satisfies Q. This

piece of the memory that is locally affected by the command C is called the footprint of the command [38].

The expressiveness and power of Separation Logic to reason about locality and disjoint heaps in imperative programs resulted in Concurrent Separation Logic (CSL) [68], which is an extension of Separation Logic to reason about *multi-threaded* programs. In case of disjoint concurrency, where the threads do not communicate, the following rule for parallel composition can be applied:

$$\frac{\{P\}\ C\ \{Q\} \quad \{P'\}\ C'\ \{Q'\}}{\{P * P'\}\ C || C'\ \{Q * Q'\}} \quad [\textsc{Par}]$$

where C does not mutate any free variable in P', C', Q', and the other way round.

To reason about interacting threads using shared memory, O'Hearn extended CSL rules to show how threads exchange *exclusive* ownership of a memory location through a synchronisation construct [68]. In the rules related to shared memory, the shared state is specified by a *resource invariant*: a predicate that expresses the properties of the shared variables that must be preserved in all the states visible by all the participating threads.

The general judgement in CSL, denoted as $I \vdash \{P\}\ C\ \{Q\}$, expresses that with a resource invariant I, if the execution of C starts with a state satisfying $P * I$, then after the execution (provided that C terminates) the final state will satisfy $I * Q$ and C must not violate I throughout its execution. Having synchronised a group of threads with a synchronisation construct, the resource invariant must only be accessed inside the body of the synchronisation construct by any thread that fulfils the conditions.

For example if the synchronisation construct is a single-entrant lock o then any thread that successfully acquires the lock obtains the full ownership of the resource invariant associated to o, *i.e.* the successful thread obtains exclusive access to the shared data. The successful thread performs its action on the shared variable and must re-establish the resource invariant before releasing the lock. In fact, by acquiring the lock, the thread attaches the shared data to its local state and by releasing the lock it detaches the shared data from its local state. This behaviour of a single-entrant lock is expressed in the following rules [39]:

$$\frac{}{\mathsf{emp} \vdash \{\mathsf{emp}\}\ \mathsf{acquire(o)}\ \{I\}} \quad [\textsc{Acquire}]$$

$$\frac{}{\mathsf{emp} \vdash \{I\}\ \mathsf{release(o)}\ \{\mathsf{emp}\}}\quad [\textsc{Release}]$$

where emp is a predicate that denotes the empty heap, I is the resource invariant. We will discuss about rules and specifications of reentrant locks in Chapter 4.

Reasoning about exclusive ownership of the shared data inside a synchronisation construct paved the way to reason about atomic operations. Vafeiadis presented CSL proof rules for a small language where atomic operations are the only synchronisation constructs [85]. An atomic operation on shared data performs like acquiring or releasing a lock protecting the shared data: 1. the thread executing an atomic operation acquires the lock, 2. it adds the data that is expressed by the resource invariant to its local state, 3. it performs its action on the data, 4. it establishes the resource invariant, and finally, 5. it releases the lock by separating itself from the resource invariant. This is described formally by the following rule:

$$\frac{\mathsf{emp} \vdash \{P * I\}\ C\ \{I * Q\}}{I \vdash \{P\}\ \mathsf{atomic}\{\ C\ \}\ \{Q\}}\quad [\textsc{Atomic}]$$

where emp the empty heap, I is the resource invariant, atomic{ C } indicates that the command C is executed atomically, P is the executing thread's local state before executing the atomic operation and Q is the local state of the executing thread after execution of the atomic operation.

CSL has been extended with permissions [24] to specify and verify *shared-reading* synchronisations [23]. In permission-based Separation Logic, any location of the heap is decorated with a fractional permission $\pi \in (0, 1]$. Any fraction $\pi \in (0, 1)$ is interpreted as a *read* permission and the full permission $\pi = 1$ denotes a *write* permission. Permissions can be transferred between threads at synchronisation points (including thread creation and joining). A thread can mutate a location if it has the write permission for that location. Based on the following rule, permissions on a location can be split and combined to change between read and write permissions:

$$e \overset{\pi}{\mapsto} v * e \overset{\pi'}{\mapsto} v \Leftrightarrow e \overset{\pi+\pi'}{\mapsto} v$$

The addition of two permissions is undefined if the result is greater than the full permission. Soundness of the logic ensures that the total number of permissions on a location never exceeds 1. Thus, at most one thread at a time

can be writing to a location, and whenever a thread has a read permission, all other threads holding a permission on this location simultaneously must have a read permission. As a result, a verified concurrent program using permission-based Separation Logic is data-race free.

The sequential variant of Separation Logic has been extended for sequential Java programs by Parkinson [72]. This logic later has been extended by Haack and Hurlin for multi-threaded Java programs with the support of: 1. thread creation and joining, and 2. reentrant locks [46, 10].

Building on the approach of Haack and Hurlin, we use a fragment of Separation Language to specify synchronisation classes in Java. Here we briefly explain this fragment of Permission-Based Separation Logic which is used to define our VerCors specification language that is used to annotate our programs.

Let E denote arithmetic expressions, B boolean expressions and R pure resource formulas, *i.e.* predicates that specify properties of the heap. In our fragment of CSL, the syntax for assertions P is defined as follows:

$$\begin{array}{lll} B & ::= & \neg B \mid B_1 \wedge B_2 \mid B_1 \vee B_2 \\ R & ::= & \mathsf{emp} \mid E_1 \overset{\pi}{\mapsto} E_2 \mid R_1 * R_2 \\ P & ::= & B \mid R \mid B * R \mid B \implies R \mid \forall x.\ P \mid \exists x.\ P \mid \underset{i \in \mathsf{I}}{\circledast} P_i \end{array}$$

In addition to the classical connectives and first order quantifiers, the main assertions are:

1. the empty heap assertion, written emp,
2. the points-to predicate augmented with permissions, denoted $E_1 \overset{\pi}{\mapsto} E_2$, meaning that expression E_1 points to a location on the heap, has permission π to access this location, and this location contains the value E_2,
3. the separating conjunction operator $*$, expressing that $R_1 * R_2$ holds for a heap if: a) the heap can be split into two disjoint sub-heaps, with the first sub-heap satisfying R_1, and the second sub-heap satisfying R_2, or b) the permission of the heap, say π, can be divided into two permissions π_1 and π_2 such that R_1 expresses the permission π_1 on the heap and R_2 expresses the permission π_2 on the heap.
4. an iterative separating conjunction over a set I, written $\underset{i \in \mathsf{I}}{\circledast} P_i$.

2.3 VerCors Specification Language

The concrete syntax of our VerCors specification language is a combination of permission-based Separation Logic with the Java Modeling Language (JML) [25]. JML specifications are attached to the source code in specially marked comments. Namely, all comments starting with an @ sign, *i.e.* //@ or /*@...@*/, indicate a formal specification that specify the behaviour of the Java program elements in the comment's context. Generally, JML is a very elaborate language, however, we only use a small subset of it:

- the **ghost** keyword is used to introduce ghost fields, that is, class fields only for specifications that extend the object state,
- the method's pre- and post-conditions are given with the **requires** and **ensures** keywords, respectively,
- multiple specification cases are conjoined with the **also** keyword.

In addition to JML conventions, in our examples, intermediate states of the resources are indicated inside /*! ... !*/, which are considered as comments for the verification tool. We present intermediate states of our examples merely for better understanding.

In our specification language we distinguish between *resource expressions* (R, typical elements r_i), *i.e.* expressions that specify properties about the heap, and *functional expressions* (E, typical elements e_i), with the subset of logical expressions of type boolean (B, typical elements b_i).

To annotate our Java programs, our fragment of Separation Logic is expressed by the following grammar:

$$
\begin{array}{rcl}
R & ::= & B \mid \mathsf{Perm(field,\ pi)} \mid (\backslash\textbf{forall}{*}\ \mathsf{T\ i}; B; R) \\
 & \mid & R_1{*}{*}R_2 \mid B{=}{=}{>}R \mid E.\mathsf{P}(E_1, \cdots, E_2) \\
E & ::= & \text{any } \textit{pure} \text{ expression} \\
B & ::= & \text{any } \textit{pure} \text{ expression of type boolean}
\end{array}
$$

where T is an arbitrary type, vi is a variable name, P is an abstract predicate [71] of a special type **resource**, field is a field reference, and pi denotes a fractional permission.

In our specification language, we divert from the classical Separation Logic notation of * for the separating conjunction to ** in order to avoid a syntactical clash with the multiplication operator of Java (and JML).

Intuitively, in a multi-threaded program an assertion Perm(e.f,pi) holds for a thread t if the expression e.f points to a location on the heap and the thread t has at least permission pi to access this location. When the value is important, we sometimes use PointsTo(e.f, pi, v) which is equivalent to Perm(e.f, pi) $**$ e.f == v. In our VerCors specification language fractional permissions are represented as $1/2^n$ where $n \geq 0$. Moreover, assertions can use abstract predicates P to encapsulate the state space [71]. In order to open or close predicates we use **unfold** (open) and **fold** (close) as proof (ghost) commands whenever necessary. Below, we sometimes use an additional requirement that the abstract predicate is a **group**. A **group** is an abstract predicate with multiple permission parameters and axioms satisfying split/merge over permissions [42].

In addition to these classic JML constructs, our method and class specifications can be preceded by a **given** clause, declaring ghost parameters for methods and classes. Ghost method parameters are passed at method calls, ghost class parameters are passed at type declaration and instance creation, resembling the parametric types mechanism of Java. In particular, this is how we pass the resource invariants to the classes. Note, that due to implicit framing of data provided by Separation Logic, there is no need to use the well known JML **assignable** clause to explicitly state method frames. Furthermore, we allow to declare abstract predicates within Java classes, these are simply given by providing the name, typing and parameter declaration in a JML comment inside the class. Building on the JML annotation language allows us to specify permission access properties side by side with complex functional properties. In the scope of the synchronisation classes, however, the permissions are the main focus of this thesis. The exact use of JML becomes apparent when we discuss our specifications in the next chapters.

CHAPTER 3

Reasoning about Thread Creation and Termination

The use of thread's start and join operations is one of the commonly used techniques to implement a divide-and-conquer strategy in a concurrent application. The main thread divides a main task into a set of sub-tasks. Then, for each sub-task the main thread starts a new thread of execution and waits for each thread to join. Finally, the main thread completes its computation by collecting the result(s) of each joining thread.

To reason about a multi-threaded application in an environment with a dynamic thread mechanism, like Java, the specification and verification of thread's creation and termination detection are crucial steps. In Java, threads can join a particular thread via multiple join points. This feature of Java programs makes the reasoning challenging. In his PhD thesis, Hurlin [46] proposed contracts for the class Thread to handle this. The contracts are specified based on permission-based separation logic developed for multi-threaded Java programs [10].

In this chapter, we first summarize the technique presented in [10] for reasoning about thread's start and join in Java and then illustrate it on a larger case study. First, in Section 3.1 we shortly explain how thread's start and join work in Java. Then, in Section 3.2 we explain the specification of the class Thread where a join token records the portion of the resources that each thread can obtain in its join points. The new contribution of this chapter (presented in Section 3.3) is that we will illustrate the technique on an example with multiple join points. The example shows a general data-processing pattern in a multi-threaded program, typically implemented in efficient signal processing applications. The verification of this pattern in Java has first been published in [10]. To present the example, we also discuss the specification and reasoning technique, which is fully formalized in [10], but that partly is based on the result of Hurlin's PhD thesis [46].

3.1 Reasoning about Dynamic Threads

In Java, the start() and join() methods provided by the class Thread are implemented natively. Calling the start() method from an instantiated object obj : C (obj has type class C which inherits from Thread) causes the virtual machine to create a new thread of execution associated with obj. The created thread, after its initialization, invokes the obj.run() method. The developer has to override the run() method in class C, to define the thread's

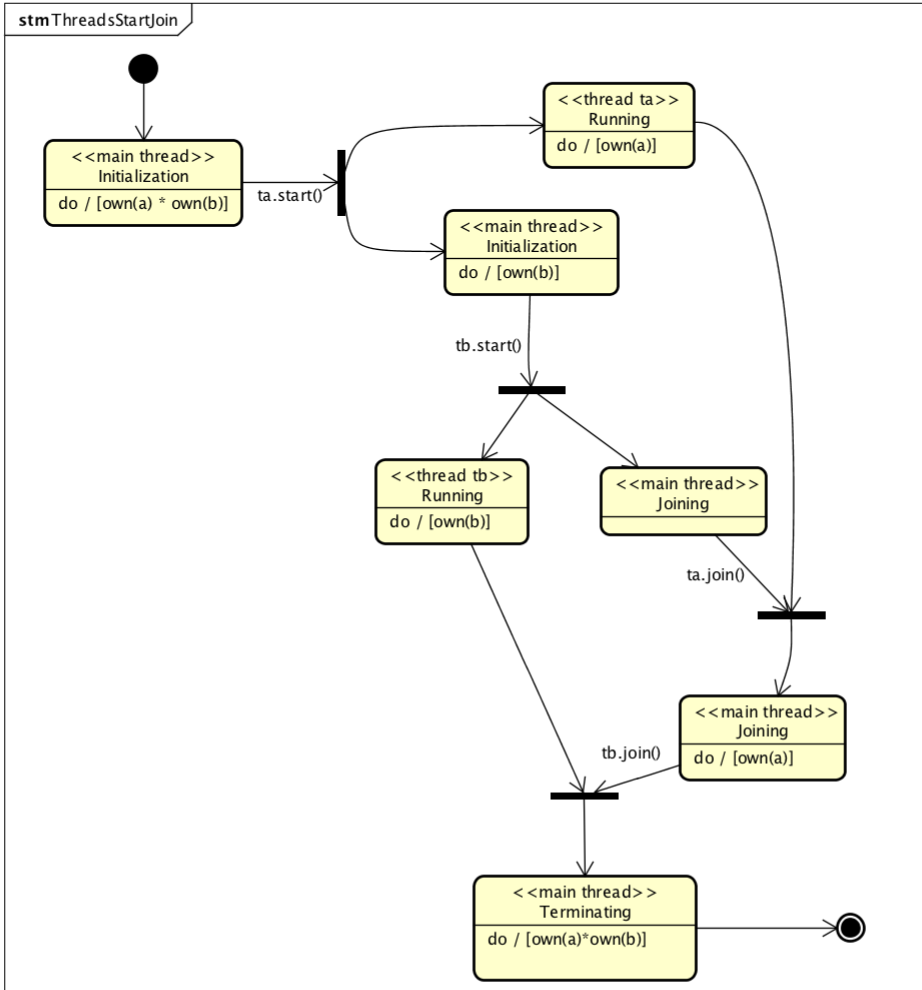

Figure 3.1: Ownership transfer by start and join

task.

Threads can wait for each other until execution of the run() method is terminated. Any thread t calling obj.join() will wait for thread t' running obj.run() to terminate. After calling obj.join(), t is blocked until the execution of obj.run() is finished. While obj.join() can be called several times by different threads, the obj.start() method may not be called more than once.

In reasoning about concurrent programs based on permissions, when a thread starts its execution, it must obtain all the permissions it may require for its processing. Similarly, when a thread t joins another thread t', permissions of the resources held by t' should be transferred to the joining thread t. Figure 3.1 presents a state machine diagram for ownership transfer for a simple pattern of joining: a main program starts two threads and then joins them later. The main program in this simple execution, at first, holds the full ownership of a and b asserted as $\mathsf{own(a)} * \mathsf{own(b)}$. After starting two threads ta and tb, the ownership of a and b are transferred to disjoint threads ta and tb. Each thread has to give back these resources to the main thread after termination as the main program is responsible to collect the results of the concurrent processes.

Reasoning about thread's start and join would be less challenging if the starter thread was the only thread that could join, because in this case the starter thread can keep track of the resources that it should expect at its join point. In Java, however the creating thread is not necessarily the only one that can join the thread t'. In fact, a group of threads can obtain obj, *i.e.* the reference to the joined thread t', from different sources, and each thread separately and independently can call obj.join() to join t'.

This multiple join mechanism in Java provides a flexible and dynamic framework for multi-threaded programming. However, this flexibility makes it difficult to verify programs. This difficulty is mainly because, in contrast to the single start-join context, in reasoning about multi-join points, the joining thread does not have any information about the resources that it should expect from the joined thread. Thus, there is a need for a technique that specifies the resources that each thread can obtain at its join points.

Hurlin in his thesis [46, 10] proposed a fully formalized contract for the class Thread. This contract, which is presented in Listing 9, is employed to reason about the correctness of the joining threads. When verifying a thread that creates or joins another thread, the calls to start and join are verified using the standard verification rule for method calls. Complete collection of verification rules is formally presented in [46, 10]. In the remainder of this chapter, first, we explain the contract of the methods and then, based on a general concurrent processing pattern we illustrate how one can verify multi-joining concurrent programs in Java.

```
class Thread implements Runnable {

  /*@
  resource start();
  resource preFork() = true;
  group resource postJoin(frac p) = true;
  group resource join(frac p); @*/

  /*@
  requires true;
  ensures start(); @*/
  public Thread();

  /*@
  requires preFork();
  ensures postJoin(1); @*/
  void run();

  /*@
  requires start()**preFork();
  ensures join(1); @*/
  public void start();

  /*@
  given frac p;
  requires join(p);
  ensures postJoin(p); @*/
  public void join();
}
```

Listing 9: Specification of class Thread.

3.2 Contract of Class Thread

In this section we use the VerCors specification language that (presented in Chapter 2) to specify the contract of class Thread. The contract for class Thread is presented in Listing 9 and will be explained in detail.

Constructor To instantiate an object from the class Thread the creator does not need to provide any resource. But, after instantiating the object, the constructor ensures a token, named start. This token is returned by the constructor to ensure that only one thread can call the start() method, which consumes the start token.

Method run() To verify that the thread functions correctly, the body of the run method is verified w.r.t. its specification. The contract of the run method specifies what permissions are transferred when threads are created and joined: the pre-condition of a thread is the pre-condition of the run method; the post-condition of a thread is the post-condition of the run method. For this purpose, we specify predicates preFork and postJoin that denote this pre- and post-condition, respectively. These predicates have trivial definitions to be extended in the classes extending and implementing Thread. Every class that extends the class Thread, *i.e.* the child class, extends the predicates preFork and postJoin to denote extra permissions that are passed to the newly created thread.

Method start() Any object starting a thread has to provide the start() token along with the resources specified by the preFork predicate. In return the start() method ensures a join token. This join token has a fraction as argument, which holds a fractional permission p. This permission specifies which part of the postJoin predicate can be obtained by the thread invoking the join method. It should be stressed here that both predicates postJoin and join have to be splittable w.r.t. this permission. Thus, both are defined as a **group** (see Chapter 2).

Method join() A thread that calls the join() method has to give up a fraction p a fraction of its join token, and in return obtains a p fraction of the resources specified by the postJoin predicate. The actual fraction of the join token that the joining thread currently holds is passed as an extra parameter to the join method, via the **given** clause.

3.3 Example: Multi-threaded Data Processing

To illustrate his approach in reasoning about thread start and termination in Java, Hurlin verified a merge-sort algorithm [46]. To demonstrate the

```
public class Buffer {
/*@ resource state(frac p)=
   Perm(inp,p)**Perm(outa,p)**Perm(outb,p); @*/
  public int inp;
  public Point outa, outb;
}
```

Listing 10: Class Buffer.

applicability of the approach in a concurrent program in presence of multiple joins, here we verify a well-known pattern used for pipe-line processing algorithms. In this pattern there can be several data flows through a sequence of parallel tasks. This is a common pattern of signal-processing applications, in which a chain of threads are connected through a shared buffer. Each thread represents a sequence of processing units usually called processing filters. Each filter obtains its input data, performs a sequence of computational operations and produces its output for the next processing filter.

Our example is a simplified version of this pattern, with one main program initializing one data sampler, two processing filters and one monitoring unit. The shared buffer is an instance of a Buffer class that encapsulates an input field and two points, see Listing 10. The sampler thread is an instance of the Sampler class, our processing filters are instances of the AFilter and BFilter classes, and the monitoring is a thread instantiated from the Plotter class. Listing 11 shows the sampler thread, Listing 12 shows the AFilter class (class BFilter is similar and not shown here), Listing 13 shows the Plotter class and finally Listing 14 shows the main application.

First, the sampler thread assigns a value to the input field of the buffer. Next, it passes the buffer to processes A and B, which are executed in parallel. Based on the value that the sampler thread stored in the inp field of Buffer, each process calculates a point and stores its value in the shared buffer. Finally, the computation results from both processes are displayed by the plotter.

What makes this example interesting is that both processes A and B join the sampler thread, *i.e.*, they wait for the sampler thread to terminate, and in this way they retrieve read permissions on the input data that was written by the sampler thread. Moreover, the plotter waits for the two processing

```
public class  Sampler extends Thread {
2   /*@
    resource preFork = Perm(buffer.inp,1);
4   group resource postJoin(frac p) = Perm(buffer.inp,p);  @*/
    Buffer buffer;
6
    // constructor
8
    /*@
10  requires preFork();
    ensures postJoin(1); @*/
12  public void run(){
      /*@ unfold preFork; @*/
14 /*! { Perm(buffer.inp,1) } !*/
      sample();
16 /*! { Perm(buffer.inp,1) } !*/
      /*@ fold postJoin(1) ; @*/
18  }

20  /*@
    requires Perm(buffer.inp,1);
22  ensures Perm(buffer.inp,1); @*/
    private void sample(){ /* fill buffer.inp */ }
24 }
```

Listing 11: Class Sampler.

threads to terminate (by joining them), to collect read permissions on the shared buffer. Then, the plotter combines the collected read permissions into full write permission and passes it to the main thread.

In our example (see Listing 14), we sketch an outline of the correctness proof. The main thread starts its execution with the full permission of all the fields from the shared buffer. Then, after creating each thread, the main method obtains a start() token for the created thread. To start any thread, the main method has to provide the start() token and folds the resources defined in the corresponding preFork(). For example, as shown in line 8 of Listing 14, in order to provide the required resources of the sampler

thread, the main thread folds the full permission of the inp field from the shared buffer. In return, the sampler thread ensures its join(1) token to the main thread. Later (see lines 14 and 18), the main method splits the join token of the sampler and transfers each half to the processing threads, as specified by their preFork predicates. Thus, the processing threads (Listing 12) use a half join token to join the sampler thread, and to obtain half the resources released by the sampler thread. Additionally, the join tokens for the processing threads are transferred to the plotter by which the plotter can join them (see line 24).

The sampler thread, upon its start is provided with the full permission on inp. When starting each processing thread, full permission on the corresponding output field, *i.e.* outa and outb, along with a partial join token for the sampler thread are transferred. Using the join token, each processing thread joins the sampler thread. However, as each thread only has a fraction of the join token they only obtain a fractional permission on inp field. At the join point, the argument of the join token enforces the processing thread to claim only a half permission on the inp field. Then, after reading the provided input field the process is performed and the result is written to the given output filed. Similarly, the plotter thread obtains a 1/2 read permission on inp and a write permission on outa by joining process A, and another 1/2 read permission on inp and a write permission on outb by joining process B. It then combines the read permissions into a write permission on inp to invoke its plot method. Finally, by joining the plotter thread, the main thread captures all the given permissions.

3.4 Conclusion and Related Work

In this chapter we applied the technique proposed by Haack and Hurlin to verify data race freedom of a concurrent Java program with multiple join points. This program is an example of a typical signal processing application. We used a permission-based Separation Logic specification for the class Thread of the java.lang package, which enables verification of concurrent programs with thread's start and join as concurrency primitives.

The contract of class Thread defines a pre-condition and a post-condition for thread's start, run, and join methods in terms of abstract predicates preFork and postJoin. Any thread (*i.e.* a class inheriting from Thread) should instantiate the definition of these predicates with the resources that the

thread needs when it is started, respectively releases when it is terminated. As demonstrated in the examples, our approach supports multiple joins. Employing a join token in the contract, we keep track of the fractions that the thread should deliver at its join points. The postJoin token should be splittable over the resources, which ensures that it can be used to reason soundly about programs that joins a thread multiple times. We use a special join token to keep track of the fraction of the postJoin predicate a thread should obtain upon joining.

The downside of our approach is that the verification may require a specific order for thread creation and start. For example, in our example, in order to create the data processing threads, we have to start the sampler thread first. Only in this way, the main method can obtain the join token of the sampler thread. This is a restriction for the programmer as the start of the threads could be done as soon as all the threads are instantiated.

A general solution to specify and reason about tokens can be an interesting direction for future work to solve this restriction. In this direction, Penninckx employed Petri nets to model and reason about concurrent I/O based programs [73]. His technique for the specification and reasoning about tokens may be applicable here.

```
public class AFilter extends Thread {
  private Sampler sampler;
  private Buffer buffer;

  /*@
  resource preFork() = Perm(buffer.outa,1)**sampler.join(1/2);
  group resource postJoin(frac p) =
        Perm(buffer.outa,p)**Perm(buffer.inp,p/2); @*/

  // constructor

  /*@
  requires preFork();
  ensures postJoin(1); @*/
  public void run(){
    /*@ unfold preFork; @*/
/*! { Perm(buffer.outa,1)**Join(sampler, 1/2) } !*/
    sampler.join();
/*! { Perm(buffer.outa,1)**sampler.postJoin(1/2) } !*/
    /*@ unfold sampler.postJoin(1/2); @*/
/*! { Perm(buffer.outa,1)**Perm(buffer.inp, 1/2) } !*/
    processA();
/*! { Perm(buffer.outa,1)**Perm(buffer.inp, 1/2) } !*/
    /*@ fold this.postJoin(1); @*/
  }

  /*@
  requires Perm(buffer.outa,1)**Perm(buffer.inp, 1/2);
  ensures Perm(buffer.outa,1)**Perm(buffer.inp, 1/2);  @*/
  private void processA(){ /* read buffer.inp and fill buffer.outa. */ }
}
```

Listing 12: Class AFilter.

```
public class Plotter extends Thread {
  private Buffer buffer; private AFilter ta; private BFilter tb;
  /*@
  resource preFork() = ta.join(1)**tb.join(1);
  group resource postJoin(frac p) = buffer.state(p); @*/

  /*@
  requires true;
  ensures start()**Perm(buffer,1)**Perm(ta,1)**Perm(tb,1); @*/
  public Plotter(Buffer buf, AFilter fa, BFilter fb){
    buffer=buf; ta=fa; tb=fb; }

  /*@
  requires preFork(); ensures postJoin(1); @*/
  public void run(){
    /*@ unfold preFork @*/
/*! { ta.join(1)**tb.join(1) } !*/
    ta.join();
/*! { ta.postJoin(1)**tb.join(1) } !*/
    tb.join();
/*! { ta.postJoin(1)**tb.postJoin(1) } !*/
    /*@
    unfold ta.postJoin(1); unfold tb.postJoin(1);
    fold buffer.state(1); @*/
/*! { buffer.state(1) } !*/
    plot();
/*! { buffer.state(1) } !*/
    /*@ fold this.postJoin(1); @*/
  }

  /*@
  requires buffer.state(1); ensures buffer.state(1); @*/
  private void plot(){ /* plots the processed data from the buffer */ }
}
```

Listing 13: Class Plotter.

```
/*@ requires buf.state(1); ensures buf.state(1); @*/
void main(){
/*! {buf.state(1) } !*/
  Sampler s=new Sampler(buf);
/*! {s.start()**buf.state(1) } !*/
  /*@ unfold buf.state(1); @*/
/*!{s.start()**Perm(buf.inp,1)**Perm(buf.outa,1)**Perm(buf.outb,1)}!*/
  /*@ fold s.preFork();  @*/
/*! {s.start()**s.preFork()**Perm(buf.outa,1)**Perm(buf.outb,1) } !*/
  s.start();
/*! {s.join(1)**Perm(buf.outa,1)**Perm(buf.outb,1) } !*/
  AFilter a = new AFilter(buf, s);
/*! {a.start()**s.join(1)**Perm(buf.outa,1)**Perm(buf.outb,1) } !*/
  /*@ fold a.preFork;  @*/
/*! {a.start()**a.preFork()**s.join(1/2)**Perm(buf.outb,1) } !*/
  BFilter b = new BFilter(buf, s);
/*!{b.start()**a.start()**a.preFork()**s.join(1/2)**Perm(buf.outb,1)}!*/
  /*@  fold b.preFork;  @*/
/*! {b.preFork()**b.start()**a.start()**a.preFork() } !*/
  a.start(); b.start();
/*! {a.join(1)**b.join(1) } !*/
  Plotter p = new Plotter(buf, a, b);
/*! {p.start()**a.join(1)**b.join(1) } !*/
  /*@ fold p.preFork;  @*/
/*! {p.start()**p.preFork() } !*/
  p.start();
/*! {p.join(1) } !*/
  p.join();
/*! {p.postJoin(1) } !*/
  /*@ unfold p.postJoin(1); @*/
/*! {buf.state(1) } !*/
}
```

Listing 14: Verification of the main program.

CHAPTER 4

Synchronisers Specifications

In Chapter 3, the contract of class Thread is explained. This can be used to reason about thread's start and join, which are considered as the most basic concurrency primitives. In addition to thread's start and join there are other synchronisation mechanisms provided in java.util.concurrent, which can be used to develop shared memory concurrent programs. The concurrent package contains implementations of several synchronisation classes, in particular variants of lock, a semaphore, a count-down latch, and a cyclic barrier.

In this chapter we discuss the specification of the main synchronisation classes in java.util.concurrent. To find out which classes are used most often, and thus, where our specifications efforts are most useful, we consider the number of references to classes in the concurrency packages for each of the projects in the standard Qualitas Corpus benchmark suite [83] using the Histogram tool [20]. In essence, the tool efficiently analyses a large collection of Java bytecode classes to build statistics on class and method usage. The result of the analysis shows [20] that the reentrant family of locks is used most often. Moreover, it reveals that CountDownLatch is used far more often in practice than it is explained in textbooks, while the CyclicBarrier, which is very similar to the latch, did not score very high in the overall statistics.

The contracts presented here for the different synchronisation classes lift all the elements of the formalization of reentrant Java locks [46, 10] to the specification layer. Therefore, the underlying concepts can be shared, reused, and adapted by the different synchronisation classes of the Java's util.concurrent package. In particular, the notion of a resource invariant is at the base of all the specifications: all specifications of the synchronisation methods express how permissions are transferred between the thread and the resource invariant.

Below, in Section 4.1, we first start with the specification of the lock family provided in the Java concurrent package. We present how the logic for built-in reentrant locks from [10, 41] is generalized to specify the Lock interface, and how the specifications of ReentrantLock and ReadWriteLock both are built on top of this general specification for Locks. Then, we discuss how we treat lock initialization at specification level. Next, in Section 4.2, Section 4.3 and Section 4.4 we discuss specifications of three other frequently used synchronisation classes: Semaphore, CountDownLatch, and CyclicBarrier. Finally, in Section 4.5 we end the chapter with a collection of examples to show how a client program can be verified using our specific-

ations. Our examples cover verification of the client programs synchronised with locks, latch and barrier. We will not show any example for semaphore as it is very similar to CountDownLatch. Our approach along with examples illustrate that synchronisation mechanisms can be specified using a unified approach. This chapter is based on [4, 1]. However, the specification of the CyclicBarrier has not been published yet.

4.1 Locks in Java

In Java, every object can function as a lock, using the **synchronized** keyword. Synchronisation using the **synchronized** keyword makes it impossible to forget to release a lock. However, its syntactic limitations make it impossible to acquire and release locks at arbitrary points in the code.

In the Java concurrency package java.util.concurrent, Lea introduced a set of synchronisation classes to address various synchronisation mechanisms [57]. Among other things, this package features: (1) locks and other synchronisation primitives, (2) the Executor framework, providing task-based parallelism, (3) thread-safe data structures, such as maps and queues, and (4) support for atomic variables.

The synchronisation classes in the Lock hierarchy in the concurrency package (see Figure 4.1) are devoted to resource locking scenarios where either full (write) access is given to one particular thread or partial (read) access is given to an indefinite number of threads. For situations where, depending on the execution context, either shared or exclusive access is required, the API defines a ReadWriteLock interface. All interfaces are implemented by classes that support lock re-entrancy. We first discuss the specification of the Lock interface, and then we proceed with specifications of different lock implementations.

Lock Interface Specification

As explained above, our specification approach of the different synchronisation mechanisms is inspired by the logic developed by Haack *et al.* [46, 10]. First, we briefly describe the logic to reason about *built-in* Java reentrant locks. Then, we explain how we lift this logic to specification-level and generalise it to other synchronisation mechanisms in the Java API.

In the logic developed for reasoning about reentrant locks, for each lock, an abstract predicate inv describing the resource invariant is specified, de-

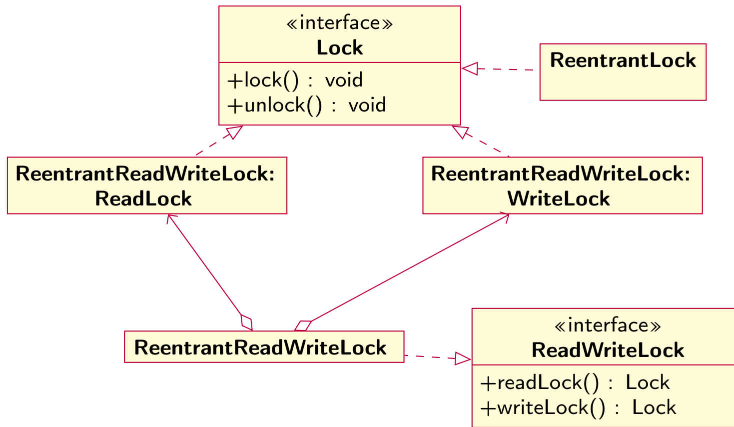

Figure 4.1: The hierarchy of locks in the java.util.concurrent package.

scribing which locations are protected by the lock. Whenever a lock is acquired for the first time, the locking thread obtains these protected locations, and thus can access the protected data. Upon final release of the lock, the thread is forced to give up the protected data. To reason about reentrant locks, the reasoning logic must be aware of the level of entrance. It means that the acquiring thread must not obtain the resource invariant if it acquires a lock that it already holds. To distinguish initial acquirings and final releases from reentrant acquirings and releases, each thread maintains a multi-set LockSet that keeps track of all locks (including their multiplicity) that the thread currently holds. The lock sets are necessary to properly treat *lock reentrancy*: if a thread acquires a lock that is already in the lock set, it does not obtain any permissions, and if a thread releases a lock, it does not have to give up any permissions if the lock afterwards is still in the lock set.

To give a flavour of this logic, Figure 4.2 quotes the proof rules from [41], preserving the original syntax, for initial and reentrant acquiring of a lock. The first rule states that if a thread locks u, and the set of currently held locks does not contain u yet, and the lock is initialised, then upon completion of the u.lock() statement, u is in the lock set, and the resource invariant has been transferred to the thread holding the lock on u. However, as indicated by the second rule, if u is already held by the current thread, no permissions are transferred, only bookkeeping of the additional acquiring of the lock is

$$\dfrac{\Gamma \vdash u, S : \mathsf{Object}, \mathsf{lockset}}{\Gamma; v \vdash \begin{array}{c}\{\mathsf{LockSet}(S) * u \notin S * u.\mathsf{init}\} \\ u.\mathsf{lock}() \\ \{\mathsf{LockSet}(u \cdot S) * u.\mathsf{inv}\}\end{array}} \text{[Lock]}$$

$$\dfrac{\Gamma \vdash u, S : \mathsf{Object}, \mathsf{lockset}}{\Gamma; v \vdash \begin{array}{c}\{\mathsf{LockSet}(u \cdot S)\} \\ u.\mathsf{lock}() \\ \{\mathsf{LockSet}(u \cdot u \cdot S)\}\end{array}} \text{[Re-Lock]}$$

Figure 4.2: Proof rules for initial and reentrant acquiring of a lock.

done.

Now we explain how to translate the rules from this logic into method specifications of the Lock interface, because the Lock interface can be used in different and wider settings. In particular, Lock implementations may be non-reentrant; they may be used to synchronise non-exclusive access; and they may be used in *coupled* pairs to change between shared and exclusive mode (see the read-write lock specification below).

Therefore, compared to the logic presented in [46, 10], the following changes for the specification given in Listing 15 are necessary:

- The locks are parametrized by two boolean variables isExclusive and isReentrant, which can be correspondingly instantiated by implementations (line 2).

- To allow non-exclusive synchronisation, resource invariants have to be groups (line 3).

- For the non-exclusive locking scenarios, the client program has to record the amount of the resource fraction that was obtained during locking, so that the lock can reclaim the complete resource fraction upon unlocking. This information is passed around in the held predicate, which holds this fraction (line 7). This is purposely not declared as a group, so that clients are obliged to return their whole share of

```
 /*@
2  given boolean isExclusive, isReentrant;
   given group (frac -> resource) inv;  @*/
4 public interface Lock {
   /*@
6  group resource initialized(frac p);
   resource held(frac p);
8  ghost public final Object parent;  @*/

10  /*@ given bag<Object> S, frac p;
      requires initialized(p);
12    requires LockSet(S) ** !(S contains this);
      requires parent != null ==> !(S contains parent);
14    ensures LockSet(this::parent::S) **
        inv(isExclusive ? 1 : p) ** held(p);
16   also
      requires isReentrant ** LockSet(S) **
18      (S contains this) ** held(p);
      ensures LockSet(this::S) ** held(p); @*/
20  void lock();

22  /*@ given bag<Object> S, frac p;
      requires LockSet(this::S) ** (S contains this) ** held(p);
24    ensures LockSet(S) ** held(p);
     also
26    requires held(p) ** inv(isExclusive ? 1 : p);
      requires LockSet(this::parent::S) ** !(S contains this);
28    ensures LockSet(S) ** initialized(p); @*/
    void unlock();
30 }
```

Listing 15: Specification of the Lock interface.

resources. The held predicate is returned during locking in exchange for the initialized predicate which is temporarily revoked for the time that the lock is acquired.

- For situations where several locks share the same resource and are

effectively coupled as one lock, we need to ensure that only one lock is locked at a time. The coupling itself is realized by holding a reference to the parent object that maintains the coupled locks (line 8). The exclusive use of coupled locks is ensured by storing and checking this parent object in the set of currently held locks.

- A separate specification case is given for reentrant locking (when the parameter isReentrant is true).

As a result, in the specification of method lock() in Listing 15, given the multi-set of locks, *i.e.* bag<Object> S, when the lock is acquired for the first time (lines 10–15), the locking thread gets permissions from the lock. If the lock is reentrant, and the thread already holds the lock (lines 17–19), then no new permission is gained, only the multi-set of locks held by the current thread is extended with this lock (where :: denotes bag addition). For coupled locks (where the parent is not null) the presence of the parent in the lock set is also checked and recorded, to prevent parallel use of the coupled locks. The specification of method unlock() in Listing 15 describes the reverse process: if the multi-set of locks contains the specific lock only once (lines 26–28), then this means the return of permissions to the lock (*i.e.*, inv does not hold in the post-condition) according to the held predicate; otherwise (lines 22–24), the thread keeps the permissions, but one occurrence of the lock is removed from the multi-set.

ReentrantLock Specification

Class ReentrantLock implements the Lock interface as an exclusive, reentrant lock. Thus, it inherits all specifications from Lock and appropriately instantiates the two class parameters isReentrant and isExclusive both to **true**:

```
/*@ given group (frac -> resource) inv; @*/
class ReentrantLock implements Lock /*@<inv,true,true>@*/ {
 ...
}
```

ReadWriteLock Specification

The ReadWriteLock is not a lock itself, but a wrapper of two coupled Lock objects: one of them provides exclusive access for writing (WriteLock), while the other allows concurrent reading by several threads (ReadLock). The

```
/*@
 given group (frac -> resource) inv;
 given boolean reentrant; @*/
interface ReadWriteLock {
 /*@ group resource initialized(frac p); @*/

 /*@ given frac p;
   requires initialized(p);
   ensures \result.parent==this ** \result.initialized(p); @*/
 /*@pure@*/ Lock /*@<inv,false,reentrant>@*/ readLock();

 /*@ given frac p;
   requires initialized(p);
   ensures \result.parent==this ** \result.initialized(p); @*/
 /*@pure@*/ Lock /*@<inv,true,reentrant>@*/ writeLock();
}
```

Listing 16: Specification of the ReadWriteLock interface.

two classes are commonly implemented as inner classes of the class that implements the ReadWriteLock interface (see Figure 4.1 on page 45). The two locks are intended to protect the same memory resources. Hence our specifications in Listing 16 state that the two getter methods (expressed with **pure**) for obtaining the two locks return a lock object with the same resource inv, but which are non-exclusive (line 10) and exclusive (line 15), respectively. The aggregate read-write lock has to be initialized itself (lines 8 and 13). Further, using the return value keyword **\result**, we state in the respective post-conditions of the getter methods (lines 9 and 14) that the obtained locks are initialized and hence can be acquired, and that they have the same parent object, which is an instance of the class implementing the ReadWriteLock interface. In Section 4.5 we illustrate how this specification can be employed to verify a concurrent producer-consumer.

Initialization of Resource Invariants

In the specification of the synchronisation classes, resource invariants have to be initialized, *i.e.*, the permissions have to be transferred "into" the synchroniser, before the synchronisation mechanism can be used. This ensures

```
/*@
  ghost boolean initialized = false;
  group resource initialized(frac p) = PointsTo(initialized, p/2, true);
 requires inv(1) ** PointsTo(initialized, 1, false);
 ensures initialized(1); @*/
public void commit();
```

Listing 17: Specifications for lock initialization.

that the resources can be passed to a user program upon synchronisation without introducing new resources.

Initialization of the resource invariant is done in the same way for all synchronisation mechanisms: class Object declares a ghost boolean field initialized that tracks information about the initialization state of the resource invariant. Newly created locks are not initialized; the specification-only method commit (see Listing 17) can be used by the client code to irreversibly initialize the lock. This means that the resources protected by the lock, as specified in inv, become shared. To achieve this, commit requires the client to provide the complete resource invariant inv(1), together with an exclusive permission to change initialized (line 4). The method consumes the invariant ("stores it into the lock"). Moreover, it ensures that initialized cannot be changed anymore by consuming part of the permission to access this field, effectively making it read-only (lines 3 and 5). For convenience, the result of commit is encapsulated in a single resource predicate initialized, which can be passed around and used as a permission ticket for locking operations, see below. The default location for the call to commit is at the end of the constructor of the synchronisation object. More complex lock implementations (that we do not discuss in this here) may require moving this call to another location in the program.

The actual resource invariant is typically decided by the user of the synchronisation class, therefore it is passed as a class parameter with the type (**frac** −> **resource**). For example, in order to protect a shared location x using a reentrant lock, the resource invariant that protects the location x is specified with xInv, which is passed both during type declaration and during instantiation of the lock:

```
/*@ resource xInv(frac p) = Perm(x, p); @*/
Lock/*@<xInv, ...>@*/ xLock = new ReentrantLock/*@<xInv>@*/();
```

```
/*@ given group (frac -> resource) inv;  @*/
public class Semaphore {
 /*@
  resource held(frac p) = initialized(p);
  ghost final int permits;  @*/

 /*@
  requires inv(1) ** permits > 0;
  ensures  initialized(1) ** this.permits == permits; @*/
  public Semaphore(int permits);

 /*@
  given frac p;
  requires initialized(p);
  ensures inv(1/permits) ** held(p); @*/
  public void acquire();

 /*@
  given frac p;
  requires inv(1/permits) ** held(p);
  ensures initialized(p); @*/
  public void release();
}
```

Listing 18: Specification of the Semaphore class.

In our specifications such parameters (of which there will be more, hence the "..." above) are received through parameters specified with the **given** keyword.

4.2 Semaphore Specification

The Semaphore class represents a *counting semaphore*. It is used to control threads' accesses to a shared resource, by restricting the number of threads that can access a resource simultaneously. Each semaphore is provided with a property *permits*, that represents the maximum number of threads that can access the protected resource. Accessing the resource must be preceded

by acquiring a permit from the semaphore. A semaphore with n permits allows a maximum of n threads to access the same resource simultaneously. If n threads are holding a permit, a new thread that tries to acquire a permit blocks until it is notified that a permit is released.

When initialized with more than 1 permit, a semaphore closely corresponds to a non-reentrant ReadLock, but with the number of threads accessing the shared resource explicitly stated and controlled. When initialized with 1 permit, it provides exclusive access, and behaves the same as a non-reentrant WriteLock. Therefore, the specification of the semaphore is a stripped-down version of the Lock specification, see Listing 18. In particular, semaphores are never reentrant, and they are not used in coupled combinations. Moreover, since the maximum number of threads that can access the shared resource is predefined with the permits field, we can also limit ourselves to simply providing each acquiring thread with an equal split of 1/permits of the resource invariant (lines 15 and 20). Note also that there is no access permission required for the permits field as it is declared to be final and hence can never change after initialization.

4.3 CountDownLatch Specification

Essentially, a count-down latch is a *scattered* multi-threaded lock. Typically, a parent thread initializes a latch with a count and then passes it to a number of worker threads together with some shared resource for the threads to work on. Each worker thread, once finished, calls method countDown() on the latch to signal that it releases its share on the resource. Threads can wait for all worker threads to finish by calling the blocking await() method. Each call to countDown() decreases the internal latch counter, and once this reaches zero, all awaiting threads unblock and can use the protected resource again.

In the context of our work, the await() is technically a *locking* operation – permissions are transferred to the calling thread – while countDown() is an *unlocking* operation – the calling thread gives up permissions for resources it worked on. Thus, intuitively, compared to the lock, the flow of resource permissions is reversed. Moreover, latches are scattered: each worker thread performs only a partial unlock to collectively achieve a full one, and all awaiting threads do a collective lock. Because of this scattered synchronisation, the specification method that employs *one* resource invariant inv does not work as good as it worked for the reentrant locks and semaphores. Yet, it is

still possible to provide a specification for simplified latch use scenarios where only one inv resource predicate is used. In this simplified scenario the distribution of the resources between the worker threads and the awaiting threads is considerably limited, we discuss this first. We then discuss an improved specification that accounts for arbitrary distribution of resources between n worker threads that call countDown and a different number m of awaiting threads. Later, in Section 4.5 we use our improved specification to verify a client program that is synchronised with an instance of CountDownLatch.

Simplified Scenario

Our simplified scenario assumes that each worker thread gets an equal non-exclusive share of some resource, and awaiting threads receive a fractional split of the whole resource. In particular, if there is only one awaiting thread, it automatically reclaims the whole resource invariant protected by the latch. The associated specifications are shown in Listing 19. When the latch is constructed (lines 6–9) two predicates are returned: initialized(1), reflecting the ability of the receiving thread to call await, plus count number of equal held predicates. The thread that created the latch should pass these to each of the worker threads, along with a corresponding split of the resource invariant, *i.e.*, it is not the responsibility of the latch to distribute the resource invariant at this point. When calling countDown (lines 12–13), the worker thread presents its held predicate and an associated fraction of the resource invariant, which is then consumed back into the latch. The await method (lines 16–19) expects at least a fraction of the initialized predicate and returns an associated split of the resource invariant. In case there is only one awaiting thread and the initialized predicate is unsplit, the complete resource invariant inv(1) is returned.

Generalized Scenario

A generalized usage scenario for the latch requires to arbitrarily split the resource invariant for the worker threads, *e.g.*, in such a way that each worker has an exclusive access to part of the resource, rather than a shared access to the whole resource, or, in fact, any combination of the two possibilities. Upon completion of all n countDown calls, the resource invariant is reconstructed into a full one and then split again according to another division schema for the m awaiting threads.

```
/*@ given group (frac -> resource) inv;  @*/
public class CountDownLatch {
/*@ resource held(frac p);  @*/

/*@
  requires count > 0;
  ensures initialized(1);
  ensures (\forall* int i;0 <= i && i < count; held(1/count)); @*/
 public CountDownLatch(int count);

/*@
 requires held(1/count) ** inv(1/count);@*/
 public void countDown();

/*@
 given frac p;
 requires initialized(p);
 ensures  inv(p);  @*/
 public void await();
}
```

Listing 19: Simplified specification of the CountDownLatch class.

The core idea for this generalized case is to provide separate splits of the resource for the countDown and await operations, and carry around the identity of the corresponding calling thread in an integer parameter of the corresponding predicate. In total, there are four groups of predicates involved in the specification (see Listing 20):

- cdRes(i) represents a resource that an i-th worker thread calling the countDown method works with,
- held(i) represents a permission for the i-th worker thread to actually call countDown,
- awRes(j) represents a resource that the j-th awaiting thread shall receive,

```
/*@
given (int −> resource) cdRes;
given (int −> resource) awRes;
given int nCountDown, mAwait; @*/
public class CountDownLatch{
  /*@ resource held(int i), awCall(int j); @*/

/*@
 requires count == nCountDown;
 requires
    (\forall* int i; 0 <= i && i < nCountDown; cdRes(i))
    *−* (\forall* int j; 0 <= j && j < mAwait; awRes(j));
 ensures (\forall* int i; 0 <= i && i < nCountDown; held(i));
 ensures (\forall* int j; 0 <= j && j < mAwait; awCall(j)); @*/
 public CountDownLatch(int count);

/*@
 given int i;
 requires held(i) ** cdRes(i); @*/
 public void countDown();

/*@
 given int j;
 requires awCall(j);
 ensures awRes(j); @*/
 public void await();
}
```

Listing 20: Improved specification of the CountDownLatch class.

- and awCall(j) represents a permission for the j-th awaiting thread to call await.

In principle the i and j represent different numbers of different worker and awaiting threads. Furthermore, all these predicates are *not* declared to be **group** and they are not parametrised by a fraction, *i.e.*, the resource shares for all the threads involved with the latch are predefined up front and are not allowed to be further split by any of the threads. Otherwise,

the specifications would become unnecessarily complex to cover scenarios that are not really found in practice (*i.e.*, when resources are lost during the operation of the latch). The soundness requirement for the specification is that the whole group of cdRes(i) predicates is resource equivalent to the whole group of awRes(j) predicates. That is, the following should hold:

```
(\forall* int i; 0 <= i && i < nCountDown; cdRes(i))
    *-* (\forall* int j; 0 <= j && j < mAwait; awRes(j))
```

where $*-*$ denotes separating equivalence, while nCountDown and mAwait correspond to the numbers of worker and awaiting threads, respectively. Virtually, both of these groups of resources would be equivalent to the regular resource invariant inv(1) that we used in the simplified specification.

Overall, this gives rise to the CountDownLatch specification given in Listing 20. Some of the main differences compared to the simplified specification in Listing 19 are the following. As mentioned, the threads that work with the latch are not allowed to split the given resource in any way and every thread involved with the latch has its own resource predicate. Hence, there is no **frac** parameter to any of the predicate class parameters. Moreover, it is the responsibility of the specifier of the client code to provide the partitioning of the resources between the worker threads and awaiting threads by providing appropriate definitions for cdRes and awRes. The specification only checks that the two groups of predicates are equivalent, see line 11 in Listing 20, to maintain soundness. Also, the number of worker and awaiting threads has to be known a priori. This is not limiting for the worker threads, as the API itself requires to pass the count parameter to the constructor, but the client code is not free to allow for arbitrary await calls. This was not the case in the simplified specification, where the initialized(p) predicate could have been split arbitrarily after the construction of the latch and consequently allow for arbitrary many await calls. Otherwise, the specification allows for arbitrary resource distribution between all involved threads. The constructor issues nCountDown and mAwait tickets for the worker and awaiting threads, respectively (lines 13–14 in Listing 20). Then, each worker thread presents its ticket and the corresponding part of the resource upon the call to countDown, at which point the resource is consumed by the latch (lines 18–20 in Listing 20). Then, the awaiting threads present their tickets when calling await and receive their corresponding part of the resource (lines 23–26 in Listing 20).

4.4 CyclicBarrier Specification

The CyclicBarrier class in Java is useful when a set of threads has to wait for each other to reach a common execution point. The difference with CountDownLatch is that the waiting threads can re-use the CyclicBarrier after they have finished a part of their execution. So, essentially a barrier allows a group of threads to periodically re-synchronise. A barrier object is instantiated with a fixed number of participating threads, called parties. Each thread, finishing its portion of the global task, calls the blocking await() method and waits for the other threads to enter the barrier. When all the threads are in the barrier, the await() method returns, upon which all the waiting threads start the next stage (cycle) of their work.

Listing 21 presents our specification for the CyclicBarrier class. Using a barrier, the participating threads in each cyclic round call the await() method to get synchronised. Therefore, using a similar mechanism as with the improved specification of the CountDownLatch, the resource that the barrier protects is parametrized with the identifier of the thread that calls await() and a cycle number of the barrier. Also, as in the latch, the threads that work with the barrier are not allowed to further split the resource. When entering the barrier, all the participating threads have to temporarily release the resource corresponding to the given cycle, represented by res(c, i) (line 15). When leaving the barrier, each thread obtains its portion of the resource corresponding to the new cycle number res(c+1, i) (line 16). The handle predicate represents the permission ticket to make a cyclic call to await(). These tickets are issued by the barrier's constructor, which witness the current cycle number of the calling thread. Similar to the specification of the latch, the main thread is responsible for distributing the actual resource among the worker threads.

The CyclicBarrier class as defined in the Java API contains another constructor which takes an instance of a class implementing the Runnable interface. This parameter assigns a task to the barrier object to be internally executed inside the barrier when all the parties arrive at the barrier. Specifying this barrier constructor is considerably more challenging. In particular, we would need to provide a very specific and detailed specification of the otherwise very general Runnable interface that would allow us to connect the Runnable's resources with the barrier and worker threads' resources in a sound way. We were not yet able to come up with a correct specification for this case, this is part of our future work.

```
/*@
 given (int, int -> resource) res; @*/
 public class CyclicBarrier {

/*@
 resource handle(int c, int p);  @*/

 /*@
 requires parties > 0;
 ensures (\forall* int i; i>=0 && i<parties; handle(0,i)); @*/
 public CyclicBarrier(int parties);

/*@
 given int i, c;
 requires handle(c,i) ** res(c,i);
 ensures handle(c+1,i) ** res(c+1,i);  @*/
 public int await();

}
```

Listing 21: Specification of the CyclicBarrier class.

4.5 Examples

In this section we discuss the verification of several client programs that are synchronised with ReadWriteLock, CountDownLatch and CyclicBarrier, making use of our specifications.

Example use of ReadWriteLock

In this example we show how the specification of ReadWriteLock helps us to reason about a single-producer multiple-consumer application. Assume an application where a single producer produces data to be used by two separate consumer threads. The producer implemented as a Producer class (see Listing 22) obtains the write lock and then exclusively accesses the shared data field. Then, each consumer (see Listing 23) tries to obtain a fractional permission of the shared data to use the value written by the producer. The producer and consumers are then combined together in class SProdMCons

```
/*@ given group (frac -> resource) pcinv; @*/
public class Producer extends Thread {
  private final Lock/*@<pcinv, true, true>@*/ lock;
  private final SProdMCons example;

  /*@
  given frac p;
  requires lock.initialized(p);
  ensures lock.initialized(p);  @*/
  public void produce(){
      /*! { lock.initialized(p) } !*/
      lock.lock();
      /*! { lock.inv(1) ** lock.held(p) } !*/
      /*@ unfold lock.inv(1); @*/
      /*! { Perm(example.data, 1) ** lock.held(p) } !*/ // from pcinv
      sample();
      /*! { Perm(example.data, 1) ** lock.held(p) } !*/
      /*@ fold lock.inv(1); @*/
      lock.unlock();
      /*! { lock.initialized(p) } !*/
  }
  // method run
}
```

Listing 22: The producer.

implemented in Listing 24.

Example use of CountDownLatch

In Listing 25 we present a simple annotated client program that uses our improved latch specification. Each of the worker threads (instances of the Worker class) works with an exclusive access to one element of the locations array of the Example class. This is specified with the cdRes that returns a single full permission to one array element given by the i parameter. In this example there are N worker threads, but only one awaiting thread. Hence, the awRes predicate returns full permission to the whole locations array if j==0, this is specified with the separating quantification \forall*,

```
/*@ given group (frac -> resource) pcinv; @*/
public class Consumer extends Thread {
  private final Lock/*@<pcinv, false, true>@*/ lock;
  private final SProdMCons example;
  private boolean flag;
  private int value;

  /*@
  given frac p;
  requires lock.initialized(p) ** Perm(this.value,1);
  ensures lock.initialized(p) ** Perm(this.value,1); @*/
  public void consume(){
      /*! { lock.initialized(p) } !*/
      lock.lock();
      /*! { lock.inv(p) ** lock.held(p) } !*/
      /*@ unfold lock.inv(p); @*/
      /*! { Perm(example.data, p) ** lock.held(p) } !*/  // from pcinv
      this.value = example.data;
      if( flag == this.example.PRINT) print( );
      if( flag == this.example.LOG) log( );
      /*@ fold lock.inv(p); @*/
      /*! { lock.held(p) ** lock.inv(p) }  !*/
      lock.unlock();
      /*! { lock.initialized(p) }  !*/
  }
  // methods run, print and log
}
```

Listing 23: The consumer.

otherwise no permissions are granted. These two predicates are passed to the CountDownLatch constructor along with the numbers of worker and awaiting threads, N and 1 respectively. The pre-condition of the constructor that all cdRes and awRes are resource equivalent is clearly met. At this point the main thread executing in the splitWork method is in possession of the complete set of held and awCall predicates (from the post-condition of the CountDownLatch constructor) and full permissions to access all elements of

```
public class SProdMCons {
  /*@ group resource pcinv(frac p) = Perm(data, p);  @*/
  public final boolean PRINT = true, LOG = false;
  public int data;
  private ReadWriteLock/*@<pcinv, true>@*/ rwl;

  void main(){
    rwl = new ReentrantReadWriteLock/*@< pcinv >@*/();

    Producer producer =
        new Producer/*@< pcinv >@*/(this, rwl.writeLock());
    Consumer printer =
        new Consumer/*@< pcinv >@*/(this, PRINT, rwl.readLock());
    Consumer log =
        new Consumer/*@< pcinv >@*/(this, LOG, rwl.readLock());

    producer.start(); printer.start(); log.start();
    producer.join(); printer.join(); log.join();
  }
}
```

Listing 24: Implementation of SProdMCons.

locations (from the pre-condition of splitWork()). When the worker threads are started in the loop, the corresponding held and cdRes predicates required by the pre-condition of run() method can be passed on through the call to start().

The awCall(0) predicate remains with the main thread and is used by the call to doneSignal.await() to regain the complete permission to the locations array through the awRes(0) predicate. This in turn establishes the post-condition of the splitWork() method.

Example use of CyclicBarrier

Assume the scenario implemented in Listings 26 and 27 where a set of threads is collecting data in a matrix to be plotted by a plotter thread. In each cycle of the process the plotter plots the currently available data, while in parallel the collector threads are collecting data to be displayed

in the next round. The plotter thread is the thread with an $id = 0$ and any thread with $id > 0$ is considered as a collector thread. There is a global two-layered two-dimensional matrix (line 3). In each cycle the plotter thread displays data related to the current cycle (line 18) in one layer, in parallel the collector threads are collecting (new) data in the other layer to be plotted in the next cycle (line 19). That is, the plotter thread and the collector threads use the two layers of the matrix interchangeably in between the subsequent barrier calls. Each collector thread is responsible to fill one row of the two-dimensional matrix.

The res(c, t) represents (1) the full permission of the elements of one layer (current cycle) for the plotter thread (line 15), and (2) the full permission of the elements of one row from other layer (next cycle) with index t for the collector thread associated with t (line 16).

When initializing the CBExample class, the full permission of the matrix in layer 0 is transferred to the plotter thread (line 20) and the full permission of each row from layer 1 of the matrix is transferred to the corresponding collector (line 22). In the first round of the execution, the plotter displays the initialized data. In parallel, the collectors are filling data for layer 1 to be displayed in the next round. The plotter thread is synchronised with all the collector threads through the barrier call (line 20), after which each thread increments its cycle number to change its associated layer of the matrix.

The barrier is in some ways similar to the count down latch, yet there are some crucial differences. The exchange of the resources between the worker threads that use the barrier happens in one place, *i.e.*, upon the call to await. In the count down latch, this exchange is done in two phases – the resources are transferred into the latch upon countDown and out of the latch upon await. Moreover, the latch has only one cycle, while the barrier can have infinitely many. This is reflected in the specification of the barrier with the additional integer parameter c to the res and handle predicates that represents the cycle number. Finally, note that a similar numbering mechanism could have been used to merge the two predicates cdRes and awRes into one, where the corresponding phase (count down or await) would be identified with a cycle number equal to 0 or 1, respectively. However, we believe it would make the client code specification unnecessarily more complex.

4.6 Conclusion and Related Work

In this chapter, based on examples of the family of Java locks, the semaphore, the count-down latch, and the cyclic barrier from the Java API, we presented a generalized approach for handling synchronisation primitives in permission-based Separation Logic for concurrent Java. We lift all mechanisms associated with synchronisation handling, and the corresponding permission transfer, to the specification layer of the logic. This way we provide a modular verification mechanism that is applicable to arbitrary concurrent Java programs, and we enable the verification of the synchronisation routine implementations themselves (as discussed in the next chapter).

The work presented here extends earlier formalisation of reentrant locks [46, 10]. Several other built-in formalizations of locks and synchronisation primitives exist. The Chalice system [58] formalizes simple non-reentrant locks built into the Chalice language. The work of Gotsman *et al.* [40] is similar to our earlier formalization, and we believe that our high-level approach could also be easily applied there to treat a wider range of synchronisation primitives. Similarly, the work of Hobor and Gherghina on formalizing barriers in Separation Logic [45] follows very similar principles that we used in our barrier specification presented in this paper. Finally, the VeriFast tool [49] adopts an approach similar to ours where locking is also specified on the API level, but only for simple and non-reentrant locks, and so-called higher-order abstract predicates are functionally similar to our class level specification parameters.

```
class Example {
  public static final int N = 20;
  private int[] locations = new int[N];
/*@ requires (\forall* int i; i>=0 && i<N; Perm(locations[i], 1));
 ensures  (\forall* int i; i>=0 && i<N; Perm(locations[i], 1)); @*/
 void splitWork() throws InterruptedException {
 /*@ resource cdRes(int i) = Perm(locations[i], 1);
    resource awRes(int j) = j != 0 ? true :
      (\forall* int k; k>=0 && k<N; Perm(locations[k],1)); @*/
    CountDownLatch/*@<cdRes,awRes,N,1>@*/ latch =
      new CountDownLatch/*@<cdRes,awRes,N,1>@*/(N);
    for (int i = 0; i < N; ++i) {
      new Thread(new Worker(this, i, latch)).start();
    }
    doneSignal.await();
  }
  public int[] getLocations() { return locations; }
}

class Worker implements Runnable {
  private final CountDownLatch latch;
  private final Example example;
  private final int id;

  Worker(Example example, int id, CountDownLatch latch) {
    this.example = example; this.id = id; this.latch = latch;
  }
  /*@ requires latch.held(id) ** latch.cdRes(id); @*/
  public void run() {
    try {
      shared.getLocations()[id] = id;
      latch.countDown();
    } catch (InterruptedException ex) {}
  }
}
```

Listing 25: Client code using CountDownLatch

```
class CBExample {
  public static final int N = 20, M = 50;
  private int[][][] matrix = new int[2][N][M];

 /*@
  resource row(int h, int r) =
       (\forall* int i; i >= 0 && i < M; Perm(matrix[h][r][i], 1)); @*/

 /*@
  requires (\forall* int i; i>=0 && i<N; row(0, i) ** row(1, i));
  ensures (\forall* int i; i>=0 && i<N; row(0, i) ** row(1, i));  @*/
  void splitMatrix() throws InterruptedException {
    /*@
     resource res(int c, int i) = (i==0) ?
         (\forall* int i; i>=0 && i<N; row((c % 2), i)) :
         row(((c+1) % 2), i-1);  @*/
    CyclicBarrier /*@<res>@*/ barrier =
        new CyclicBarrier/*@<res>@*/(N);

    new Thread(new Process(this, 0, barrier)).start();

    for (int i = 1; i < N+1; i++) {
       new Thread(new Process(this, i, barrier)).start();
    }
  }

  public int[] getRow(int h,int i) { return matrix[h][i]; }
}
```

Listing 26: Client code using the CyclicBarrier

```
class Process implements Runnable {
  private final CyclicBarrier barrier;
  private final CBExample example;
  private final int id;
  private int cyc;

  Process(CBExample example, int id, CyclicBarrier barrier) {
      this.example = example;
      this.id = id;
      this.barrier = barrier;
      cyc = 0;
  }
  public void run(){  while(true) process();  }
 /*@ given int c;
  requires barrier.handle(c, id) ** barrier.res(c, id);  @*/
  private void process() {
      try {
        if( id == 0 ) plot();
        if( id > 0 ) collect();
        barrier.await();
        this.cyc++;
      } catch (InterruptedException ex) {}
  }

/*@ requires (\forall* int i; i>=0 && i<N; row((cyc % 2), i));
    ensures (\forall* int i; i>=0 && i<N; row((cyc % 2), i));  @*/
  public void plot(){
 /* display matrix[(cyc+1)%2][n][m] for 0 ≤m<M and 0≤n<N */
  }

/*@ requires row(((cyc+1) % 2), id−1);
    ensures row(((cyc+1) % 2), id−1);  @*/
  public void collect(){
/* fill matrix[(cyc+1)%2][id−1][m] for 0 ≤m<M */ }
}
```

Listing 27: The processing code for the example in Listing 26.

CHAPTER 5

Verification of Synchronisers: Exclusive Access

To reason about concurrent programs, it is vital to have the contracts of the synchronisation classes. In Chapter 4 we have presented the contracts of several synchronisation classes provided in java.util.concurrent. Our proposed specifications are justified by a set of examples. However, still the correctness of these specifications has not been discussed. In this chapter we propose a technique to verify the correctness of the implementation of the synchronisation mechanisms with respect to their specification.

In Chapter 2 we have discussed how *atomic classes* are used to implement synchronisation mechanisms. This chapter first identifies which different synchronisation patterns can be implemented by using atomic operations. Then, it proposes an approach to specify the behavior of an *atomic class* as a synchroniser. Additionally, the approach is used to specify Java's AtomicInteger class, and it is discussed how different synchronisation mechanisms can be built and verified using atomic integer as the synchronisation primitive.

In our approach, any thread has a local view of the atomic variable. The global state is then defined in terms of the atomic variable and all the local views. In addition, the atomic variable is instrumented with a protocol that describes what the legal state transitions are. The protocol is used by the thread to derive the guarantees that the environment provides. Additionally, a resource invariant is declared, which specifies which resources are protected by the synchroniser. The derived specifications for the AtomicInteger operations are thus parameterized by this protocol and the resource invariant. This specification expresses how AtomicInteger, as an atomic synchroniser, grants and retains permissions to access the shared resource specified by the resource invariant *exclusively*. To describe the specifications and the predicates encoding the views and the protocol, we use permission-based Separation Logic for Java [23, 10].

The specifications for the methods in the AtomicInteger class are derived from the classical concurrent Separation Logic rule [85] for atomic operations. A main characteristic of our specification is its ease of use. To verify an implementation of a synchronisation mechanism, only a few intuitive parameters have to be provided. Particularly, the user only has to specify (1) what are the different roles of the threads participating in the synchronisation, (2) what are the legal state transitions in the synchroniser, and (3) what share of the resource invariant can be obtained in a certain state, given the role of the current thread. For all implementations, we provide

a machine-checked proof that the implementations correctly implement the synchroniser.

This chapter is structured as follows: in Section 5.1 we introduce several synchronisation patterns using AtomicInteger as a synchronisation primitive, each supported by an example. Section 5.2 derives contracts for atomic read, write, and compare-and-set. Section 5.4 explains the generalized specification of the AtomicInteger class and discusses correctness proofs of the clients using AtomicInteger. Finally, Section 5.5 draws conclusions and presents related work.

5.1 Synchronisation Patterns

As we discussed in Chapter 2, to support thread-safe access to single variables, Java provides the package java.util.concurrent.atomic, as part of Java's general concurrency API. This package provides wrappers for volatile variables with appropriate atomic operations for read, write, and compare-and-swap. As a commonly used atomic class, the AtomicInteger class encapsulates a volatile field of type integer. Essentially, it provides the following methods: get(), returning the value that was last written to the field; set(**int** v), atomically assigning the value v to the field; and compareAndSet(**int** x,**int** n), atomically checking the current value and updating it to n, if it is equal to the expected value x, otherwise leaving the state unchanged, and returning a boolean to indicate whether the update succeeded.

In a shared-memory concurrency setting, two kinds of thread interactions via a synchroniser can be distinguished: *cooperation* and *competition* [75]. In a cooperative interaction, threads employ a *cooperative synchroniser* as a communication channel to cooperatively share a resource. In a competitive interaction, a *competitive synchroniser* runs a competition and provides (temporary) access to the shared resource to the winner. A synchroniser can behave cooperatively or competitively in different states, this is called a *hybrid* interaction. Various patterns of synchronisation can be described in terms of atomic integer operations:

Pattern GS (get and set)

Threads can cooperatively interact using atomic read and write. Every thread has a designated state in which it obtains the resource, and all threads attempt to reach their designated state. When a thread writes

```
public class ProducerConsumer{
  private final int E = 0, F=1;
  private AtomicInteger sync;
  private int data; // shared buffer

  ProducerConsumer(){ sync = new AtomicInteger(E); }

  void produce(){
    write();
    sync.set(F); // signal
    while(sync.get() == F); // wait
  }
  void consume(){
    while(sync.get()==E);  // wait
    read();
    sync.set(E); // signal
  }
  // methods write() and read()
}
```

Listing 28: ProducerConsumer: cooperation.

to the atomic integer, it implicitly signals who *should* own the resource next (cooperation). Based on the value written into the synchroniser, ownership of the resource is transferred to the appropriate thread waiting for that particular value. Producer-Consumer and Dekker's critical section algorithm are examples of this pattern. Listing 28 shows ProducerConsumer with two methods produce and consume, sharing a field data, that implements this algorithm. Typically, these methods will be executed as part of a surrounding loop. The AtomicInteger denotes the state of the buffer: full (F) or empty (E). Both the producer and the consumer wait until the buffer gets into their desired state. As soon as the state changes to the expected value, the waiting thread obtains the shared resource. When it is done, it changes the state, so that the other thread can access the resource.

```
public class SpinLock{
  private final int U = 0, L=1;
  private AtomicInteger sync;
  SpinLock(){ sync = new AtomicInteger(U); }

  void lock(){
    while(!sync.compareAndSet(U,L));
  }

  void unlock(){
    sync.set(U);
  }
}
```

Listing 29: SpinLock: competition.

Pattern SC (set and compareAndSet)

Atomic write and conditional update can be used to implement a competitive synchroniser. Threads are competing to obtain the protected resource by calling compareAndSet. A thread that succeeds in changing the state, obtains the resource. When it no longer needs the resource, it sets the state to the initial value, to signal its availability. Failing threads continue to try to acquire the resource by checking whether the state is reverted back to the initial state. A spin-lock implementation using AtomicInteger (see Listing 29) is a known example of this pattern where the atomic integer value encapsulates the state of the lock: locked (L) or unlocked (U). If a thread successfully updates the state from U to L, it acquires the lock (method lock). Consequently, failing threads enter a try-wait loop, until the lock is released. To release the lock, the thread holding the lock executes set(U) (method unlock).

Pattern GC (get and compareAndSet)

Atomic read and compare-and-set are suited to implement a synchronisation mechanism that *partially* transfers the resources between the participating threads. Shared reading synchronisation mechanisms using AtomicInteger like Semaphore and CountDownLatch are typical instances of this pattern. Also lock-free pointer-based data structures using AtomicReference are ex-

amples of this pattern. Since, here, we are only looking at exclusive synchronisation mechanisms, we do not discuss this pattern further. However, a generalization of our approach to reason about partial resource ownership using atomics is the topic of Chapter 6.

Pettern GSC (get, set and compareAndSet)

All basic operations of AtomicInteger can be used together to implement a hybrid synchroniser. Threads compete with each other to obtain the resource by calling compareAndSet. A thread that succeeds in changing the state, wins the resource. Failing threads may not compete any more to change the state. But, they have to wait for the resource availability. When the winner thread no longer needs the resource, it updates the state to signal how the resource should be used afterwards. Listing 30 shows the implementation of a SingleCell algorithm, which illustrates a hybrid pattern[1]. It provides a single method to find or put a value in a shared storage cell. The storage cell is always in one of these states: empty (E), writing (W) or done (D). The cell containing the value (the state D) must be immutable. Initially, all threads are competing to assign their value. If a thread succeeds in obtaining writing access to the resource, the state becomes W. After completing the assignment, it will report its success (returns PUT). All other threads have to wait until the value is assigned, and then they check the stored value. If the value in the cell is equal to the value the thread holds, it will return the value SEEN, otherwise it will signal a collision (returns COLN).

5.2 Ownership Exchange via Atomics

In this section we show how the contracts in permission-based Separation Logic for the basic atomic operations can be derived. We base ourselves on the work by Vafeiadis [85], which enables us to define a language where atomic commands, denoted atomic{ C }, are the only constructs for synchronisation.

We divide the domain of the heap into a set of *atomic* locations ALoc (*e.g.*, the volatile field of AtomicInteger) and a set of *non-atomic* locations

[1] This is a simplified version of a lock-less hash table, especially designed for state space exploration in the multi-core model checker LTSmin [56].

```
public class SingleCell{
  final private int E = 0, W=1, D=2;
  final private int PUT = 0, SEEN = 1, COLN = 2;
  private AtomicInteger sync;
  private int data;

  SingleCell(){ sync=new AtomicInteger(E); }

  int findOrPut(int v){
    if(sync.compareAndSet(E,W)){ data=v; sync.set(D); return PUT; }
    if(sync.get()!=E){
      while(sync.get()==W); // wait
      if(sync.get() == D)
        if(data == v)  return SEEN;
        else  return COLN;
    }
  }
}
```

Listing 30: SingleCell: hybrid.

NLoc (*e.g.*, data in Listing 28). An atomic location $s \in$ ALoc may only be accessed using:

1. get(s) for atomic read of the atomic location s,
2. set(s, n) for atomic update of s with n, and
3. cas(s, x, n) for atomic conditional update of s.

We use the term *atomic value* to refer to the value that an atomic variable contains and the term *resources* to refer to *non-atomic* locations of the heap.

As proposed by O'Hearn, in a concurrent setting a resource invariant is attached with a synchroniser. This associates ownership of a part of the state space with possible states of the synchroniser [68]. For example, the resource invariant for a lock lock $\in$ ALoc that protects the resource x $\in$ NLoc is defined as:

$$I_{\mathsf{lock}} = \exists\, v \in \{0,1\}.\ \mathsf{lock} \overset{1}{\mapsto} v * ((v = 1 \implies \mathsf{emp}) * (v = 0 \implies \mathsf{x} \overset{1}{\mapsto} -))$$

This expresses that full ownership of the location x is available to win when $[\mathsf{lock}] = 0$, while if $[\mathsf{lock}] = 1$ then emp (interpreted as nothing) can be obtained.

In general, using a function res that maps an atomic value to a set of disjoint resources, given Val as the set of values and $s \in \mathsf{ALoc}$, the resource invariant I_s is defined as:

$$I_s = \exists\, v \in \mathsf{Val}.\ s \overset{1}{\mapsto} v * \mathsf{S}(s, v) \quad \text{where} \quad \mathsf{S}(s, v) = \underset{r \in \mathsf{res}(s,v)}{\circledast} r \overset{1}{\mapsto} -$$

In CSL, a judgment $I \vdash \{P\}\ C\ \{Q\}$ expresses the following: given a globally accessible resource invariant I and a local pre-condition P, if a statement C starts its execution in a state satisfying $P * I$, and if C terminates, then its final state satisfies $Q * I$. The proof rule for atomic commands [85] expresses that to prove correctness of $\mathsf{atomic}\{\ C\ \}$, the resource invariant I can be used for the verification of the atomic body C. Thus, I is not accessible to the environment. Moreover, within the body C, the resource invariant I may be invalidated, because it is not visible to the environment, but it must be re-established before C is finished:

$$\frac{\mathsf{emp} \vdash \{P * I\}\ C\ \{I * Q\}}{I \vdash \{P\}\ \mathsf{atomic}\{\ C\ \}\ \{Q\}}\ [\textsc{Atomic}]$$

We use the rule [Atomic] to derive specifications for the basic atomic operations `get`, `set` and `cas` when they are coordinating a set of threads to (exclusively) access a shared resource. The specifications should capture all exclusive synchronisation patterns mentioned above: cooperative, competitive and hybrid. Therefore, we need to enrich the resource invariant definition with an abstraction of local state and feasible states, which allows one to deduce what the environment guarantees. Next, we instantiate the [Atomic] rule to derive the resources that `set`, `get` and `cas` exchange to perform exclusive access synchronisation.

Synchronisation Protocol

Assuming a set of threads Thr, for each atomic location s that is synchronising the threads, we define the *view* of a thread $t \in \mathsf{Thr}$ as an *atomic ghost variable*, denoted s_t. Each thread stores the last visited atomic value in its view. We define the view to be atomic in order to restrict the thread t, using ghost code, to update *its* view *only* inside an atomic block. To do

so, the ownership of a view is split in half between the owner thread and the resource invariant, *i.e.* the shared state. Therefore, a thread can always read its own view, but it can only update its view when it captures the other half permission inside an atomic block by accessing the resource invariant. Views of threads indexed by thread identifiers are written as a vector of views $\overrightarrow{s_t}$. Similarly, $\overrightarrow{v_t}$ denotes a vector of values pointed to by the views, indexed by the corresponding thread identifiers, while $\overrightarrow{v_t}_{\{v_\tau=x\}}$ denotes a vector such that the item indexed with τ is equal to x. For the sake of simplicity we assume that there is only one single atomic location s functioning as the synchroniser. However, the approach is generalizable for multiple atomic location.

We define the *(global) atomic state* as a tuple of the atomic value and all thread local views of it, denoted $(s, \overrightarrow{s_t})$. An atomic state is *admissible* if at least one thread has a correct view of the synchroniser. An admissible atomic state is *feasible* if either (1) it is an initialization state where all the threads have an identical view of the initialized atomic location, or (2) it is reachable from the initialization state by a finite set of atomic operations.

As the views must be updated only inside the atomic operations, they can reflect the actions that the environment can perform w.r.t. the atomic location. The current definition of the resource invariant is too restrictive to reflect this. So, first, we define the protocol of the synchroniser in terms of the atomic state:

$$\mathsf{P}_s^{\mathsf{Thr}} = \bigvee_{v, \overrightarrow{w_t} \in \mathsf{Val} \cdot \mathsf{fsbl}(v, \overrightarrow{w_t})} ([s] = v \wedge [\overrightarrow{s_t}] = \overrightarrow{w_t})$$

where fsbl determines whether the *atomic state* is feasible.

Example 5.2.1. *Protocol for ProducerConsumer*

To illustrate our definition of feasible states, consider ProducerConsumer, where we have two threads p (producer) and c (consumer) with corresponding views, i.e. $\mathsf{s_p}$ and $\mathsf{s_c}$, respectively, given an atomic variable s:

$$P_{\mathsf{s}}^{\{\mathsf{p},\mathsf{c}\}} = \\ (([\mathsf{s}] = E \wedge [\mathsf{s_p}] = E \wedge [\mathsf{s_c}] = E) \vee ([\mathsf{s}] = F \wedge [\mathsf{s_p}] = F \wedge [\mathsf{s_c}] = E) \vee \\ ([\mathsf{s}] = F \wedge [\mathsf{s_p}] = F \wedge [\mathsf{s_c}] = F) \vee ([\mathsf{s}] = E \wedge [\mathsf{s_p}] = F \wedge [\mathsf{s_c}] = E))$$

Note that $([\mathsf{s}] = F, [\mathsf{s_p}] = E, [\mathsf{s_c}] = F)$ is not a feasible state. Therefore, when p believes that the buffer is empty (E), it can safely rely *on the fact*

that no other thread is allowed to modify s *to full (F). Thus,* p *deduces that it exclusively owns* s*, so* $[\mathsf{s}]$ must be E *when* $[\mathsf{s_p}] = E$*.*

Example 5.2.2. *Protocol for SpinLock*

Consider the SpinLock example, which is a competitive pattern. Its protocol is defined as follows:

$$\begin{aligned} P_s^{\mathsf{Thr}} = \quad & ([\mathsf{s}] = U \ \wedge (\forall\ t \in \mathsf{Thr}.\ [\mathsf{s}_t] = U)) \ \vee \\ & ([\mathsf{s}] = L \ \wedge (\exists\ \tau \in \mathsf{Thr}.\ [\mathsf{s}_\tau] = L \ \wedge \forall\ t \in \mathsf{Thr} \setminus \{\tau\}.\ [\mathsf{s}_t] = U)) \end{aligned}$$

This expresses that either the lock is available and all threads have a correct view of the state, or there is only one thread that has acquired the lock and updated its view while all others have failed to change their beliefs. This makes it possible for the unlocking thread to rely on its view, knowing that it will be the only one that has the correct view.

The protocol suffices to derive the contracts for the basic atomic operations when they are involved in a *competitive* pattern. To cover cooperative patterns, where *threads obtain the shared resources based on their views*, in addition, the resource invariant has to express what resources are protected in terms of the atomic state. In fact, instead of one single atomic variable s, $(s, \vec{s_t})$ plays the role of a global synchroniser. Similar to res, we define ares to map the atomic state to a set of disjoint resources. Therefore, we replace $\mathsf{S}(s, v)$ with $\mathsf{R}(s, v, \vec{s_t}, \vec{w_t})$ to denote all the resources associated with $[s] = v$ and $[\vec{s_t}] = \vec{w_t}$.

Now we are ready to define precisely what we mean by a synchronisation primitive, based on our extended definition of resource invariant.

Definition 5.2.1. *State-based synchroniser*

An atomic location s together with the basic atomic operations $\mathsf{ACmd} = \{\mathit{get}, \mathit{set}, \mathit{cas}\}$ *define a* state-based primitive synchronisation mechanism *for a set of threads* Thr *if it is instrumented with a resource invariant defined as follows:*

$$I_s = \exists v, \vec{w_t} \in \mathsf{Val} \cdot s \overset{1}{\mapsto} v \ast (\underset{t \in \mathsf{Thr}}{\circledast} s_t \overset{\frac{1}{2}}{\mapsto} w_t) \ast R(s, v, \vec{s_t}, \vec{w_t}) \ast P_s^{\mathsf{Thr}}$$

where $R(s, v, \vec{s_t}, \vec{w_t}) = \underset{r \in \mathit{ares}(s, v, \vec{s_t}, \vec{w_t})}{\circledast} r \overset{1}{\mapsto} -.$

Example 5.2.3. *Synchroniser for ProducerConsumer*

Based on the protocol defined in Example 5.2.1, we define the resource invariant of the atomic synchroniser s *to synchronise* p *and* c*:*

$$I_s = \exists v, w_p, w_c \in \{E, F\} \cdot s \overset{1}{\mapsto} v * s_p \overset{\frac{1}{2}}{\mapsto} w_p * s_c \overset{\frac{1}{2}}{\mapsto} w_c * R(s, v, \vec{s_t}, \vec{w_t}) * P_s^{\{p,c\}}$$

where $R(s, v, \vec{s_t}, \vec{w_t})$ *is* $\texttt{data} \overset{1}{\mapsto} -$ *if* $v = E$, $w_p = F$, $w_c = E$ *and* $v = F$, $w_p = F$, $w_c = E$, *and* $R(s, v, \vec{s_t}, \vec{w_t})$ *is* emp *if threads agree on the value of* s*. This expresses that* s *holds the full ownership of* $\texttt{data}$ *when threads do not agree on the value of the synchroniser (*i.e.*, during the transition phase).*

Example 5.2.4. *Synchroniser for SpinLock*

Considering the SpinLock protocol in Example 5.2.2, we define the resource invariant for s*. Here, regardless of the views of the threads, the resource invariant holds the full resource when the state is* U*, otherwise the winning thread holds it.*

$$I_s = \exists v, \vec{w_t} \in \{U, L\} \cdot s \overset{1}{\mapsto} v * (\underset{t \in \mathsf{Thr}}{\circledast} s_t \overset{\frac{1}{2}}{\mapsto} w_t) * R(s, v, \vec{s_t}, \vec{w_t}) * P_s^{\mathsf{Thr}}$$

where $R(s, v, \vec{s_t}, \vec{w_t})$ *will be* $\texttt{data} \overset{1}{\mapsto} -$ *when* $v = U$ *and* emp *when* $v = L$.

Next we investigate how the three basic atomic operations can exchange the shared resources.

5.3 Specifications of Atomics

This section derives contracts for the three basic atomic operations for state-based synchronisation. The contracts, shown in Figure 5.1, essentially express that in an exclusive state-based synchronisation, the thread τ executing an atomic operation to update the state of the synchroniser, should *provide* the resources associated with the state after the operation, and in return will *receive* the resources associated with the previous state of the synchroniser. In Figure 5.1, we used $R_s^{\mathsf{Thr}}(\tau, x, y)$ to denote all the resources when $s = x$ and $s_\tau = y$. First, we explain the specification for each atomic operation. Then, we present their derivations, and finally, we discuss how the specification can be adopted to thread-modular contracts.

$$\textit{Let } R_s^{\mathsf{Thr}}(\tau, x, y) = \bigcircledast_{\vec{v_t}_{\{v_\tau = y\}} \in \mathsf{Val}.\mathsf{fsbl}(x, \vec{v_t}_{\{v_\tau = y\}})} \mathsf{R}(s, x, \vec{s_t}, \vec{v_t}_{\{v_\tau = y\}})$$

$$\frac{\forall\ v, \vec{v_t} \in \mathsf{Val}.\ v_\tau = \mathsf{d} \wedge \mathsf{fsbl}(v, \vec{v_t}_{\{v_\tau = \mathsf{d}\}}) \implies \mathsf{fsbl}(n, \vec{v_t}_{\{v_\tau = n\}})}{I_s \vdash \begin{array}{c}\{s_\tau \overset{\frac{1}{2}}{\mapsto} \mathsf{d} * R_s^{\mathsf{Thr}}(\tau, n, n)\} \\ \mathtt{set}_\tau(s, n) \\ \{s_\tau \overset{\frac{1}{2}}{\mapsto} n * R_s^{\mathsf{Thr}}(\tau, \mathsf{d}, \mathsf{d})\}\end{array}} \text{[WATM]}$$

$$\frac{}{I_s \vdash \begin{array}{c}\{s_\tau \overset{\frac{1}{2}}{\mapsto} \mathsf{d}\} \\ \mathtt{get}_\tau(s) \\ \{s_\tau \overset{\frac{1}{2}}{\mapsto} ret * (R_s^{\mathsf{Thr}}(\tau, ret, ret) \mathrel{-\!\!*} R_s^{\mathsf{Thr}}(\tau, ret, \mathsf{d}))\}\end{array}} \text{[RATM]}$$

$$\frac{\forall\ v, \vec{v_t} \in \mathsf{Val}.\ v_\tau = x \wedge \mathsf{fsbl}(v, \vec{v_t}_{\{v_\tau = x\}}) \implies \mathsf{fsbl}(n, \vec{v_t}_{\{v_\tau = n\}})}{I_s \vdash \begin{array}{c}\{s_\tau \overset{\frac{1}{2}}{\mapsto} x * R_s^{\mathsf{Thr}}(\tau, n, n)\} \\ \mathtt{cas}_\tau(s, x, n) \\ \{(ret = true \wedge s_\tau \overset{\frac{1}{2}}{\mapsto} n * R_s^{\mathsf{Thr}}(\tau, x, x)) \vee \\ (ret = false \wedge s_\tau \overset{\frac{1}{2}}{\mapsto} x * R_s^{\mathsf{Thr}}(\tau, n, n)\}\end{array}} \text{[CATM]}$$

Figure 5.1: Contracts derived for `set`, `get` and `cas`

Specification: Atomic Write

Operation $\mathtt{set}_\tau(s, n)$ denotes the atomic update of s with n by a particular thread τ. We derive rule [WATM], expressing that the executing thread with the view d delivers all the resources associated with the feasible atomic state after the update. We should stress here that this contract is specific to using atomic write for synchronisation, it is not the most general contract possible.

For an atomic synchroniser for *exclusive* resource access, it is crucial that the value inferred by the protocol coincides with the thread's view. In

other word, the protocol embedded in the resource invariant must prove that the thread executing an atomic write has the *full permission* to do the `set` action, otherwise, it is not guaranteed that the thread intended to execute `set`, can indeed accomplish this safely.

Specification: Atomic read

The read action for a particular thread $\tau \in \mathsf{Thr}$ with a view s_τ that has the last visited value d from the atomic value s is indicated by $\mathtt{get}_\tau(s)$. In the rule [RAtm], the contract of the atomic read specifies that the atomic variable does not change its value, while the atomic state is modified because the reading thread updates its view. So the thread has to establish the resource invariant with the resources associated with the updated view inside the atomic body. As a result, it obtains the remainder as its post-condition, which is formalized using a magic wand operator. A magic wand (also known as resource implication) formula R_1 -* R_2 which holds for any heap that has the following property: if the heap is extended with a *disjoint* heap that satisfies R_1, then the combined heap satisfies R_2, and finally According to [76] this rule is correct if our resource assertions are *strictly exact*[2]. Syntactically any formula that only consists of points-to predicate and * operator is a strictly exact formula [88]. In a fragment of CSL that we use as our specification language (see Section 2.3), all resource formulas are indeed strictly exact.

Specification: Conditional update

Finally, rule [CAtm] specifies $\mathtt{cas}_\tau(s, x, n)$ with the expected value x and the value to be updated n. The calling thread assumes that the synchroniser contains a value equal to an expected value and then calls the operation to try to modify the atomic synchroniser to n. Therefore, the thread has to provide the resources associated with the updated atomic state and it will gain the resources associated with the expected value, if the operation succeeds. Otherwise, the operation returns all the provided resources.

[2] A formula in Separation Logic is strictly exact if it satisfies a *unique* heap [76].

Derivation of the Specifications

In order to derive the specifications, for each basic atomic operation we propose an implementation using basic instructions. We instantiate the [Atomic] rule for each operation with a pre-condition about the thread's view and thread's local state, containing the required resources. Then, we derive the post-condition from the pre-condition and the body, taking into account that I_s is available inside the body, providing the resources associated to the current state of the synchroniser. Inside the body, either the atomic location or the view of the thread is updated. Reasoning of the body is straightforward as it uses sequential SL rules. Folding the resource invariant at the end of the body, we can derive the predicates that the operation ensures. The derivations show that the thread consumes the resources it currently holds to re-establish I_s and exits the atomic body with an updated atomic state and the resources it obtains as the result of the update. The derivation of the contracts presented for the basic atomic operations in Section 5.2 are shown in Figure 5.2, Figure 5.3 and Figure 5.4.

As shown in Figure 5.2, the thread executing `set` obtains I_s inside the atomic block and using the feasible atomic state encoded inside I_s, it proves that after its last visit, the environment has not updated the atomic value. As we stressed in Section 5.2 it is crucial that the executing thread and the atomic value both have an identical value, otherwise the execution of `set` is not safe.

The implementation of the atomic read (`get`) (see Figure 5.3) updates the thread's view and stores the atomic value to a local variable named *ret*.

Similarly, the thread calling the atomic conditional update (see Figure 5.4) assumes that the atomic value equals to an expected value x and then calls the operation, trying to modify the atomic synchroniser into n. The executing thread reads the atomic synchroniser and compares it with the expected value. If it is equal to the expected value then the thread updates both the synchroniser and its view with n and re-establishes the resource invariant with the resources associated with n provided in the pre-condition. Otherwise, the thread establishes I_s without changing its view. The result of the operation is stored in the variable *ret*.

Let $R_s^{\mathsf{Thr}}(\tau, x, y) = \underset{\vec{v}_{t\{v_\tau = y\}} \in \mathsf{Val}}{\circledast} \mathsf{R}(s, x, \vec{s_t}, \vec{v}_{t\{v_\tau = y\}})$

and $I_s = (\exists\, v, \vec{w_t} \in \mathsf{Val}.\ s \overset{1}{\mapsto} v * (\underset{t \in \mathsf{Thr}}{\circledast} s_t \overset{\frac{1}{2}}{\mapsto} w_t) * \mathsf{R}(s, v, \vec{s_t}, \vec{w_t})) * \mathsf{P}_s^{\mathsf{Thr}}$

1: $\{s_\tau \overset{\frac{1}{2}}{\mapsto} \mathsf{d} * R_s^{\mathsf{Thr}}(\tau, n, n)\}$
$\mathtt{set}_\tau(s, n) \triangleq \langle$

2: $\{s_\tau \overset{\frac{1}{2}}{\mapsto} \mathsf{d} * R_s^{\mathsf{Thr}}(\tau, n, n) *$
$\left(\exists\, \vec{v_t} \in \mathsf{Val}.\ s \overset{1}{\mapsto} - * (\underset{t \in \mathsf{Thr}}{\circledast} s_t \overset{\frac{1}{2}}{\mapsto} -) * \mathsf{R}(s, [s], \vec{s_t}, \vec{v}_{t\{v_\tau = \mathsf{d}\}})\right)$
$* \mathsf{P}_s^{\mathsf{Thr}}\ \}$

3: $\{R_s^{\mathsf{Thr}}(\tau, n, n) * s_\tau \overset{1}{\mapsto} \mathsf{d} * s \overset{1}{\mapsto} \mathsf{d} * (\underset{t \in \mathsf{Thr} \setminus \{\tau\}}{\circledast} s_t \overset{\frac{1}{2}}{\mapsto} -) * R_s^{\mathsf{Thr}}(\tau, \mathsf{d}, \mathsf{d})\}$

4: `[s]:= n;` `[` s_τ `]:= n;`

5: $\{R_s^{\mathsf{Thr}}(\tau, n, n) * s_\tau \overset{1}{\mapsto} n * s \overset{1}{\mapsto} n * (\underset{t \in \mathsf{Thr} \setminus \{\tau\}}{\circledast} s_t \overset{\frac{1}{2}}{\mapsto} -) * R_s^{\mathsf{Thr}}(\tau, \mathsf{d}, \mathsf{d})\}$

6: $\{s_\tau \overset{\frac{1}{2}}{\mapsto} n * R_s^{\mathsf{Thr}}(\tau, \mathsf{d}, \mathsf{d}) *$
$\left(\exists\, \vec{v_t} \in \mathsf{Val}.\ s \overset{1}{\mapsto} n * (\underset{t \in \mathsf{Thr}}{\circledast} s_t \overset{\frac{1}{2}}{\mapsto} -) * \mathsf{R}(s, [s], \vec{s_t}, \vec{v}_{t\{v_\tau = n\}})\right)$
$* \mathsf{P}_s^{\mathsf{Thr}}\ \}$

7: $\{s_\tau \overset{\frac{1}{2}}{\mapsto} n * R_s^{\mathsf{Thr}}(\tau, \mathsf{d}, \mathsf{d}) * I_s\}\quad \rangle$

8: $\{s_\tau \overset{\frac{1}{2}}{\mapsto} n * R_s^{\mathsf{Thr}}(\tau, \mathsf{d}, \mathsf{d})\}$

Figure 5.2: Derivation of the specification for `set`

Thread-Modular Contracts

The last step is to adapt the derived contracts for the atomic operations to a thread-modular specification. In particular, this means that the specifications should express the pre- and post-conditions using local information only, *i.e.*, using:

1. the atomic value as a globally known state, and

Let $R_s^{\mathsf{Thr}}(\tau, x, y) = \underset{\vec{v_t}_{\{v_\tau = y\}} \in \mathsf{Val}}{\circledast} \mathsf{R}(s, x, \vec{s_t}, \vec{v_t}_{\{v_\tau=y\}})$

and $I_s = (\exists\, v, \vec{w_t} \in \mathsf{Val}.\ s \overset{1}{\mapsto} v * (\underset{t \in \mathsf{Thr}}{\circledast} s_t \overset{\frac{1}{2}}{\mapsto} w_t) * \mathsf{R}(s, v, \vec{s_t}, \vec{w_t})) * \mathsf{P}_s^{\mathsf{Thr}}$

1: $\{s_\tau \overset{\frac{1}{2}}{\mapsto} \mathsf{d}\}$
$\mathtt{get}_\tau(s) \triangleq \langle$

2: $\{s_\tau \overset{\frac{1}{2}}{\mapsto} \mathsf{d} *$
$\left(\exists\, \vec{v_t} \in \mathsf{Val}.\ s \overset{1}{\mapsto} - * (\underset{t \in \mathsf{Thr}}{\circledast} s_t \overset{\frac{1}{2}}{\mapsto} -) * \mathsf{R}(s, [s], \vec{s_t}, \vec{v_t}_{\{v_\tau = \mathsf{d}\}})\right) * \mathsf{P}_s^{\mathsf{Thr}}\}$

3: $\{s_\tau \overset{1}{\mapsto} \mathsf{d} * s \overset{1}{\mapsto} - * (\underset{t \in \mathsf{Thr} \setminus \{\tau\}}{\circledast} s_t \overset{\frac{1}{2}}{\mapsto} -) * R_s^{\mathsf{Thr}}(\tau, [s], \mathsf{d})\}$

4: s_τ`:= [s];` `ret:= [s];`

5: $\{s_\tau \overset{1}{\mapsto} ret * s \overset{1}{\mapsto} ret * (\underset{t \in \mathsf{Thr} \setminus \{\tau\}}{\circledast} s_t \overset{\frac{1}{2}}{\mapsto} -) * R_s^{\mathsf{Thr}}(\tau, ret, \mathsf{d})\}$

6: $\{s_\tau \overset{1}{\mapsto} ret * s \overset{1}{\mapsto} ret * (\underset{t \in \mathsf{Thr} \setminus \{\tau\}}{\circledast} s_t \overset{\frac{1}{2}}{\mapsto} -) *$
$R_s^{\mathsf{Thr}}(\tau, ret, ret) * (R_s^{\mathsf{Thr}}(\tau, ret, ret) \mathrel{-\!\!*} R_s^{\mathsf{Thr}}(\tau, ret, \mathsf{d}))\}$

7: $\{s_\tau \overset{\frac{1}{2}}{\mapsto} ret * I_s * (R_s^{\mathsf{Thr}}(\tau, ret, ret) \mathrel{-\!\!*} R_s^{\mathsf{Thr}}(\tau, ret, \mathsf{d}))\} \quad \rangle$

8: $\{s_\tau \overset{\frac{1}{2}}{\mapsto} ret * (R_s^{\mathsf{Thr}}(\tau, ret, ret) \mathrel{-\!\!*} R_s^{\mathsf{Thr}}(\tau, ret, \mathsf{d}))\}$

Figure 5.3: Derivation of the specification for `get`

2. local information that contains the view of the executing thread.

Note that the resource invariant expresses when the *synchroniser* holds the resources. For example, the resource invariant of ProducerConsumer does *not* specify when a particular thread can obtain the buffer. Generally, in cooperative patterns, the synchroniser holds the resource *temporarily*, until one of the waiting threads updates its view. We take advantage of this to simplify the contracts by defining the resources using two components:

1. the resources that the *synchroniser* holds for the competition, which is used to associate resources to the atomic values in classical definition

Let $R_s^{\mathsf{Thr}}(\tau, x, y) = \circledast_{\vec{v_t}_{\{v_\tau = y\}} \in \mathsf{Val}} \mathsf{R}(s, x, \vec{s_t}, \vec{v_t}_{\{v_\tau = y\}})$

and $I_s = (\exists\, v, \vec{w_t} \in \mathsf{Val}.\ s \overset{1}{\mapsto} v * (\circledast_{t \in \mathsf{Thr}} s_t \overset{\frac{1}{2}}{\mapsto} w_t) * \mathsf{R}(s, v, \vec{s_t}, \vec{w_t})) * \mathsf{P}_s^{\mathsf{Thr}}$

1: $\{s_\tau \overset{\frac{1}{2}}{\mapsto} x * R_s^{\mathsf{Thr}}(\tau, n, n)\}$
$\mathsf{cas}_\tau(s, x, n) \triangleq \langle$
2: $\{s_\tau \overset{\frac{1}{2}}{\mapsto} x * R_s^{\mathsf{Thr}}(\tau, n, n) *$
$\left(\exists\, \vec{v_t} \in \mathsf{Val}.\ s \overset{1}{\mapsto} - * (\circledast_{t \in \mathsf{Thr}} s_t \overset{\frac{1}{2}}{\mapsto} -) * \mathsf{R}(s, [s], \vec{s_t}, \vec{v_t}_{\{v_\tau = x\}})\right) * \mathsf{P}_s^{\mathsf{Thr}}\}$
3: **if**$([s] = x\ )$
4: $\{R_s^{\mathsf{Thr}}(\tau, n, n) * s_\tau \overset{1}{\mapsto} x * s \overset{1}{\mapsto} x * (\circledast_{t \in \mathsf{Thr} \setminus \{\tau\}} s_t \overset{\frac{1}{2}}{\mapsto} -) * R_s^{\mathsf{Thr}}(\tau, x, x)\}$
5: `{ [s]:= n;` s_τ `:= n; ret:= true; }`
6: $\{ret = true \wedge (R_s^{\mathsf{Thr}}(\tau, n, n) *$
$s_\tau \overset{1}{\mapsto} n * s \overset{1}{\mapsto} n * (\circledast_{t \in \mathsf{Thr} \setminus \{\tau\}} s_t \overset{\frac{1}{2}}{\mapsto} -) * R_s^{\mathsf{Thr}}(\tau, x, x))\}$
7: $\{ret = true \wedge (I_s * s_\tau \overset{\frac{1}{2}}{\mapsto} n * R_s^{\mathsf{Thr}}(\tau, x, x))\}$
else
8: `ret:= false;`
9: $\{ret = false \wedge (I_s * s_\tau \overset{\frac{1}{2}}{\mapsto} x * R_s^{\mathsf{Thr}}(\tau, n, n))\}$
$\rangle$
10: $\{(ret = true \wedge s_\tau \overset{\frac{1}{2}}{\mapsto} n * R_s^{\mathsf{Thr}}(\tau, x, x)) \vee$
$(ret = false \wedge s_\tau \overset{\frac{1}{2}}{\mapsto} x * R_s^{\mathsf{Thr}}(\tau, n, n))\}$

Figure 5.4: Derivation of the specification for `cas`

of the resource invariant, *i.e.* S, and

2. the resources that *threads* obtain when they are updating their views, denoted with T. Basically, $\mathsf{T}(s_\tau, v)$ indicates resources to be held by thread τ when $s_\tau = v$.

$$\frac{\forall\, v, \vec{v_t} \in \mathsf{Val}.\ v_\tau = \mathsf{d} \wedge \mathsf{fsbl}(v, \vec{v}_{t\{v_\tau = \mathsf{d}\}}) \implies \mathsf{fsbl}(n, \vec{v}_{t\{v_\tau = n\}})}{I_s \vdash \begin{array}{c}\{s_\tau \overset{\frac{1}{2}}{\mapsto} \mathsf{d} * \mathsf{S}(s, n) * \mathsf{T}(s_\tau, \mathsf{d})\} \\ \mathtt{set}_\tau(s, n) \\ \{s_\tau \overset{\frac{1}{2}}{\mapsto} n * \mathsf{S}(s, \mathsf{d}) * \mathsf{T}(s_\tau, n)\}\end{array}}\ [\text{WATM}]$$

$$\frac{}{I_s \vdash \begin{array}{c}\{s_\tau \overset{\frac{1}{2}}{\mapsto} \mathsf{d} * \mathsf{T}(s_\tau, \mathsf{d})\} \\ \mathtt{get}_\tau(s) \\ \{s_\tau \overset{\frac{1}{2}}{\mapsto} ret * \mathsf{T}(s_\tau, ret)\}\end{array}}\ [\text{RATM}]$$

$$\frac{\forall\, v, \vec{v_t} \in \mathsf{Val}.\ v_\tau = x \wedge \mathsf{fsbl}(v, \vec{v}_{t\{v_\tau = x\}}) \implies \mathsf{fsbl}(n, \vec{v}_{t\{v_\tau = n\}})}{I_s \ \vdash \begin{array}{c}\{s_\tau \overset{\frac{1}{2}}{\mapsto} x * \mathsf{S}(s, n) * \mathsf{T}(s_\tau, x)\} \\ \mathtt{cas}_\tau(s, x, n) \\ \{(ret = true \wedge s_\tau \overset{\frac{1}{2}}{\mapsto} n * \mathsf{S}(s, x) * \mathsf{T}(s_\tau, n)) \vee \\ (ret = false \wedge s_\tau \overset{\frac{1}{2}}{\mapsto} x * \mathsf{S}(s, n) * \mathsf{T}(s_\tau, x)\}\end{array}}\ [\text{CATM}]$$

Figure 5.5: Thread-modular specifications of atomic operations

We exploit these two components to decompose $R_s^{\mathsf{Thr}}(\tau, x, y)$ (defined in Figure 5.1) into a global and a thread local components.

These resources are either associated to the atomic value x, which will be obtained competitively using a `cas` operation, or associated to a particular view of a thread, which will be obtained by updating the view. We can formally express this decomposition for $\tau \in \mathsf{Thr}$, $x, y \in \mathsf{Val}$ as:

$$R_s^{\mathsf{Thr}}(\tau, x, y) \Leftrightarrow \mathsf{S}(s, x) * \underset{t \in \mathsf{Thr}, v_t \in \mathsf{Val}}{\circledast} (\mathsf{T}(s_t, v_t) \mathrel{-\!\!*} \mathsf{T}(s_t, x))$$

where $\mathsf{T}(s_t, v_t) \mathrel{-\!\!*} \mathsf{T}(s_t, x)$ specifies the resources that thread t exchanges when it updates its view from v_t to x.

In summary, for a competitive pattern, resources are merely associated with the state of the synchroniser using $\mathsf{S}(s, x)$. A *cooperative pattern* exploits the definition of $\mathsf{T}(s_t, v_t)$, which associates the resources to the view

of a thread expressing when the *thread* holds a resource. A *hybrid pattern* uses both $\mathsf{T}(s_t, v_t)$ and $\mathsf{S}(s, x)$ to reason about the resource exchanges.

We use this decomposition and update the contracts based on the fact that the executing thread may have resources obtained based on its *current* view. This results in thread-modular specifications for the basic atomic operations, as shown in Figure 5.5. which generally express that the executing thread must *provide*

1. the resources associated with its current view, and

2. the resources associated with the new state of the synchroniser.

In return the thread *obtains*

1. the resources associated with its updated view, and

2. the resources associated with the previous state of the synchroniser.

Note that in the patterns that we studied, the `cas` and `set` operations do not exchange resources using the thread views, and we are not aware of algorithms where these operations can transfer ownership based on their views.

5.4 Contracts of AtomicInteger

Based on the specifications derived above, we specify the behavior of the AtomicInteger class as an *exclusive-access atomic synchronisation primitive.* First, we explain all predicates and functions that we use in our specification of AtomicInteger, and then, we present the complete specification.

Predicates and Parameters

Any client program instantiating the AtomicInteger class as an exclusive atomic synchronisation primitive has to provide the *protocol* of the synchroniser object. In fact, a protocol of a synchronisation construct is an abstract state machine instrumented with an interpretation function that maps each state of the state machine to a fraction of the resources that the synchroniser object or a particular thread must hold in that state. Especially, in our settings, a protocol of a synchronisation construct must specify:

1. identification of the participants,
2. the shared resource that has to be protected by the synchronisation construct,
3. the fraction of the shared resource to be held by the synchroniser or a thread in each atomic state, and
4. the transitions that are valid for the synchroniser object.

To make a single specification of AtomicInteger that can capture all exclusive access patterns, the specification is parameterized by

1. a set of roles, which basically is an abstraction of the identification of the participating threads,
2. an abstract predicate as a resource invariant, specifying the shared resources to be protected by AtomicInteger,
3. a function to associate the states of the atomic integer as the synchroniser with the fraction of the shared resource,
4. a boolean predicate, encoding all the valid transitions that a particular instance of AtomicInteger can take, and
5. a handle token.

To make a single specification of AtomicInteger that can capture all exclusive access patterns, the specification is parameterized by

1. a set of roles,
2. valid transitions,
3. a resource invariant,
4. a function to specify the fraction of the shared resource, and

A *role* abstraction abstracts the identity of threads to a set of roles. This makes our specification unbounded in the number of threads. The synchroniser is defined as a globally known, special role, written S, that coordinates the threads. This role is declared as a publicly visible constant in class AtomicInteger, to hold the resource when the class runs the competition.

The validity of the transitions is encoded in the trans predicate. More importantly this encoding enables us to extract the set of the feasible states. The trans predicate expects as arguments the role of the invoking thread, the current and the intended update state of the synchroniser.

The shared resources are described by inv(**frac** p), a resource formula parameterized with permissions (of type `frac`), and defined as a `group`, *i.e.* it should be splittable over permissions. To associate the *fraction of the shared resources* with the state of the atomic integer, we define the function share, which is parameterized by a role, and the value of the atomic integer. Our role abstraction allows us to express S and T in the specification presented in Figure 5.5 using only inv parameterized with share.

For example, instantiating AtomicInteger for ProducerConsumer we define:

```
/*@ group inv(frac p) = Perm(data,p);
  pred trans(role r,int c,int n)=
    (r==P && c==E && n==F) || (r==C && c==F && n==E);
  frac share(role r,int s){
    return (r==P && s==E) ? 1: ((r==C && s==F) ? 1: 0); } @*/
```

where the definition of share shows that the full ownership of the shared resource, *i.e.* data, is only associated with the views of the threads. In the specification presented in Figure 5.5 this would mean that the S component would be emp and the T component associates the full ownership of data to the views of the threads. Similarly, instantiating AtomicInteger for SpinLock we use these definitions:

```
/*@ group inv = resinv;
 pred trans(role r,int c,int n) = (c==U && n==L)||(c==L && n==U);
 frac share(role r,int s){ return (r==S && s==U)?1:0; }; @*/
```

where resinv would be the shared resource to be protected by the lock which is passed as a class parameter to SpinLock. As it is specified in the definition of share the synchroniser, defined with the globally known role S, will hold the full resource when its state is U (unlocked). This can be expressed in the specification presented in Figure 5.5 with T defined as emp while the component S associates the full ownership of resinv to the unlocked state of the atomic location.

To invoke an operation from AtomicInteger, the calling thread must provide the correct required arguments which are demanded by the contracts. For this purpose, the AtomicInteger specification defines a special token, called handle, which can be used to prove that a thread has the

right to invoke an action. The post-condition ensures that appropriate new handles for new actions are handed out to the invoking thread. The handle is carrying the role of the calling thread which witnesses its role and its view from the state (last observed value) of AtomicInteger. Any instance of a synchronisation mechanism is associated with a particular set of threads. Therefore any thread (1) without a handle (*i.e.* outside of the coordinated threads), (2) with an incorrect role, or (3) with a visited value that is outside of the synchroniser's reachable states, will therefore not be able to interfere with the threads that participate in this synchronisation.

Handles are specified as group without a definition. At the initialization of the AtomicInteger, the constructor issues a full handle for all roles that are passed to the synchroniser. These full handles are all given back to the thread that created the AtomicInteger. These full handles may then be split and passed on to any other thread participating in the synchronisation.

Specification

Finally, Listing 31 shows the complete specification of class AtomicInteger. We briefly discuss the method specifications.

The constructor requires the fraction associated to the initial value of the atomic integer. These are the resources that are initially stored inside the synchroniser (S), and that can be won by the winning thread in a competition. Notice that in a cooperative synchronisation mechanism, the resources initially are supposed to be with one of the threads, and the synchroniser is only used as a medium to pass the resources on to the next thread. The post-condition of the constructor provides handles for all roles (except the S role) that are involved in the synchronisation, which can be split and passed to all threads that want to access the shared resource.

The contracts of the methods in AtomicInteger are all specified based on the specifications we derived in Figure 5.5 of Section 5.2. Given the role of the invoking thread, its last visited value from the state (view) and the fraction of handle, they all require handles carrying this information. New handles are returned as part of the post-conditions. State changing methods, *i.e.* set and compareAndSet, require that the intended transition is valid, as specified by the trans predicate. Finally, the fraction of the resource invariant to be exchanged is specified using inv and share based on the specifications derived for the basic atomic operations.

```
/*@
 given group (frac->group) inv;
 given (role,int->frac) share;
 given (role,int,int-> boolean) trans;
 given Set<role> rs;  @*/
class AtomicInteger {
  private volatile int value;
  /*@ group handle(role r,int d,frac p);  @*/
 /*@
  requires inv(share(S,v));
  ensures (\forall* role r; rs.has(r) ; handle(r,v,1)); @*/
  AtomicInteger(int v);
 /*@ given role r, int d, frac p;
  requires handle(r,d,p) ** inv(share(r,d));
  ensures handle(r,\result,p) ** inv(share(r,\result)); @*/
  public int get();
 /*@ given role r, int d, frac p;
  requires handle(r,d,p) ** trans(r,d,v);
  requires inv(share(S,v)) ** inv(share(r,d));
  ensures handle(r,v,p) ** inv(share(S,d)) ** inv(share(r,v)); @*/
  public void set(int v);
 /*@ given role r, frac p;
  requires handle(r,x,p)** trans(r,x,n);
  requires inv(share(S,n)) ** inv(share(r,x));
  ensures \result==> (handle(r,n,p)**inv(share(S,x))**inv(share(r,n)));
  ensures !\result==> (handle(r,x,p)**inv(share(S,n))**inv(share(r,x)));
  @*/
  boolean compareAndSet(int x, int n);
}
```

Listing 31: Contracts for AtomicInteger

Verification

In verifying client programs using AtomicInteger, it is vital to check the definition of share, as it should not allow the synchroniser to invent permissions. The distribution defined by share should satisfy the following property: *in all states, the total sum of the permissions held by the threads for a resource must not exceed the full permission.* To ensure that the definition of share fulfills the condition, we generate proof obligations stating that in any snapshot of the execution, the sum of the fractions assigned to all the threads and the synchroniser must not exceed 1. To show that this proof obligation is respected, we use the definitions of trans to extract the set of the valid states, and share to determine the resource distribution. The former draws the *maximal state machine* for each role, which shows *all possible transitions* that a role can take. The latter assigns the fraction that each role must hold in each state. Finally, the product of the maximal state machines is exploited to reason about the sum of the shares for each *feasible snapshot*.

This section presents how the contract for AtomicInteger is used to prove the correctness of the examples in Section 5.1. As explained, the function share together with predicate trans define a protocol by which the synchroniser controls the fractions of the resources that each thread must hold in each state. The correctness of the programs are accomplished in two steps: first we have to check if the definition of the protocol does not produce resources out of thin air. Then, having a correct definition of the protocol we can verify the correctness of the program. The verification of the programs are presented as proof outline where the specifications is annotated with the VerCors syntax and the intermediate states (shown inside the brackets) express the resources that the executing thread hold. For clarity, the intermediate states are only presenting the predicates that transform resources between the synchroniser and the participating threads. All the predicates that evaluated to **true** are omitted.

Verification of Case Studies

In the following we explain the correctness of the SingleCell in detail as it involves all the basic operations from AtomicInteger. Then, it should be easy to see how ProducerConsumer and SpinLock can be verified. Based on the given definitions for inv, share and trans, Listing 34 and Listing 35, annotated with in the VerCors syntax, show how the specification of AtomicInteger verifies

ProducerConsumer and SpinLock along with the sanity condition check for the given definition of share depicted in Figure 5.7 and Figure 5.8.

Protocol of SingleCell In order to check the sanity condition of the defined protocol, we start with introducing some notations. This technique is presented for each case study along with its proof-outlines in Figure 5.6, Figure 5.7 and Figure 5.8.

In our representations for verifying protocols, $\mathcal{A}_r$ denotes the state machine for the role r. The maximal model is represented as $\Pi_{\mathcal{A}}$. Nodes are annotated with r_v^π to denote that the role r assumes value v as its last observed state; and, based on function share, holds fraction π. Transitions are labelled w, denoting a *write* to the synchroniser, and r, indicating a *read* of state of the synchroniser. The transitions of the special role S (*i.e.* the synchroniser) are not labeled, since these transitions are due to received method calls. Finally, dashed transitions and nodes indicate impossible actions and states, respectively.

In SingleCell, the permissions are defined with three different values: *nothing*, *full* or *immutable*, see lines 3 - 6 of Listing 32. So to check the sanity condition of the share function in SingleCell (see Figure 5.8), it suffices to show the reachable states for only two arbitrary threads (t and t' are two threads with role T) and an instance of the synchroniser. For clarity, in $\Pi_{\mathcal{A}}$, the self-loops and unreachable states are not shown. Moreover, we just show the subset of states where thread t is the winning thread.

State machines $\mathcal{A}_\mathsf{T}$ and $\mathcal{A}_\mathsf{S}$ show the one way protocol of SingleCell. As can be seen, restricting w.r.t the one way protocol of the atomic integer prunes many traces in $\Pi_{\mathcal{A}}$ that are not feasible in the execution. Further, in $\Pi_{\mathcal{A}}$, the protocol allows t to update the state (n_0) from E to D, and to obtain the resource (see node n_1). As long as t holds the resource, t' (or any failing thread) can only update its view by reading the state: see transitions $n_1 \xrightarrow{\mathsf{r}_{t'}} n_2$, $n_3 \xrightarrow{\mathsf{r}_{t'}} n_5$ and $n_4 \xrightarrow{\mathsf{r}_{t'}} n_5$. The reading in $n_3 \rightarrow n_4$ is not valid since t' is taking an invalid step by this read operation. All other transitions labeled with w_t, interpreted as the write action taken by t, are valid transitions based on the contracts and definitions provided for the protocols. Finally, from the feasible maximal model, it is easy to conclude that the sum of shares never exceeds the full share.

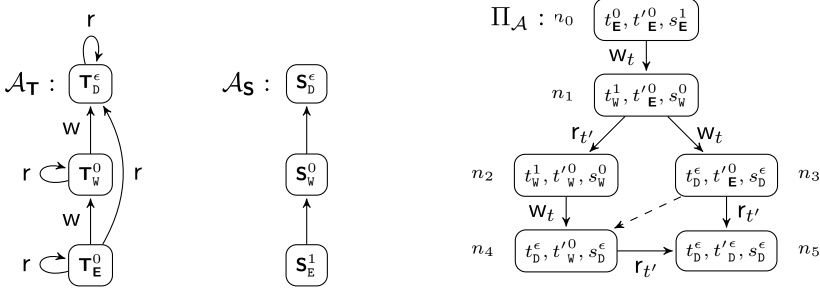

Figure 5.6: Verified SingleCell

Proof outline of SingleCell Having a definition of share that fulfils the sanity condition, we continue with the correctness proof of SingleCell. As shown, in the constructor the full permission on the shared resource data, is stored in the synchroniser. Then, the constructor ensures a full handle for role T initialized with E. The main program splits the handle, defined as a **group**, among the participating threads. So any thread calling the method findOrPut holds a fraction of the handle containing the view E.

The pre-condition of the compareAndSet operation in line 12 (see Listing 33), *i.e.* sync.compareAndSet(E,W), requires predicates inv(share(S,W)), inv(share(T,E)) and trans(T,E,W), which all are **true**. If the call is successful, the thread obtains the permission associated to the expected value, *i.e.* inv(share(S,E)), which can be opened to obtain the full permission to write data. Next sync.set(D) will be called for which inv(share(S,D)) is closed using the full permission. After the call, this thread only holds inv(+0), and knows that data == v. It will return with value PUT and the method's post-condition is established.

If the call to compareAndSet was not successful, the thread calls the sync.get() method, which only requires a handle on E. When the thread calling get updates its view to D, the thread obtains inv(+0), and thus it can read data. If data == v, the method returns SEEN and satisfies the post-condition. Otherwise, the method returns COLN and the post-condition trivially holds.

The verification of SingleCell class that we have explained so far, illustrates reasoning about the GSC synchronisation pattern (explained in Section 5.1), which employs all the methods and specification of AtomicInteger. The verification of ProducerConsumer shows an example of verifying the GS

```
public class SingleCell{
2  /*@
   group inv(frac p) = Perm(data,p);
4  pred trans(role r,int c,int n)=(c==E && n==W)||(c==W && n==D);
   frac  share(role r,int v){ return (r==S && v==E) ? 1:
6        ((r==S && v==D) ? +0: ((r==T && v==D) ? +0:0)); }   @*/
   /*@
8  requires Perm(data,1); ensures handle(T,E,1);  @*/
   SingleCell(){
10  /*! {inv(share(S,E))} !*/
    sync = new AtomicInteger(E);
12  /*! {handle(T,E,1)} !*/
   }
14 /*@  given frac f;
   requires handle(T,E,f);
16 ensures handle(T,D,f);
   ensures \result == PUT ==> PointsTo(data,+0,v);
18 ensures \result == SEEN ==> PointsTo(data,+0,v);  @*/
   int findOrPut(int v){  ...  }
20 }
```

Listing 32: Verification of SingelCell: constructor

pattern and the verification of SpinLock presents a reasoning of the SC pattern. Correctness proof of these two case studies, *i.e.* ProducerConsumer and SpinLock are shown in Listing 34 and Listing 35, respectively. Figure 5.7 represents the sanity condition for ProducerConsumer and Figure 5.8 depicts our check for the sanity condition of SpinLock. Based on our explanation for SingleCell it should be straightforward to follow the definitions and correctness proofs of the later case studies.

All the case studies discussed above are verified with our VerCors tool set available at [87]. This tool set has been developed to reason about multi-threaded Java programs annotated with permission-based SL. The tool leverages existing verification solutions to multi-threaded Java programs, by encoding verification problems into the Chalice language [58]. The Chalice verifier is then used to prove the translated program correct w.r.t. its specification. All case studies are verified automatically, after providing

```
public class SingleCell{
 ...

 /*@
  given frac f;
  requires handle(T,E,f);
  ensures handle(T,D,f);
  ensures \result == PUT ==> PointsTo(data,+0,v);
  ensures \result == SEEN ==> PointsTo(data,+0,v);  @*/
  int findOrPut(int v){
/*!{handle(T,E,f)**trans(T,E,W)**inv(share(T,E))**inv(share(S,W))}!*/
    if(sync.compareAndSet(E,W)){
/*! {handle(T,W,f)**inv(share(T,W))**inv(share(S,E))} !*/
/*! {handle(T,W,f)**Perm(data,1)} !*/
       data = v;
/*! {handle(T,W,f)**PointsTo(data,1,v)}  !*/
/*!{handle(T,W,f)**trans(T,W,D)**inv(share(S,D))
        **inv(share(T,W))}!*/
       sync.set(D);
/*! {handle(T,D,f)**inv(share(S,W))**inv(share(T,D))**(data==v)} !*/
/*! {handle(T,D,f) ** Perm(data,+0) ** (data==v)}  !*/
       return PUT;
     }
/*! {handle(T,E,f) ** inv(share(T,E)) ** inv(share(S,W)) }  !*/
     if(sync.get()!=E){
/*! {handle(T,val,f) ** inv(share(T,val)) ** (val!=E) }  !*/
       while(sync.get()==W);
/*! {handle(T,val,f) ** inv(share(T,val)) ** (val!=E) ** (val!=W)}  !*/
       if(sync.get() == D)
/*! {handle(T,D,f) ** inv(share(T,D))}  !*/
/*! {handle(T,D,f) ** Perm(data,+0)}  !*/
         if(data == v)  return SEEN;
         else  return COLN;
     }
  }
}
```

Listing 33: Verification of SingelCell::findOrPut()

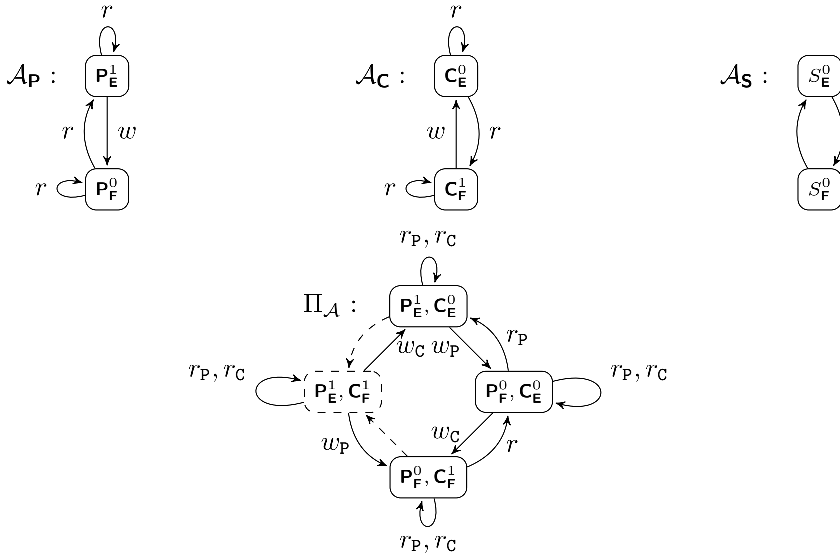

Figure 5.7: Verified ProducerConsumer

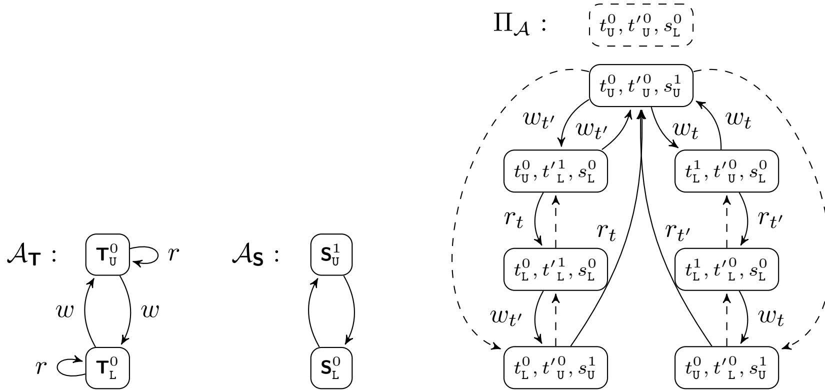

Figure 5.8: Verified SpinLock

a few proof hints in terms of intermediate state annotations that we left out here for clarity of presentation. In the presented proof outlines, for clarity, we only annotated the intermediate states of the proof with the predicates that transform resources between the synchroniser and the participating threads.

```
public class ProducerConsumer{
2   /*@
    group inv(frac p) = Perm(data,p);
4   pred trans(role r,int c,int n)=
     (r==P && c==E && n==F) || (r==C && c==F && n==E);
6   frac  share(role r,int s){
     return (r==P && s==E) ? 1: ((r==C && s==F) ? 1:0);
8   } @*/
  /*@
10  requires Perm(data,1);
    ensures Perm(data,1) ** handle(P,E,1) ** handle(C,E,1);  @*/
12  ProducerConsumer(){
  /*!  { Perm(data,1) }   !*/
14    sync = new AtomicInteger(E);
  /*!  { Perm(data,1) ** handle(P,E,1) ** handle(C,E,1) } !*/
16  }
   /*@
18  requires handle(P,E,1) ** Perm(data,1);
    ensures handle(P,E,1) ** Perm(data,1);  @*/
20  void produce(){
  /*!  {handle(P,E,1) ** inv(share(P,E))}  !*/
22    write(); // updates shared buffer
  /*!  {handle(P,E,1) ** inv(share(P,E))}  !*/
24    sync.set(F);
  /*!  {handle(P,F,1) ** inv(share(P,F))}  !*/
26    while(sync.get()==F);
  /*!  {handle(P,E,1) ** inv(share(P,E))} !*/
28  /*!  {handle(P,E,1) ** Perm(data,1)}  !*/
    }
30
   /*@  requires Perm(data,1); ensures Perm(data,1);  @*/
32  void write(){...}

34 /*@  requires true; ensures true;  @*/
    void consume(){ /* very similar to produce() */ }
36 }
```

Listing 34: Veirication of ProducerConsumer

```
/*@ given group (frac -> group) resinv;  @*/
public class SpinLock{
  /*@
  group inv = resinv;
  pred trans(role r,int c,int n) =
    (c==U && n==L) || (c==L && n==U);
  frac share(role r,int s){
    return (r == S && s == U) ? 1 : 0;  };  @*/

 /*@
  requires resinv(1);
  ensures handle(T,U,1); @*/
  SpinLock(){
/*!  {inv(share(S,U))} !*/
    sync = new AtomicInteger(U);
/*!  {handle(T,U,1)} !*/
  }

 /*@ given frac f;
  requires handle(T,U,f);
  ensures handle(T,L,f) ** resinv(1);  @*/
  void lock(){
     while(!sync.compareAndSet(U,L));
/*!  {handle(T,L,f) ** inv(share(S,U))} !*/
  }

 /*@ given frac f;
  requires handle(T,L,f) ** resinv(1);
  ensures handle(T,U,f);  @*/
  void unlock(){
/*! {handle(T,L,f) ** inv(share(S,U))} !*/
    sync.set(U);
/*! {handle(T,U,f) ** inv(share(S,L))} !*/
  }
}

```

Listing 35: Verification of SpinLock.

5.5 Conclusion and Related Work

This chapter proposes an approach to specify and reason about atomics as synchronisation constructs. Our approach separates the verification of

1. the correctness of the communication protocol, and
2. the code obeying the protocol, which carries out a rely-guarantee style proof in Separation Logic.

Moreover, the chapter discusses different patterns to synchronise a set of threads to access a shared resource using atomic read, write and compare-and-set. Based on these patterns, we provide a simple, thread-modular and practical specification of the class AtomicInteger from the atomic package of Java, using permission-based Separation Logic. The specification is easy and intuitive to be used, it only has to be instantiated by: the threads' roles; the shared resources that are protected by the synchroniser; a relation defining allowed state changes; a function that describes for each state change which share of the shared resource is transferred from the thread to the synchroniser, or vice versa; and the handle, as a witness for the provided information.

Using CSL, as a well-established logic, we derived the specification from the standard proof rule for atomic statements. To ensure overall soundness of the approach, it has to be ensured that the sharing function does not implicitly allow the creation of resources. We also briefly discussed how this can be verified.

Related Work Different program logics based on Separation Logic for concurrent programs can be found in the literature. Vafeiadis and Parkinson combined Rely-Guarantee reasoning and Separation Logic in RGSep to reason about fine-grained concurrent programs [86]. Assertions in RGSep distinguish between local and shared state, and actions are used to describe the interferences on the shared state between parallel processes. Later, Young *et al.* embedded permission-annotated actions in their assertion language and extended abstract predicates [71] to Concurrent Abstract Predicates (CAP) [33]. Abstract predicates in CAP encapsulate both resources and interferences, which allows one to reason about the client program without having to deal with all the underlying interferences and resources. The rule

for atomics in CAP uses a so called *repartitioning operator*, to extract the resources that the atomic operation requires or ensures.

In CAP it is not possible to reason about synchroniser objects that protect *external* shared resources. Inspired by Jacobs and Piessens [48], and Dodds *et al.* [35], CAP was extended by Svendsen and Birkedal resulting in Higher-Order CAP (HOCAP) [82] and later Impredicative CAP (iCAP) [81] to specify client usage protocols, suitable for synchronisers. iCAP is an important step towards reasoning about synchronisation mechanisms that protect client defined external states.

Ley-Wild and Nanevski [59] proposed Subjective CSL where the thread's *self* view and an *other* view (as a collective effect of the environment) are used to reason about coarse-grained concurrency. Finally, Hobor *et al.* [45] proposed a rule in CSL to reason about programs using barriers as their main synchronisation construct. But they didn't verify the implementation of the barrier.

All techniques mentioned above develop new program logics to reason about concurrent programs. Instead, here, we treat synchronisers at the specification level and we reuse existing verification technology to derive our practical and easy to use specifications from O'Hearn's classical CSL.

CHAPTER 6

Verification of Synchronisers: Shared Reading

In Chapter 5,using permission-based Separation Logic [23] we proposed an approach to specify atomic operations involved in exclusive access synchronisation. Then, using our specification for atomic operations we showed how to verify *exclusive access* synchronisation constructs, like Lock. The work we present here extends our approach for verification of exclusive resource protection [2] to cover *shared-readings*. The original approach from Chapter 5 defines a global synchroniser which has two main components: (1) the value of the atomic variable which is called an atomic state, and (2) the views of the participating threads which is the latest values that each thread remembers from the atomic state. The client program using the synchronisation class specifies a synchronisation protocol associated with the roles of the threads and the resource invariant for the global synchroniser. The protocol defines the expected behaviour of the synchroniser and the resource invariant associates the state of the global synchroniser to the shared memory.

In our study we found out that instead of the full permission, one can associate the state of the synchroniser to *fractions* of the shared memory. The result has an impact on the competitive-based synchronisations using compare-and-set which is the only synchronisation pattern in shared-reading synchronisers. In this chapter, the results of applying this idea is formalised as the specification of the atomic operations which is translated into the contract of AtomicInteger. Our new contract can be used to verify both *exclusive access* and *shared-reading* synchronisation classes. The applicability of the approach are demonstrated through verification of commonly used synchronisers: Semaphore, CountDownLatch and Lock. All the examples are mechanically verified using our VerCors tool-set [19].

The work we present here:

- is built based on our previous results on *specifying* (without verification) synchronisation primitives in Java presented in Chapter 4, and
- extends our approach for verification of *exclusive* resource protection using atomic operations (see Chapter 5) to cover *shared-readings*.

This chapter is structured as follows: Section 6.1 provides sample implementations of shared-reading synchronisers. Using permission-based Separation Logic, Section 6.2 provides the specifications for the three main atomic operations. Then, Section 6.3 specifies the contract for AtomicInteger as

the main element for the implementation of Java synchronisation classes. Later, Section 6.4 provides the verification of the synchronisers using the proposed specification of the AtomicInteger. Finally, Section 6.5 concludes the chapter along with the related work.

6.1 Synchronisation Classes

In this section, we provide simplified implementation of two different shared-reading synchronisation classes in Java: Semaphore and CountDownLatch. In our implementations, we removed fairness conditions of the original code, *i.e.* we didn't implement algorithms to fairly pick the next candidate for the shared resource competition.

A semaphore (see Listing 36) is a synchroniser where all the participating threads equally compete with each other to acquire or release protected portions of the shared resource. In a concurrent program synchronised with a semaphore, any thread trying to acquire a portion, has to win the competition by atomically decrementing the number of available portions (see line 12 of Listing 36). Similarly, as it is implemented in line 22 a releasing thread (again in a competition) must atomically increment the number of available portions.

Now assume an application with two distinct sets of active and passive threads, where active threads initially own a portion of the shared resource and passive threads wait for active threads to release their portions. As the implementation in in Listing 37 shows CountDownLatch performs as a synchronisation mechanism by which the passive threads are blocked until all active threads release their portion of the shared resource and transfer the ownership to the passive threads to proceed.

A CountDownLatch encapsulates a counter to denote the number of active threads working on the shared resource. Each active thread, once finished, calls countDown() on the latch, which decreases the counter (see line 11), to signal that it has delivered its portion. The passive threads wait for all the active threads by calling the blocking await() method on the latch. Inside this method, the passive threads are continuously reading the state of the latch until it reaches zero (line 21). In fact, the latch collectively accumulates the full shared resource and the waiting passive threads can continue their task only when they see that there is no more active thread possessing a portion of the shared resource.

```
public class Semaphore{
  private AtomicInteger sync;

  Semaphore(int n){ sync = new AtomicInteger(n); }

  public void acquire(){
    boolean stop = false; int c = 0;
    while(!stop) {
      c = sync.get();
      if( c > 0 ){
        int nextc = c−1;
        stop = sync.compareAndSet(c,nextc);
      }
    }
  }

  public void release(){
    boolean stop = false;
    while(!stop) {
      int c = sync.get();
      int nextc = c+1;
      stop = sync.compareAndSet(c,nextc);
    }
  }
}
```

Listing 36: Implementation of a Semaphore.

6.2 Reasoning about Atomics

In this section, we extend the formal specifications of the atomic operations presented in [2] in such a way that they can be used to verify both exclusive access and shared reading synchronisation constructs. To explain the essence of our specification, first, we focus on competitive resource acquisition using the cas operation. We start with a simple example that illustrates the behavior of atomic variables to see how the ownership of the resources is exchanged when an atomic variable is used as a shared reading synchronisation

```
public class CountDownLatch{
  private AtomicInteger sync;

  CountDownLatch(int count){
   sync=new AtomicInteger(count); }

  void countDown(){
    boolean stop = false;
    int c = 0 , nextc = 0;
    while(!stop){
      c = sync.get();
      if (c > 0){
        nextc = c−1;
        stop = sync.compareAndSet(c, nextc);
      }
    }
  }

  void await(){
    int c = sync.get();
    while(c!=0) {  c = sync.get(); }
  }
}
```

Listing 37: Implementation of a CountDownLatch.

mechanism.

Here we follow the formalisation from Chapter 5. In our new setting, we extend the interval of the permissions to include 0 and we define $e \overset{0}{\mapsto} - \equiv \mathsf{emp}$. As an example, using a semaphore $s \in \mathsf{ALoc}$ to protect a location $r \in \mathsf{NLoc}$, the value of the atomic location s (defined as atomic state) indicates the number of available fractions in the semaphore. The resource invariant for s associates the value of s with the maximum number of threads that concurrently can read r and is defined as:

$$I_s = \exists\, v \in \{0, \cdots, M\}.\ s \overset{1}{\mapsto} v * r \overset{\frac{v}{M}}{\mapsto} -$$

In an implementation of the semaphore, any thread that wishes to ac-

quire a portion of the shared resource must atomically decrement the value of s by 1. This transfers $\frac{1}{M}$ of r from s to the calling thread. This fraction is stored back to s by releasing the semaphore, which increments the current value of s atomically by 1. In fact, this behavior is justified by the atomic rule. In the implementations of acquire and release, the executing thread with an expected value x enters into the atomic body of the cas operation. Then, it obtains I_s inside the body, thus, has full access of s. Besides, if the current state equals x, it obtains $\frac{x}{M}$ fraction of r. So, the thread updates s with $n = x - 1$ for acquire or $n = x + 1$ for release, and re-establishes I_s with $r \overset{\frac{n}{M}}{\mapsto} -$ before leaving the body. To do so, the thread either acquires a $(\frac{x}{M} - \frac{n}{M})$ fraction of r or releases a $(\frac{n}{M} - \frac{x}{M})$ fraction of r. This provides the intuition to derive the specifications of the cas operation.

If we express the shared resources to be protected by the atomic location s using $\mathsf{res}(s)$, then we can define the resource invariant as:

$$I_s = \exists\, v \in \{0, \cdots, M\}.\ s \overset{1}{\mapsto} v * S(s, \pi) \quad \text{where} \quad S(s,\pi) = \underset{r \in \mathsf{res}(s)}{\circledast} r \overset{\pi}{\mapsto} -$$

Using I_s, the atomic location s can be the owner of the resources for which the participating threads through the cas operation have to compete to obtain or release their permissions. Based on this general definition of resource invariant, we can make our first attempt to specify the behavior of cas. For a shared-reading synchroniser s, if p maps the state of the synchroniser to the fractions with a maximum number of threads M, then we can axiomatise cas as follows:

$$\frac{\pi = p(s,x,M) \quad \pi' = p(s,n,M)}{\begin{array}{ll} & \{S(s, \pi' \dot{-} \pi)\} \\ I_s \vdash & \mathsf{cas}_\tau(s,x,n) \\ & \{(ret \implies S(s, \pi \dot{-} \pi')) * (\neg ret \implies S(s, \pi' \dot{-} \pi))\} \end{array}} \quad [\text{CAtm}]$$

where $\dot{-}$ denotes the cut-off subtraction over the fractions $\pi \in [0,1]$ with the following definition:

$$a \dot{-} b = \begin{cases} a - b & \text{iff } a \geq b, \\ 0 & \text{otherwise} \end{cases}$$

Surprisingly, the behaviors of both atomic read and write are more subtle than the cas operation. In some cases, the atomic read operation only updates the knowledge of the executing thread without transferring any

resource: see lines 9 and 20 in Listing 36, line 11 in Listing 37; whereas the waiting threads in CountDownLatch (see line 21 from Listing 37) obtain their fractions when they realize that the latch reached zero. On the other hand, unconditionally updating of the atomic write demands a *rely-guarantee* [50] style of reasoning as the writing thread must adhere to a *protocol* which guarantees the safety of the write to the environment. This is thoroughly discussed and formalised in Chapter 5. Here we extend the formal definition of the resource invariant from Chapter 5 to associate the state of the atomic variable with the fractions of the resources.

The first component is the global resource invariant that associates the resources to the *global* atomic state:

$$I_s = \exists v, \overrightarrow{w_t} \in \mathsf{Val} \cdot s \overset{1}{\mapsto} v * (\underset{t \in \mathsf{Thr}}{\circledast} s_t \overset{\frac{1}{2}}{\mapsto} w_t) * S(s, \pi) * \mathsf{P}_s^{\mathsf{Thr}}$$

where:

- having fsbl for determining the feasibility of the values taken by the atomic location and all the thread views $\mathsf{P}_s^{\mathsf{Thr}}$ is defined as follows:

$$\mathsf{P}_s^{\mathsf{Thr}} = \bigvee_{v, \overrightarrow{w_t} \in \mathsf{Val} \cdot \mathsf{fsbl}(v, \overrightarrow{w_t})} ([s] = v \wedge [\overrightarrow{s_t}] = \overrightarrow{w_t})$$

- the fraction of the resources is associated with the atomic state via $\pi = p(s, v, M)$, and
- finally, $S(s, \pi) = \underset{r \in \mathsf{res}(s)}{\circledast} r \overset{\pi}{\mapsto} -$

The second component associates the fractions of the resources to the thread views which can be exchanged through a collaborative synchronisation:

$$T(s_t, \pi) = \underset{r \in \mathsf{res}(s_t)}{\circledast} r \overset{\pi}{\mapsto} - \quad \text{where} \quad \pi = p(s_t, w_t, M)$$

By giving a definition for $T(s_t, \pi)$ one can express when a thread with a particular knowledge may obtain the resource. The `set` absorbs the resources either through I_s to the atomic location or through $T(s_t, \pi)$ to the reader thread. This is formally specified in the contracts for the basic atomic operations which are presented in Figure 6.1. The next section presents how the specification from Figure 6.1 translates into a contract for AtomicInteger, using our VerCors [87] specification language.

$$\frac{\begin{array}{l}\pi = p(s,n,M) \quad \pi' = p(s_\tau,\mathsf{d},M)\\ \forall\ v,\overrightarrow{v_t} \in \mathsf{Val}.\ v_\tau = \mathsf{d} \wedge \mathsf{fsbl}(v,\overrightarrow{v_t}_{\{v_\tau=\mathsf{d}\}}) \implies \mathsf{fsbl}(n,\overrightarrow{v_t}_{\{v_\tau=n\}})\end{array}}{I_s \vdash \quad \{s_\tau \overset{\frac{1}{2}}{\mapsto} \mathsf{d} * \mathsf{S}(s,\pi) * \mathsf{T}(s_\tau,\pi')\}\ \mathtt{set}_\tau(s,n)\ \{s_\tau \overset{\frac{1}{2}}{\mapsto} n\}}\ [\text{WAtm}]$$

$$\frac{\pi = p(s_\tau,M) \quad \pi' = p(s_\tau,M)}{I_s \vdash \quad \{s_\tau \overset{\frac{1}{2}}{\mapsto} \mathsf{d} * \mathsf{T}(s_\tau,\pi)\}\ \mathtt{get}_\tau(s)\ \{s_\tau \overset{\frac{1}{2}}{\mapsto} ret * \mathsf{T}(s_\tau,\pi')\}}\ [\text{RAtm}]$$

$$\frac{\begin{array}{l}\pi = p(s,x,M) \quad \pi' = p(s,n,M)\\ \forall\ v,\overrightarrow{v_t} \in \mathsf{Val}.\ v_\tau = x \wedge \mathsf{fsbl}(v,\overrightarrow{v_t}_{\{v_\tau=x\}}) \implies \mathsf{fsbl}(n,\overrightarrow{v_t}_{\{v_\tau=n\}})\end{array}}{\begin{array}{ll} I_s \ \vdash & \{s_\tau \overset{\frac{1}{2}}{\mapsto} x * \mathsf{S}(s,\pi' \dot{-} \pi)\}\\ & \mathtt{cas}_\tau(s,x,n)\\ & \{(ret \implies s_\tau \overset{\frac{1}{2}}{\mapsto} n * \mathsf{S}(s,\pi \dot{-} \pi')) \vee\\ & (\neg ret \implies s_\tau \overset{\frac{1}{2}}{\mapsto} x * \mathsf{S}(s,\pi' \dot{-} \pi)\}\end{array}}\ [\text{CAtm}]$$

Figure 6.1: Thread-modular specifications of atomic operations

6.3 Contract of AtomicInteger

The new contract of AtomicInteger is presented in Listing 38. The new specification is very similar to what we have already explained in Chapter 5. Therefore we only explain the contract for the operation compareAndSet where the permissions are exchanged using the cut-off subtraction.

The thread trying to atomically update the value of an atomic integer by compareAndSet(**int** x,**int** n), similar to other methods, has to have the permission for the transition from x to n and the right handle to call this operation. Following our formal specification (see Figure 6.1), our extension of the contract is that the compareAndSet(**int** x, **int** v) method absorbs the *difference* between the resources that the synchroniser will hold in case of a successful update, *i.e.* the resources associated with n, and the resources that the synchroniser object currently holds, *i.e.* the resources associated with x. If the operation succeeds, the operation ensures the difference between

```
/*@given Set<role> rs;
given group (frac->group) inv;
given (role,int->frac) share;
given (role,int,int-> boolean) trans;   @*/
class AtomicInteger {
  private volatile int value;
/*@group handle(role r,int d,frac p);   @*/

/*@requires inv(share(S,v));
ensures (\forall* r in rs: handle(r,v,1));  @*/
  AtomicInteger(int v);

/*@given role r, int d, frac p;
requires handle(r,d,p) ** inv(share(r,d));
ensures handle(r,\result,p) ** inv(share(r,\result));  @*/
  public int get();

/*@given role r, int d, frac p;
requires handle(r,d,p) ** trans(r,d,v);
requires inv(share(S,v)) ** inv(share(r,d));
ensures handle(r,v,p);@*/
  public void set(int v);

/*@given role r, int m, frac p;
requires handle(r,x,p) ** trans(r,x,n)
requires inv(share(S,n)-share(S,x));
ensures  \result==>
          (handle(r,n,p) ** inv(share(S,x) - share(S,n));
ensures !\result==>
          (handle(r,x,p) ** inv(share(S,n) - share(S,x));  @*/
  boolean compareAndSet(int x, int n);
}
```

Listing 38: Contracts for AtomicInteger: Exclusive and Shared

the resources that the synchroniser owned before the call, *i.e.* the resources associated on x, and the resources that holds after the successful update, *i.e.* the resources associated with n. If the operation fails, no resources are exchanged. Instead all resources specified in the pre-condition are returned. In the specification of AtomicInteger, the difference between the resources, turns into the subtraction operation between two fraction types. The subtraction between two permissions is defined as zero if the result of the operation becomes negative. Besides, as we explained before, inv(0) is equivalent to **true**. Therefore, as expressed in the contract of compareAndSet, the difference between the resources associated with the two states x and n determines if the calling thread releases or obtains fractions of the shared resource.

In the next section, we demonstrate how to employ our proposed specification of AtomicInteger to verify shared-reading synchronisation classes in Java.

6.4 Verification

In this section we demonstrate how one can verify the specification of a shared-reading synchroniser w.r.t. its implementation using an object of AtomicInteger.

We explain the verification of Semaphore which is a shared reading synchronisation class. Full tool-verified version of Semaphore is available online at [77]. The verification of CountDownLatch is very similar to Semaphore. The full verification of CountDownLatch is available online at [28]. In order to show that our new specification still supports exclusive access synchronisers, we provided (see [61]) a sample implementation of SpinLock which is verified with our extended specification. All these examples can be verified using online version of our VerCors verification tool accessible at [87].

Semaphore: verification

The Semaphore class implements a synchroniser where a group of threads with identical roles simultaneously have read access to a shared resource. Here we explain the verification of the Semaphore class, which is specified and verified in Listing 39, Listing 40 and Listing 41.

The Semaphore class is parametrized with the resource invariant defined by its client program. The instantiated semaphore uses two predicates as

tokens to detect if a thread holds a fraction of the shared resource: initialized and held.

An instance of a semaphore protects a shared resource with a specified maximum number of permits which is defined as a ghost variable within the class (line 3 of Listing 39). To instantiate an object of AtomicInteger, the Semaphore class has to define the required protocol. The resource acquirement using a semaphore is through a compare-and-set based competition. All the participating threads access the semaphore with an identical role. The shared resource to be protected by AtomicInteger is the same as the main program passes to the semaphore (Listing 39, line 7). The definition of the share function defines the fraction of the shared resource that must be held by AtomicInteger in each state (Listing 39, line 8). The definition given for the valid transitions expresses that in each update the difference between two states must be one unit (Listing 39, line 10).

Constructor

The client of the semaphore instantiates the object with a number of available units to acquire. Thus, it has to provide the resources associated with the initial value of the semaphore. After storing the maximum number of permits in the ghost field, the body of the constructor can feed the AtomicInteger class with the resources associated with its initial value (see line 20 of Listing 39). In return, the constructor of AtomicInteger returns its handle which can be used to establish the post-condition of the constructor of the semaphore as defined in line 5. Finally, the semaphore ensures a full initialized token to the client program which can be dispensed in portions among the participating threads.

Methods

The verification of the methods acquire() and release() are presented in Listing 40 and Listing 41, respectively. Having a fraction of the initialized token given by the client program, each thread is authorized to start its competition to acquire a permit of the shared resource protected by the semaphore. First, the acquiring thread has to read the current state of the atomic integer to see how many permits are still available. To achieve this, the body of the acquire has to unfold the provided initialized token to capture the required handle of the get method from AtomicInteger (line 5 of Listing 40). According to the provided protocol for the AtomicInteger,

```
/*@ given group (frac −> resource) rinv;  @*/
public class Semaphore{
 /*@ ghost final int num;
  ghost Set<role> roles = {T};
  group initialized(int d,frac p) = sync.handle(T,d,p);
  resource held(int d,frac p) = initialized(d,p);
  group inv(frac p) = rinv(p);
  frac share(role r, int c){
        return (r==S && c>=0 && c<num)?(c/num):0;}
  boolean trans(role r, int c, int n){
        return (r==T && c>0 && n==c−1)||
            (r==T && c<max && n==c+1); @*/
 private AtomicInteger/*@ <roles,inv,share,trans> @*/ sync;

/*@
 requires rinv(1) ** n>0;
 ensures initialized(n,1) ** num == n;   @*/
 Semaphore(int n){
/*@ set num = n;
      fold sync.inv(share(n));   @*/
  sync=new AtomicInteger/*@<roles,inv,share,trans>@*/(n);
/*@ fold initialized(n,1); @*/
 }

/*@ given int d, frac p;
requires initialized(d,p) ** d<=num ** d>0;
ensures held(?w,p) ** inv(1/num) ** w<num ** w>=0; @*/
  public void acquire(){ ... }

/*@ given int d, frac p;
requires held(d,p) ** inv(1/num) ** d<num ** d>=0;
ensures initialized(?w,p) ** w<=num ** w>0;  @*/
  public void release(){ ... }
}
```

Listing 39: Verification of Semaphore: constructor.

```
/*@ given int d, frac p;
2 requires initialized(d, p) ** d<=num ** d>0;
ensures held(?w,p) ** rinv(1/num) ** w<num ** w>=0;  @*/
4  public void acquire(){
/*@ unfold initialized(d,p);  @*/
6    boolean stop = false; int c = 0;
     while(!stop) {
8 /*@ fold sync.inv(sync.share(T,d)); @*/
       c = sem.get();
10     if( c > 0 ){
         int nextc = c−1;
12  /*@ fold sync.trans(T,c,nextc);
         fold sync.inv(sync.share(S,nextc)−sync.share(S,c)); @*/
14       stop = sem.compareAndSet(c,nextc);
       }
16   }
/*@ fold held(nextc,p); @*/
18 }
```

Listing 40: Verification of Semaphore::acquire().

the thread does not have any resource associated with its view. Therefore, having the right handle suffices to read the current state of the sync object (see line 9 of Listing 40). To acquire a unit of the available permits the thread must decrement the current state by one. So it folds all the required abstract predicates as the specification of compareAndSet demands. Based on the given definition for the protocol, the acquiring thread does not need to provide any resource at this step. In case of successful update, the compareAndSet(c,nextc) returns one unit of the shared resource, *i.e.* inv(1/num) (see 14 of Listing 40). The successful thread can leave the body of acquire after folding the held predicate using the available handle from AtomicInteger. In the post-condition of the acquire method, ?w denotes the existence of a view for the calling thread after the call. Finally, if the thread fails to decrement the current state, it has to continue with reading the current state and trying to atomically decrement the state.

Releasing a fraction of the shared resource is symmetric to the acquire method. It should be easy to follow the reasoning steps presented in List-

```
   /*@ given inr d,frac p;
 2 requires held(d,p) ** rinv(1/num) ** d<num ** d>=0;
   ensures initialized(?w,p) ** w<=num ** w>0 ;  @*/
 4  public void release(){
   /*@ unfold held(d,p); unfold initialized(d,p); @*/
 6    boolean stop = false;
      while(!stop) {
 8      int c = sync.get();
        int nextc = c+1;
10 /*@ fold sync.trans(T,c,nextc); @*/
   /*@ fold sync.inv(sync.share(S,nextc)-sync.share(S,c)); @*/
12      stop = sync.compareAndSet(c,nextc);
      }
14 /*@ fold initialized(nextc,p); @*/
    }
```

Listing 41: Verification of Semaphore::release().

ing 41. We only note here that the thread calling the release method provides the fraction of the shared resource it owns. Then, in an attempt to increment the current state of the atomic integer, if it succeeds gives up the permit by folding the inv abstract predicate required by compareAndSet(c,nextc) at line 12 of Listing 41.

6.5 Conclusion and Related Work

Many different extensions of CSL are proposed in the literature. RGSep [86] and Deny-Guarantee Reasoning [34] are the first logics that were used to reason about shared state by encoding the interferences on the shared state. Later, CAP [33] was introduced. In CAP, resources are encoded together with the environment interference in an atomic rule to reason about synchronisation with finer granularity. The dream of having an universal logic for concurrent programs resulted in developing various extensions of CAP, namely HOCAP [82], iCAP [81] and, finally, Iris [55]. Iris is a parametrized SL based logic to reason about fine-grained concurrency supporting resource algebras, invariants and higher-order predicates. The user has to instantiate the logic with the elements of the target programming language. Currently, Iris-based verification is performed in Coq. Finally, Caper [32] is a verific-

ation tool where the core logic is based on CAP with additional features taken mainly from iCAP and Iris.

All the above mentioned works focus on the development of a generic, universal and powerful program logics. Instead, we treat reasoning about atomic operations at the specification level using an already existing logic, *i.e.* permission-based Separation Logic. We modified our approach in such a way that the new specification of AtomicInteger can be used to verify both exclusive and shared-reading synchronisers. This is done by defining a function that associates the atomic state to the fractions of the shared resources. The definitions of protocols and resource invariant are updated accordingly. Then, by proposing the cut-off subtraction operation in permissions we updated the specifications of atomic operations. We presented a set of examples to demonstrate how our new specifications can be used to verified realistic implementations of synchronisers.

Compared to IRIS, our approach supports a more intuitive assertion language. In IRIS, invariants and and resource algebras that specify the behavioural protocols are a part of the underlying logic, while we reuse an existing specification language (JML) as much as possible. This allows one to employ currently existing permission-based Separation Logic verifiers like Silicon [66] and Verifast [49]. VerCors is the tool that encodes our specified programs to intermediate languages like Viper [66] and Chalice [58] to be verified by permission-based SL back-ends.

CHAPTER 7

Multi-layer Verification based on Concurrent Separation Logic

This chapter discusses how several concurrent program verification techniques can be combined in a layered approach, where each layer is especially suited to verify one aspect of concurrent programs, thus making verification of concurrent programs practical. The approach is supported with VerCors which combines different logic-based verification techniques. Each layer captures a different category of properties. In the lowest layer, we care only about data race freedom; in the middle layer we verify resource invariants that relate thread-local variables to globally shared variables, while in the top layer we verify arbitrary functional correctness properties. Strictly speaking, this separation in layers is not necessary, but it helps to keep the specification and verification tractable and to make mechanical verification feasible.

At the bottom layer, we use a combination of Implicit Dynamic Frames (IDF) [78] and CSL-style resource invariants, to reason about data race freedom of programs. We illustrate this on the verification of a lock-free queue implementation. On top of this, layer 2 enables reasoning about resource invariants that express a relationship between thread-local and shared variables. This is illustrated by the verification of a reentrant lock implementation, where thread-locality is used to specify for a thread which locks it holds, while there is a global notion of ownership, expressing for a lock by which thread it is held. Finally, the top layer adds a notion of histories to reason about functional properties. We illustrate how this is used to prove that the lock-free queue preserves the order of elements, without having to reverify the aspects related to data race freedom.

This chapter describes for each layer of the verification stack how the verification is handled in our VerCors tool set. The key idea behind the VerCors tool set is that it works as a transforming compiler, reducing complex verification problems into a verification problem for basic Separation Logic. As a back end, we use the Silicon verifier [54], thus all annotated programs are encoded into annotated Silver programs, which is the intermediate language used by Silicon, and has dedicated support to reason about access permissions. For each layer in the verification stack, we describe how the encoding into Silicon is defined. Because of our layered approach to verification, the transformation is easily manageable, and can be guaranteed to be correct.

We illustrate the usability of our layered approach by presenting a nontrivial verification example for each layer. At the lowest layer, we verify data

race freedom of a lock-free queue implementation, derived from the standard Java API lock-free queue. At the middle layer, we show how a relation between thread-local and shared variables is used to verify an implementation of reentrant locks, again derived from the Java API implementation, ensuring that two threads never can simultaneously hold the lock. At the top layer, we use histories to prove that the lock-free queue implementation preserves the order of elements stored in the queue.

It should be noted that this approach differs from recent proposals for a large range of powerful and expressive logics to reason about concurrent software, such as CAP [33], iCAP [81] and CaReSL [84], in that we do not aim at a highly expressive logic, but instead focus on an easily manageable approach to the verification of concurrent software, by breaking down the verification problem into smaller, more manageable problems. Moreover, the focus of our work is on efficient tool support, reusing currently available technologies, while the focus of these logics is expressiveness, and the ability to capture all concurrent programming patterns.

To summarize, in this chapter we will present:

- a layer-based approach to the verification of concurrent software that identifies different kinds of verification problems, which all need their own level of annotations;
- for each verification layer, a discussion how the verification problem is encoded into a simpler verification problem in basic Separation Logic; and
- verification of examples at each layer of the verification stack.

The chapter has been published in [3] and is structured as follows: Section 7.1 discusses how the combination of IDF and CSL-style resource invariants allows to verify data race freedom. Section 7.2 then discusses how the relation between global properties and thread-local state can be maintained as part of the resource invariant, while Section 7.3 discusses the verification of functional properties. Finally, Section 7.4 discusses conclusions and related work.

7.1 Layer 1: Permissions and Resource Invariants

IDF [78] is another program logic that extends Hoare Logic with the ability to reason about access to the heap by means of access permissions to heap locations, similar to permission-based Separation Logic. However, IDF and permission-based Separation Logic differ in how value-specifications are handled: in IDF, one uses side-effect-free expressions in the underlying programming language, while using permission-based Separation Logic, one first relates the program variables to logical variables and then states properties about these logical variables.

To encode the behavior of language constructs that are not part of IDF, we will use two of its proof commands. The exhale command first asserts that a formula is true and then drops the resources specified by the formula. The inhale command assumes the given formula and adds the resources specified by the formula.

At the bottom layer of the VerCors verification stack, a combination of IDF and CSL is used to reason about data race freedom. In this layer resource invariants capture access to the shared state. There are two ways to access shared state: by using locks (and other synchronizers), and by using atomics. Here we focus on atomic operations, and their encoding into Silicon, however reasoning about shared state protected by a lock is done in a similar way.

We illustrate our approach by discussing the verification of a lock-free queue, derived from ConcurrentLinkedQueue in the Java API. This example was also specified and verified by Jacobs *et al.* in richer logics [47, 79], but our version is a third shorter in length.

Reasoning about Atomic Blocks

In the VerCors tool, internally an atomic operation on object o with body S is modeled as atomic(o){S}. The resources associated to object o are specified by defining an appropriate resource invariant csl_invariant, which has to be established when the object is initialized, thus making it an implicit postcondition of all constructors of the class that defines the resource invariant.

To reason about atomic operations, CSL uses the rule [Atomic] [85]:

$$\frac{\{\mathsf{o.csl_invariant}() * P\}\ \mathsf{S}\ \{\mathsf{o.csl_invariant}() * Q\}}{\{P\}\ \mathsf{atomic(o)\{S\}}\ \{Q\}}\ [\text{Atomic}]$$

where csl_invariant is the resource invariant that specifies the shared state.

Encoding of Atomic Blocks

In Java, we do not directly write atomic statements. Instead, the atomic package provides several classes, whose methods perform the atomic operations get, set, and compareAndSet for all Java types. Thus, the encoding of atomic methods from Java into Silicon is done in two steps: first the atomic method call is transformed into a VerCors atomic instruction, and in the next step, the use of the proof rule [Atomic] is encoded into Silicon.

Java's atomic operations are encoded as VerCors atomic instructions, using several additional ghost variables to store the results of argument evaluation. For example, if var is an AtomicInteger then an atomic compare-and-set call, *i.e.* res=var.compareAndSet(expect,replace), is internally transformed into:

```
int obj=var; int x=expect; int v=replace;
atomic(this){
  if (obj.val==x) { obj.val=v; res=true; }
  else { res=false; }
}
```

Note that var, expect and replace are evaluated outside of the atomic block. The call to compareAndSet might be annotated with with{ G1 } then{ G2 } as ghost statements, to maintain the resource invariant, or to provide proof hints to prove correctness of the atomic body. In that case, these ghost instructions are evaluated inside the atomic block, after lines 2 and 4, respectively.

Finally, the encoding of the [Atomic] proof rule in Silicon is simple; each occurrence of the statement atomic(o){S} is replaced by the sequence:

```
inhale o.csl_invariant();
S;
exhale o.csl_invariant();
```

This transformation, first, uses the instruction inhale to add the resources and knowledge of the resource invariant. Then, using the added resources the body of the atomic block S is verified, and finally, exhale checks that the resource invariant holds and then removes it.

```
resource csl_invariant() = Value(begin) **
2    RPerm(head) ** ([read]reachable(begin,head.val)) **
     RPerm(tail) ** ([read]reachable(begin,tail.val)) **
4    Perm(last,1)   ** ([read]reachable(begin,last)) **
     chain(head.val,last) ** RPointsTo(last.next,null);
```

Listing 42: CSL resource invariant of the lock-free queue.

Verification of a Non-blocking Queue

We demonstrate the usability of our approach by verifying data race freedom of the essential methods of ConcurrentLinkedQueue from the util.concurrent package of Java, which implements a lock-free algorithm for queue as proposed by Michael and Scott [63]. First, we briefly explain the data structure, and then we describe how the class is specified and verified.

Implementation

The queue consists of (1) two atomic references: head and tail, and (2) a chain of nodes, where each node contains a value field and an atomic reference field to the next node. The head points to a *sentinel* node, *i.e.* its value does not contribute to the queue. The last node of the queue can be identified by its null-valued next field. A queue is empty when both the head and the tail point to the sentinel node with a **null**-valued next field.

Specification

The main part of the specification is the resource invariant, which characterizes a valid queue structure. The specification additionally uses two ghost fields with type Node: (1) begin, which represents the original head of the queue, *i.e.* the head of the queue when the data structure is initialized, and (2) last, which points to the last node of the queue.

The resource invariant uses several auxiliary predicates. First, RPerm and RPointsTo, which combine permission to read an AtomicNode with Perm and PointsTo on the embedded field, respectively. Next, the reachable(n,m) predicate captures that there is a path from n to m, and chain(n,m) specifies full ownership of the data element in the nodes of the queue located between n and m. The resource invariant (see Listing 42) states that begin can be read and is immutable. The fields head.val, tail.val, and last are writable and

reachable from begin. The elements between head and last are fully owned and the last.next is writable and **null**.

Verification

The essential part in the verification of the lock-free queue is proving that all atomic operations preserve the resource invariant. In addition to our encoding, this uses the following lemmas (which all have inductive proofs):

(i) The reachable predicate is transitive.

(ii) Given a node from which both the last node and some other nodes are reachable, either the other node is the last node, or the next node of the other node is also reachable.

(iii) Appending one node to a permission chain yields a permission chain.

Listing 43 shows a fully specified implementation of the method try_deq, which attempts to dequeue a node of the queue. First, it copies the current head of the queue to n1. Next, it copies the next of n1 to n2, which is allowed because due to lemma (ii) we have either write or read. If n2 is not **null** then the queue was not empty and compareAndSet is used to change the head to the next node. Upon success the element is returned as an Integer. In all other cases **null** is returned to signal failure. A full tool-verified specification of the queue is available at [27] which can be verified using the online version of the VerCors [87].

7.2 Layer 2: Relating Thread-Local and Global Variables

To understand why in layer 2 we add the notion of thread-local state, it is important to realize that the queue specification above does not allow threads to express any property about the elements in the queue, even though the specification of the queue describes the queue's behavior in terms of all elements the queue has held and still holds. This list, however, is only available for reasoning during atomic operations, because the access permissions on the list are maintained in the invariant. Hence, it is not possible for threads to reason about the elements of the queue outside of atomic regions, and worse, because a thread does not have any permission on these variables

```
/*@ requires Value(head) ** Value(tail);
2     ensures  Value(head) ** Value(tail)
      ** (\result != null ==> Perm(\result.val,1)); @*/
4  Integer try_deq(){
     Node n1,n2;  boolean tmp;  Integer res=null;
6    n1=head.get();
     n2=n1.next.get() /*@ with {
8      lemma_readable_or_last(this.begin,n1);
     } @*/;
10   if (n2!=null) {
       tmp=head.compareAndSet(n1,n2)/*@ with {
12       if (head.val==n1) {
           unfold chain(head.val,last);
14       }
       } @*/;
16     if(tmp){
         res=new Integer(n2.val);
18     }
     }
20   return res;
   }
```

Listing 43: Dequeue attempt.

outside of atomic regions, it is forced to forget all knowledge about them: after all, any other thread might modify them.

The simplest way to avoid this loss of information is to add thread-local state and to keep this thread-local state synchronised with the global state. The concept of thread-local state is old; it is already used in the classical example of Owicki-Gries [70] where two threads independently atomically increment a variable by one. To prove that the end result increases the initial value by two, two thread-local ghost variables are used that account for the behavior of each thread. These ghost variables are then used to state a resource invariant that precisely captures the value of the shared variable.

In our approach, this combination of thread-local and global state is established as follows. Full permission on the shared variable is kept in the invariant, thus it may be modified during atomic operations. For thread-

local state, half the permission is held by the invariant, thus it may be read to specify the relation with the shared state. In addition, each thread holds the other half permission on its own thread-local state, which means that it can read its thread-local state at any time during execution, and moreover, it has the ability to change its own thread-local state when it holds the resource invariant, *i.e.*, during atomic operations.

Encoding

The concept of a thread-local variable is present in many programming languages. Typically, thread-locality is not a primitive of the language, but it is added by means of a library. For the moment, we use a manual encoding of thread-locality on top of layer 1. Our encoding assumes that an application has access to a fixed number of unique objects (or integers) identifying each of the threads, which allows to encode thread-local variables as an array, where each array element corresponds to the thread-local variable for the corresponding array index.

Additionally, a special treatment is required for reasoning about the current thread id. We have a specification construct \current_thread, which yields the id of the current thread. Thus, as its semantics depends on the thread in which it is evaluated, \current_thread cannot be used in invariants. In fact, it may not be used in any predicate, unless the specification modifier thread_local is used for the predicate. Moreover, any predicate that invokes a thread_local predicate has to be marked thread_local itself.

We encode \current_thread by adding it as an argument to all methods, constructors, and thread_local predicates and all their invocations. This allows us to detect illegal use of \current_thread, *i.e.*, in a non-thread_local predicate, because every illegal use would result in a local variable that was not declared. Moreover, we check that csl_invariant is not declared thread_local.

Specification and Verification of a Reentrant Lock.

To illustrate the kind of verifications that can be done at layer 2, we discuss the specification and verification of the implementation of a reentrant lock. The specification is adapted from [10, 5], while the implementation is a simplified version of OpenJDK6 java.concurrent.locks.ReentrantLock.

```
interface Lock {
/*@ resource lock_invariant();
    given    bag<Lock> S;
    requires lockset(S);
    ensures  lockset(S+seq<int>{this});
    ensures  !(this \memberof S) ==> lock_invariant(); @*/
    void lock();
}
```

Listing 44: Interface Lock.

The major challenge in specification and verification is the *reentrant* lock behavior. In separation logic, the specification of the behavior of non-reentrant locks is simple: when obtaining the lock, the resource invariant attached to the lock, *i.e.* access to the shared data protected by the lock, is also obtained and upon unlocking the invariant must be released. Assuming that double locking leads to an unchecked error or a deadlock, this behavior can be specified with straight-forward contracts. However, for reentrant locks more care is required: the resource invariant is only obtained upon locking for the first time and it must be yielded when unlocking for the last time only. This means that when obtaining the lock, the invariant can only be obtained if there is a proof that the lock is obtained for the first time.

Implementation

We follow the ReentrantLock implementation in OpenJDK6 by having two fields: an atomic integer count and an integer owner. The latter variable is set to the thread id of the current owner of the lock, or -1 otherwise. If a thread already is the owner then a (re)lock is done by atomically increasing count. Otherwise, the lock must be obtained by changing count from 0 to 1 using compare-and-set. To release the lock, the count is decreased, where the owner must be cleared before the final decrease to 0.

Specification

The specification of a reentrant lock (see Listing 44) uses the predicate lockset(S), where S is a multi-set of locks. The predicate lockset(S) holds for a thread if the multiplicity of any lock in the lock set is the number of times

```
/*@
resource csl_invariant()=
  Value(T) ** T > 0 ** Value(held) ** Value(subject) **
  APerm(count,1/2) ** APerm(owner,1/2) **
  (count.val==0 ==> subject.inv() **
  APerm(count,1/2) ** APerm(owner,1/2)) **
  Perm(holder,1) ** -1<=holder<T **
  (holder==-1) == (count.val==0) **
  (\forall* int i; 0 <= i < T ;
    Perm(held[i],1/2) ** (i!=holder ==> held[i]==0)
    ** held[i] >= 0 ** (held[i]==0 ==> owner.val!=i)
  );
resource lockset_part()=
  Perm(held[\current_thread],1/2) **
  (held[\current_thread] > 0 ==>
    APointsTo(count,1/2,held[\current_thread]) **
    APointsTo(owner,1/2,\current_thread)); @*/
```

Listing 45: The definition of the lock invariant in the Lock class.

the thread holds that lock. Hence, obtaining a lock adds the lock to the lock set, while releasing a lock means removing it from the lock set. Moreover, when a lock does not occur in the lock set before locking, the resource invariant is obtained, and when it is no longer present after unlocking, it has to be yielded. In our previous work [10], the lockset(S) was added to the specification language as a primitive. In this chapter, we define it in terms of the current thread and an invariant.

Invariant

To define the invariant for a lock (see Listing 45), we use several ghost variables. Without loss of generality, we assume a fixed number of threads (T), which is also a ghost variable. We use a ghost array held with T entries that functions as the thread-local count for each thread. And we use a ghost variable holder that tracks the owner of the lock. Note that we cannot use the implementation field owner because it cannot change at the same time as the implementation field count changes. Ghost fields can change at the same time and thus preserve an invariant.

Permission for the various fields of a lock are divided between the invari-

```
/*@
thread_local resource lockset(bag<int> S)=
  Value(T) ** 0 <= \current_thread < T **
  Value(L) ** L > 0 ** Value(locks) **
  (\forall* int l ; 0 <= l < L ;
     Value(locks[l]) ** Value(locks[l].subject) **
     Value(locks[l].T) ** locks[l].T==T **
     Value(locks[l].held) ** locks[l].lockset_part() **
     locks[l].held[\current_thread]==(l \memberof S)
  ); @*/
```

Listing 46: The definition of the lockset predicate in the Thread class.

ant and the lockset_part predicate that will be used in the lockset definition. The invariant holds permission $\frac{1}{2}$ on each element of the held array and the count and owner atomic fields. The other $\frac{1}{2}$ for held elements is held in the corresponding lockset, while the other $\frac{1}{2}$ for the atomic fields is kept in the lockset for the owning thread and in the invariant if the lock is free.

The lockset predicate is defined in Listing 46. The most important lines are the last two: for every lock, the lockset holds the permissions defined in lockset_part and the held count for the current thread is precisely the multiplicity of the lock id in the lockset.

The full listing of the lockset specified implementation of our reentrant lock can be found online [87].

7.3 Layer 3: Functional Properties using Histories

Invariants, with or without thread-locals, are adequate for specifying and verifying data race freedom and basic functional properties. The verification of more complex functional properties, however, can get very tedious because all interactions between threads have to be specified in great detail. Therefore, this section discusses a different approach, adding the notion of histories [22] on top of layer 2, and uses this to prove that the order of elements is preserved in the lock-free queue.

Reasoning with Histories

The key idea of history-based reasoning is that functional verification is not performed on the program directly, but on an abstract model of the program. This idea combines data abstraction [44] with process algebra [16].

An abstract model is defined in terms of variables and actions on those variables. Each action abstracts away from a concrete operation that can be carried out on the program data: an action contract specifies which concrete operations the action corresponds to. For example, Listing 48 specifies an abstract model for queues: q specifies a queue state, while get is an abstract action on the queue. Actions can be combined into processes using standard operators, known from process algebra, such as choice, sequential composition, and parallel composition. To capture action repetition, the behavior of processes also can be described using a recursive definition, which must be paired with a contract. See for example the definition of process get_all in Listing 48 (lines 12-15).

In the method specifications, we record the local history of the actions performed in a thread. For example, the method specification of method get in Listing 49 expresses that this method performs an abstract get action. In the method bodies, annotations are added to mark blocks that implement actions. For example, in the method try_deq in Listing 43 we add the following statement:

```
/*@
  action hist, p , P, hist.get(n2.val);
  hist.q=tail(hist.q);  @*/
```

after the **unfold** at line 13, stating that we extend the history hist, for which we own a fraction p and which is P, with an action get. This is done by removing the head element from hist.q.

Typically, whenever a thread is created, it starts with an empty local history. When threads terminate and are joined, the local history of the terminated thread is merged with the history of the joining thread. Eventually, this results in one global history of abstract actions over which the desired functional property can be verified. To guide the verification, some additional annotations for the treatment of histories may be provided as proof hints: initialize a new, empty history over a set of program fields, destroy a history etc., see [22] for a full overview.

The verification of these annotations consists of two main tasks. First, the code must be verified to ensure that it implements a linearizable sequence

```
/*@
  requires Perm(q,1) ** PointsTo(q_mode,1,0);
  ensures Perm(q,1) ** PointsTo(q_mode,1,1) **
    Perm(q_init,1/2) ** q == \old(q) ** q_init == \old(q) **
    hist_passive(1,p_empty()); @*/
  void create_hist();
```

Listing 47: The method that encodes creating a history

of actions, as specified in the history annotations. Second, the history specification has to be verified to ensure that every possible trace satisfies the behavior specified in the form of the process contracts.

This has two advantages for the verification of functional properties. First, we can abstract from implementation details (e.g. the linked list in the queue becomes a sequence in the history). Second, because we have already verified data race freedom, we can verify the properties in a non-deterministic sequential setting, which makes it less complicated.

Encoding

In theory, histories are defined over arbitrary sets of locations. The input language for the tool however does not use locations as first class citizens, so it defines histories over the fields of a History class. Actions and processes are also defined using an appropriate ADT in the same class. The predicate Hist that describes (part of) the recorded history has three arguments: a reference to a history object (instead of a set of locations), a fraction and a process expression that denotes the history accounted for. The initial state of a history is specified using the predicate HistInit, whose arguments are a reference to a history and a formula. For example, starting with an empty queue is specified as `HistInit(hist,q==\seq<int>{})`.

To complete the first verification task *i.e.*, to ensure linearizability, modifications of the fields of the history object have to be grouped in action blocks, which must keep full permission on every field written during the action for the duration of the entire action. This is managed by having three forms of permissions on the fields: normal (Perm), passive (HPerm) and active (APerm). Passive permissions can only be used to read from history fields. To write a field you need full active permission. Permission changes, as described in [22], are encoded by replacing every history annotation by

a method call whose contract matches the behavior of the proof rules for the annotation. For example, Listing 47 shows the method that encodes the creation of a history. Note how the initial state is in an extra ghost field (q_init) in order to be able to reason about it. The hist_passive predicate encodes Hist(**this**,1,empty). Note how the empty expression is replaced by an expression in the ADT.

The verification of action blocks records the initial state of the variables to be modified in ghost fields (for checking the actions post-condition), exchanges full passive for full active permission, and assumes any pre-condition of the action. In addition, the encoding of the primitive Hist predicate is inhaled. At the end of the action block, it asserts the post-condition of the action and changes the permissions back. In addition, the encoding of the Hist predicate with an extra action appended is exhaled.

The second part of the verification is to show that every execution trace of a history satisfies its contract. To do this, we generate a method for every defined process, whose body is constructed as follows: sequential composition on processes becomes sequential composition of statements, choice on processes becomes a non-deterministic choice, and every mention of an action or a defined process becomes a method invocation. For example, the method generated for put_all is

```
  ensures q==\old(q)+es;
2 void put_all(seq<int> es){
    if (|es|==0){
4   } else {
      put(head(es));
6     put_all(tail(es));
    }
8 }
```

Note that we call put_all in the body. This is safe because this call is guarded by a call to the action method put, which means that by induction on the length of a trace we can assume that this call satisfies its contract. If a process expression contains unguarded calls, the laws of process algebra are used to compute an equivalent guarded form.

Verification of a Queue History

To illustrate reasoning about complex functional properties using histories, we prove a functional property for the lock-free queue discussed in layer 1:

```
   seq<int> q;
2
   modifies q;
4  ensures \old(q)==seq<int>{e}+q;
   process get(int e);
6
   modifies q;
8  ensures \old(q)==es+q;
   process get_all(seq<int> es)= |es| == 0?empty:
10                   (get(head(es))*get_all(tail(es)));

12 ensures get_all(es)*get(e)==get_all(es+seq<int>{e});
   void get_lemma(seq<int> es,int e){
14   if (|es|>0){ get_lemma(tail(es),e); }
   }
16
   modifies q;
18 ensures \old(q)+input==output+q;
   process feed(seq<int> input,seq<int> output)=
20                   put_all(input)||get_all(output);
```

Listing 48: Fragment of History Specification for Queues.

if one thread is given an array with elements that it puts into an empty queue, and a second thread is given an array of the same length that it fills by getting elements from the queue then, once both threads terminate, the contents of both arrays are identical. Essentially, this captures that the order of elements is preserved in the queue.

First, Listing 48 shows part of the history specification. The data managed by the history is a single field q that contains a sequence of integers, *i.e.*, an abstraction of the queue contents. Next, the contract of the get action shows that it removes the first element of q. Then, we define process get_all, which appends a whole sequence of integers to the queue. The history specification is extended with a lemma that shows a useful property about the get_all process, namely that the sequential composition of a get_all and a get is again a get_all. Finally, we define the feed process on whose contract the whole verification hinges; it states the behavior of putting and getting

```
/*@
2   History hist;
    boolean hist_active;
4   given frac p;  given process P;
    requires Value(hist) ** Hist(hist,p,P) **
6       p!=none ** PointsTo(hist_active,p/2,true);
    ensures Value(hist) ** Hist(hist,p,P*hist.get(\result)) **
8       p!=none ** PointsTo(hist_active,p/2,true);  @*/
public int get();
```

Listing 49: History specification of the get method.

two sequences in parallel: the old contents plus the new elements have to be the same as the retrieved elements plus the current state.

Listing 50 shows the run method of the receiver, annotated with a loop invariant that describes its behavior (the method contract itself is not shown as it is essentially an instance of the loop invariant). The body of the loop uses the get method specified in Listing 49 to fill the array. Note how the ghost field vals is used to maintain the list of elements written into the array. Also note that the loop invariant only refers to the output array and the vals field: the specification does not depend on the behavior of other threads that may operate on the queue. The comparison of the orders of the input and the output occurs when the two threads are joined by the thread that forked them. This thread knows that: (i) at first q was empty, (ii) the program in parallel put all input array elements into the queue and got all output array elements from the queue. (iii) these arrays have the same length. What (ii) says is that the program performed the process feed (Listing 48, line 18) for the contents of the two arrays. From the contract of that process, it can be inferred that those contents are the same and the current q is empty too. Thus, modular verification is achieved.

Finally, we revisit the lock free queue try_deq method in Listing 43. To add history support we do the following:

- We add a ghost field History hist; to keep the history state.
- We add a ghost variable **boolean** hist_active=**true**; that denotes if the history is active.

```
public void run(){
  int N=output.length;
  int i=0;
  /*@
   vals=seq<int>{};
   loop_invariant Value(queue) ** Value(queue.hist)
     ** Value(output) ** Perm(vals,1) ** 0 <= i <= N
     ** i==|vals| ** N==output.length
     ** PointsTo(queue.hist_active,1/4,true)
     ** (\forall* int k; 0 <= k < N ; Perm(output[k],1))
     ** (\forall int k; 0 <= k < i ; output[k]==vals[k])
     ** Hist(queue.hist,1/2,queue.hist.get_all(vals)); @*/
  while(i<N){
    output[i]=queue.get()
    /*@ with { p=1/2 ; P = queue.hist.get_all(vals); } @*/;
     /*@ vals=vals+seq<int>{output[i]}; @*/
     i=i+1;
  }
}
```

Listing 50: Specified run method of the receiver

- We modify the definition of chain to have a third argument that contains the contents of the chain, and we propagate this change to all uses of chain, including the lemmas.

- We change chain(head.ref,last) in the invariant to

  Perm(hist_active,1/2) ** Value(hist) **
  (hist_active ==> HPerm(hist.q,1)**chain(head.ref,last,hist.q))

  i.e., we can access list_active and hist, and while the history is active we hold protected permission for hist.q, whose value is precisely the contents of the queue.

- We insert an action block after the **unfold** at line 13 to keep the queue contents and hist.q equal, as discussed above:

  action hist, p , P, hist.get(n2.val); hist.q=tail(hist.q);

7.4 Conclusion and Related Work

In this chapter, we have shown how a layered combination of CSL-based verification approaches for concurrent programs can be used to make mechanical verification feasible and practical. Each layer focuses on a particular class of properties, and reuses the results of the lower layers. The layered approach enables a compositional encoding into a simple verification language, Silicon, with appropriate tool support. Because the encoding focuses on a simple aspect of the verification, it is easier to become convinced of the correctness of the encoding. We also illustrate how verification can take advantage of the layered approach. In particular, at layer 3, we verify a functional property of a lock-free queue, for which we have already shown data race freedom at the lower layer 1. Moreover, in layer 2, we verify correctness of a standard reentrant lock implementation with respect to its contracts mainly specifying its reentrancy properties [5].

As already mentioned above, various program logics have recently been proposed to reason about concurrent programs, *e.g.*, CAP [33], iCAP [81], CaReSL [84] , TaDA [29] and IRIS [55]. These are all highly expressive logics, which are able to reason about similar properties as discussed in this paper, but as far as we are aware, there is no tool support for them, while our focus is to make verification of concurrent programs practical. The difference between CAP and our approach is that we use a predicate as the resource invariant, rather than an arbitrary boxed formula. As a result, we can only use explicitly declared variables to specify a relation between the local state and the invariant. This is less elegant, but also prevents the problem of stability of boxed formulas that is caused by implicitly crossing the boundary between the shared and the local state. Essentially, iCAP enriches CAP with impredicative protocols and CaReSL introduces thread-locals to modularly reason about fine-grained data-structures. However, so far there is no tool support for these program logics. Moreover, these logics share the property that functional verification happens in parallel with the verification of the data race freedom. In contract, using histories the functional verification happens separately. The logic TaDA has one feature that our logic does not (yet) have: it allows proving that a method behaves as if it atomically performs an action, while we can only axiomatize this.

Closely related to our work, Jacobs and Piessens propose a technique to verify functional properties of lock-free data-structures involving atomic operations in VeriFast [47], which has recently been extended with support

for Rely-Guarantee reasoning [79]. They also study the Michael-Scott queue as an example and their work is implemented in VeriFast. As far as data race freedom is concerned, their work and ours are identical in concept, but quite different in the organization of the specification language. For example, we write glue code in with and then blocks for every atomic method invocation. They declare several ways in which an atomic method can operate up front. For the functional properties, their method does not achieve the complete separation between local and global reasoning that is enabled by histories. Also, their notion of action is less general than ours, as theirs is limited to a single atomic operation, while ours can combine multiple atomic operations into a single action.

Compared to the logics mentioned above, we use a simple form of invariant. Due to this simplicity, our invariants are easy to verify by encoding them in existing tool supported languages. Using thread local variables in a systematic way, our notion is powerful enough to prove data race freedom. Rather than designing extra features for functional verification into the invariant mechanism, we use a separate mechanism which offers a better separation of concerns.

The claim made about IRIS [55] that invariants and monoids are all that one needs to reason about concurrent software is in spirit identical to the claim we make in this paper. Invariants in our approach are similar to IRIS's invariants, but, while our process algebra is a monoid, the mechanism for executing actions has no equivalent in IRIS. Each action block consists of a number of atomic steps that are independent of any other step that it may be interleaved with.

Our logic also satisfies the requirements put forth by Da Rocha Pinto *et al.* [30]: Our thread-local state is auxiliary state, histories provide interference abstraction, we have resource ownership through separation logic, and we get atomicity through the use of atomic methods and blocks.

Our approach is also in line with the works of Jones *et al.* [53, 52], which proposes a limit to the expressive power of specification formalisms in order to keep specifications analyzable and warn of not using auxiliary variables beyond the point where they are appropriate.

We have used thread-local specification patterns in our earlier work on atomic operations [2], OpenCL kernel programs [7], and Parallel Loops [18]. The encodings used to implement the latter two are similar in style to the ones introduced in this paper: they translate the proof requirements into a method that must be checked and modify code by inserting exhale and inhale

instructions.

Rely-Guarantee reasoning [50] is a reasoning style that is intuitive and often elegant. We believe it is possible to encode this style into layer 2 at the price of a large amount of (automatically) added annotations. It is much easier to employ rely-guarantee reasoning at layer 3: we can put the properties on which an action relies as its pre-condition and put the properties it guarantees as its post-condition.

Several directions can be considered as future work. It might be possible to add other layers to the stack (or have a branching structure of verification techniques) to enable verification of other classes of properties. We also see that verification at the moment requires a large amount of annotations, and we plan to investigate if some of these can be generated automatically. Finally, we need to do large verification case studies, to show how well the mechanical verification scales. Especially, in layer 2, we are interested in case studies with richer concurrency protocols.

CHAPTER 8

Specification and Verification of Atomic Operations in GPGPU Programs

General purpose GPU (GPGPU) programming enables programmers to use the power of massively parallel accelerator devices to solve computationally intensive problems with a significant speed up. However, massive parallelism also makes programming more error prone: data races might be difficult to detect, and moreover ensuring functional correctness becomes a challenge. To address this issue, different verification techniques for GPGPU programs have been developed [21, 17], based on Separation Logic and abstraction, respectively. However, these techniques do not support reasoning about functional properties of kernels using atomic operations. This chapter discusses how the Separation Logic approach to reason about GPGPU programs is extended to reason about programs that use atomics for synchronisation.

GPU programming is based on the notion of *kernels*. A kernel consists of a large number (typically hundreds) of parallel threads that all execute the same instructions. The GPU execution model is an extension of the *Single Instruction Multiple Data* (SIMD) model[1], in which each thread executes the same instruction but on different data. For efficiency reasons, threads are grouped into *work groups*. Each work group has its own *local memory*, shared among all threads in the work group. Further, the kernel has a *global memory*, which is shared among all threads on the GPU device. Threads within a work group usually synchronise by *barriers*. Atomic operations provide asynchronous updates on shared memory locations (either in global or local memory) and are the only mechanism to support inter-group synchronisation in GPU programs. Moreover, atomic operations are also sometimes used for synchronisation within a work group, because they enable more flexible parallel behaviors than using barriers alone. For example, the *Parallel add* example in Section 8.2 and the *Histogram* example in the Parboil benchmark [80] benefit from the flexible parallel behavior of atomic operations.

In an earlier work, Blom *et al.* [21] used permission-based Separation Logic to reason about data race freedom and functional correctness of GPGPU kernels that use barriers as the *only* synchronisation construct. This chapter extends this logic to reason about kernels that also use *atomic operations*. The main idea of our work in this chapter is to adapt the notion of

[1] To be precise, the GPU execution model is Single Instruction Multiple Thread (SIMT), which extends SIMD with more flexibility in the control flow.

resource invariants in Concurrent Separation Logic (CSL) to reason about the behavior of atomic operations w.r.t. the GPU memory hierarchy.

Resource invariants capture the properties of shared memory locations. These properties only may be violated by a thread that is in the critical section, and thus has exclusive access to the shared memory locations. Before leaving the critical section, the thread has to ensure that the resource invariants are re-established. Because of the GPU memory hierarchy, shared memory locations can be both in local memory (shared between threads in a single work group) and in global memory (shared between all threads). Therefore, in our approach we use *group resource invariants* that capture the properties for local shared memory locations, and *kernel resource invariants* to capture the properties for global shared memory locations. For each kernel, there always is a single kernel resource invariant, while for each work group there is a group resource invariant. However, by parametrizing the group resource invariant with the group identifier *gid*, this can be specified with a single formula.

Note that we use the term shared memory locations instead of atomic variables, because the atomicity of a variable may change between different barrier intervals. Therefore, resource invariants should be re-established when a thread executes either an atomic operation or a barrier.

The remainder of this chapter is organized as follows. After some background information, Section 8.2 explains how the behavior of GPGPU kernels with atomic operations is specified. Then, Section 8.3 formalizes our approach, while we conclude with related work in Section 8.4.

8.1 Background

This section first gives a short background of the approach proposed by Blom *et al.* [21] to use CSL in reasoning about GPGPU programs with barriers. Later, we extend the logic to cover GPGPU programs employing both barriers and atomic operations.

Reasoning about GPGPU Programs

In the approached proposed in [21], kernels, work groups, threads, and barriers are specified and verified modularly w.r.t. their specifications.

We illustrate the approach using the example in Listing 51, which contains a kernel program annotated with a *thread specification*, plus a *barrier*

```
/*@  requires Perm(a[gtid],1) ** Perm(b[gtid],1);
      ensures  Perm(b[gtid],1) ** b[gtid] = (gtid+1) % gsize; @*/
 kernel void rotate(global int a, global int b){
   a[gtid]=gtid;
   barrier(global){
   /*@  requires a[gtid]=gtid;
         ensures Perm(a[(gtid+1) % gsize],1/2) ** Perm(b[gtid],1);
         ensures a[(gtid+1) % gsize]=(gtid+1) % gsize;  @*/
   }
   b[gtid]=a[(gtid+1) % gsize];
 }
```

Listing 51: An example of a kernel with specifications

specification for each barrier The specifications use the keywords gtid to denote the global thread identifier, and gsize to denote the number of threads in each work group, respectively. A thread specification specifies the permissions a thread should hold before (keyword **requires**) and after (keywords **ensures**) execution, together with the thread's functional behavior. In the example, write permission to position gtid of both array a and b is required and it is ensured that position gtid of array b can be written and contains (gtid+1)% gsize. To illustrate the use of a barrier, the kernel is implemented in a non-standard way: first gtid is assigned to a[gtid] and then access to the array is rotated by synchronisation on a barrier, after which the thread reads a[(gtid+1) % gsize]. This rotation is specified with a *barrier specification*, which specifies (1) how permissions are redistributed over the threads in the work group, and (2) the functional pre- and post-conditions that must hold before and after execution of the barrier.

There are two ways to specify the redistribution of permissions at a barrier in a work group. First, one can choose to redistribute all permissions available to the work group, assuming that each thread loses all permissions at a barrier. Second, one can force the user to explicitly specify which permissions are lost. The original paper, *i.e.* [21] and the example use the first approach, which is efficient for proving data race freedom. In the rest of this chapter, we use the second approach, which is more convenient for functional properties, as it ensures all functional properties are properly *framed* [21].

Given a thread specification which is parametrized by *gtid*, the *group specification* and *kernel specification* are defined as the *universal separating conjunction* of the thread specification over all threads in the same work group and over all threads in the GPU, respectively. Thus, group and kernel specifications are automatically derived from the thread specifications, and do not have to be explicitly given. Group specifications capture the resources in global memory that can be used by the threads in a particular work group, including its pre- and post-condition. Notice that locations defined in local memory are only valid inside the work group and thus the work group always holds write permissions for these locations. In the kernel specification, resources that are required from the host program along with the necessary pre-conditions and provided post-conditions are specified. An invocation of a kernel by a host program is correct if the host program transfers the necessary resources and fulfills the kernel pre-conditions.

8.2 Specification

This section discusses two examples that illustrate our approach to the specification of kernels with atomic operations. The first example uses a single atomic add; the second example illustrates how we reason about kernels which use both barriers and atomic operations for synchronisation, and where the atomicity of a variable may change in different barrier intervals.

Specification of a Kernel with Parallel Addition

Listing 52 contains an annotated parallel add kernel, where ltid indicates the local thread identifier. For simplicity, in this first example we assume that we have a single work group[2], later we extend our technique also to multiple work groups. We first explain the permission specifications, followed by an explanation of the functional properties (the highlighted annotations).

In Listing 52, each thread atomically adds its contribution, which is stored in values[ltid], to the shared variable x. The **requires** and **ensures** clauses express a single thread's pre- and post-conditions. The pre-condition specifies that each thread needs to have read permission on its corresponding index of values. Additionally, we specify a group resource invariant for the

[2]The number of work groups is determined in the host code before launching the kernel.

```
/*@ given int cont[gsize] ;
2  group invariant Perm(x,1)** Perm(cont[*],1/2) ** x==(\sum cont[*]) ;
   requires Perm(values[ltid],1/2)** Perm(cont[ltid],1/2) **
4               cont[ltid]==0 ;
   ensures Perm(values[ltid],1/2)** Perm(cont[ltid],1/2) **
6                     cont[ltid]==values[ltid] ; @*/
   kernel void gpadd(local int x, local int values){
8    atomic_add(x,values[ltid]) /*@ then { cont[ltid]=values[ltid]; } @*/;   }
```

Listing 52: Specification of parallel add in a work group.

local shared memory variable x, which expresses that the thread executing the atomic add operation has exclusive write access to x. With this specification, it is straightforward to prove that the program is free of data races, as it is guaranteed that there is only one thread executing the atomic operation and exclusively accessing the shared variable.

To reason about functional properties, the specification expresses the accumulative contributions of the threads on the shared variable. To track these contributions, we use an array cont[ ], added as a ghost parameter (line 1) to the kernel. The idea is that the contribution of each thread (cont[ltid]) is 0 before it executes and values[ltid] after it finishes, while the invariant $\sum_{i=0}^{\texttt{gsize-1}} \texttt{cont}[i] = \texttt{x}$ is maintained in order to prove that the kernel computes the sum of the values. To make this work, the thread's pre-condition (line 4) states that each tread obtains a read permission on cont[ltid], in order to be able to use cont in the specifications. Each thread has to track its contribution towards the total in x in its own location in the cont array. This is done during the atomic operation by injecting an assignment statement as ghost code (specified as a then clause, see line 8). The thread executing atomic_add, first adds values[ltid] to x, and then executes the injected ghost code, *i.e.* cont[ltid]=values[ltid]. To achieve this, the group resource invariant is extended with a half permission on *all* elements of cont, written Perm(cont[*],1/2) [3]. Thus, when thread ltid at the

[3]This is syntactic sugar for universal quantification of the permissions over all the

beginning of the atomic body obtains the resource invariants, it has *twice* a read permission Perm(cont[ltid],1/2), which can be combined into a single write permission Perm(cont[ltid],1).

Parallel Addition with Multiple Work Groups

As a next example, we discuss the specification of a kernel with multiple work groups, which employs both barriers and atomic operations for synchronisation. This is a common pattern to avoid making global memory access a bottleneck: first all threads in a work group compute an intermediate result in local memory, then the intermediate result is combined with the global result in global memory. It is used, for example, in the parallel implementation of BFS in the Parboil benchmark [80]. The kernel in Listing 53 is an extension of the previous example, using multiple work groups and a barrier, where ksize denotes the number of work groups. The kernel is implemented by the following steps: (1) each thread atomically adds its element of the global array values to its local accumulator, *i.e.* a locally shared variable x; (2) all threads within a work group are synchronised by a barrier (line 16); (3) after all threads have passed the barrier, one thread per work group (here ltid$= 0$) adds the work group's final value of x to a *globally* shared variable r (line 23). Eventually, r contains the collective contributions of all the threads in the kernel. Similar to the single work group example, to track the contributions at each step, the kernel program uses ghost arrays cont and sums, with all elements initialized with zero. We use cont to specify the current value of the local variable x. Similarly, array sums is used to sum up the total accumulated contributions of the work groups. Updating the local cont is explained in the previous example. In a similar way, using the ghost code at line 23, in each work group, the thread with ltid$= 0$ stores its contribution (the final value of x) to the global sums[gid], *i.e.* the index corresponding to the executing work group from the sums array.

In Listing 53, there are two invariants that are maintained:

1. $\sum_{i=0}^{\texttt{gsize}-1} \texttt{cont}[i] = \texttt{x}$ for each work group; and
2. $\sum_{i=0}^{\texttt{ksize-1}} \texttt{sums}[i] = \texttt{r}$ for the kernel.

---

indices of cont[].

```
/*@ given global int sums[ksize]={0} ;
2   given local int cont[gsize]={0}, region=0 ;
    kernel invariant
4    Perm(r,1)** Perm(sums[*],1/2) ** r==(\sum sums[*]) ;
    group invariant Perm(region,1/(gsize+1))**Perm(x,region==0?1:1/2)
6          ** Perm(cont[*],1/2) ** x==(\sum cont[*]) ;
    requires Perm(region,1/(gsize+1))**Perm(values[gtid],1/2);
8   requires Perm(cont[ltid],1/2) ** cont[ltid]==0 ;
    requires ltid==0 ==> Perm(sums[gid],1/2)**sums[gid]==0 ;
10  ensures Perm(region,1/(gsize+1))**Perm(values[gtid],1/2);
    ensures Perm(cont[ltid],1/4) ** cont[ltid]==values[gtid] ;
12  ensures ltid==0 ==> Perm(cont[*],1/4)**Perm(sums[gid],1/2) ;
    ensures ltid==0 ==> sums[gid]==(\sum cont[*]) ; @*/
14 kernel void KParallelAdd(local int x, global int values, global int r){
     atomic_add(x,values[gtid]) /*@ then { cont[ltid]=values[gtid]; } @*/;
16   barrier(local)/*@
      requires Perm(region,1/(gsize+1))**region==0}**
18          Perm(cont[ltid],1/4) ;
      ensures Perm(region,1/(gsize+1))**region==1;
20    ensures ltid==0 ==> Perm(cont[*],1/4)**x==(\sum cont[*]) ;
     { region=1; } @*/;
22   if(ltid==0)
       atomic_add(r,x)/*@ then { sums[gid]=x; } @*/;   }
```

Listing 53: Specification of global parallel add.

After termination of work group gid, we use the group invariant to conclude that:

$$\texttt{sums[gid]} = \sum_{i=\texttt{gsize}\times\texttt{gid}}^{\texttt{gsize}\times\texttt{gid}+\texttt{gsize}-1} \texttt{values[}i\texttt{]} \ .$$

Hence after termination of all work groups we can prove that:

$$\mathtt{r} = \sum_{i=0}^{\mathtt{ksize-1}} \mathtt{sums}[i] = \sum_{j=0}^{\mathtt{ksize-1}} \sum_{i=j\times\mathtt{gsize}}^{(j+1)\times\mathtt{gsize}-1} \mathtt{values}[i]$$

Again, we first explain the permission specifications. The permission specifications for values are similar to the specifications in Listing 52. The barrier divides the program into regions, and within a region the distribution of permissions over the threads and the resource invariants does not change. Only when all threads reach the barrier, permissions may be redistributed. This means in particular that a variable that is treated as a shared memory variable in one region, may become unshared in a next region (or vice versa). Thus, resource invariants often depend on the current barrier region. To keep track of the current barrier region, we use a ghost variable region initialised at 0 (line 2). Each thread at all times has read access to this region variable, and whenever all the threads go through the barrier, the region is updated (see line 21). The group resource invariant specifies that within region 0 (before the barrier instruction), variable x is a shared variable in local memory, while in region 1 (after the barrier), x is not shared any more. So, after the barrier x can be read concurrently by all the threads within a work group. The kernel resource invariant specifies that r is a shared variable in global memory, but that only threads with a local thread identifier 0 are able to correctly update r, because only threads with ltid$=0$ can construct a write permission of sums[gid] (see lines 4 and 9) to store the contributions.

The barrier specification expresses that threads keep read access on region, and that the value of region is updated to 1. Moreover, the specification asserts that upon entering the barrier each thread gives up $1/4$ permission to access its contribution element, *i.e.* cont[ltid]. The barrier redistributes these permissions to the thread with ltid$=0$, which ensures that the thread with ltid$=0$ has sufficient permissions to frame (\sum cont[$*$]) in the barrier post-condition. Notice that when all threads have reached the barrier, all read accesses on region together (including the group resource invariant) can be combined into a write permission on region, thus enabling the update of this ghost variable within the barrier.

Next, we discuss the functional property specifications. As we stated before, two resource invariants specify the values of the shared variables: (1) the local shared variable x must always express the accumulation of the con-

tributions of the threads executing the first atomic operation (line 6), and (2) the global shared variable r must always express the accumulation of x's final value in each work, group which is stored in sums[gid] (line 4). To prove these invariants, each thread must ensure that it correctly stores its contribution as specified in line 11. Moreover, the barrier must ensure that the thread with ltid= 0 knows the final value of x as specified by x==(\sum cont[$*$]) in the barrier's post-condition. Finally, the thread with ltid= 0 must guarantee that the final value of x is stored in sums[gid] (line 13). Therefore, the verifier can prove that the value of r is the collective contributions of all the threads in the kernel.

8.3 Formalization

The previous section illustrated how we specify permissions and functional properties of kernel programs in the presence of atomic operations and barriers on several examples. This section defines the approach formally. Rather than presenting this work on the full language, we will present it for a core kernel programming language. In our verification technique barrier divergence is not taken into consideration, *i.e.* if threads in a work group arrive at a barrier they all arrive at the *same one*. This is a realistic assumption: according to the OpenCL semantics, the behavior of programs with barrier divergence is unspecified [67]. Moreover, in our earlier work [21], we proposed syntactical restrictions to determine whether a kernel programs is free of barrier divergence.

We first introduce syntax and semantics of our core kernel language, and also formally define the formula language to write the specifications. Then we present the Hoare logic rules used to reason about kernels with atomics, and we prove soundness of the proof rules. Finally, we also briefly discuss tool implementation.

Syntax and Semantics

Programming Language Figure 8.1 presents the syntax for our kernel programming language, which adapts the Kernel Programming Language (KPL) of [17] by extending it with atomic operations and changing the barrier statement. For simplicity, in this language, global and local memory are assumed to be single shared arrays. There are two local memory access operations: read from location e_1 in local memory ($v := \mathsf{rdloc}(e_1)$), and

write e_2 to location e_1 in local memory ($\mathsf{wrloc}(e_1, e_2)$). Similarly, read and write operations in global memory are represented by $v := \mathsf{rdglob}(e)$ and $\mathsf{wrglob}(e_1, e_2)$, respectively. W.r.t. to the original KPL language, barriers are different. As in KPL, a barrier is labelled with a flag F, which denotes which memories it synchronises. That is, it always acts both as synchronisation between the threads in a work group and as a memory fence. Depending on the flag, it is either for local or for global memory. Additionally, a barrier is labelled with an identifier *bid*, which is used to distinguish different barrier instances, and it is extended with a block of statements to be executed while all threads are in the barrier. Further, we add an atomic block statement to the language, which a label to denote whether it accesses global or local shared memory. The (annotated) OpenCL atomic operations can be easily embedded into this atomic block statement.

The state of a kernel program consists of the state of the global memory, the states of the local memories and the state of all the threads. On these states, three steps are possible:

1. A thread performs a non-atomic statement, see [21] for details of the operational semantics;

2. A thread *atomically* performs all statements in an $\mathsf{atomic}(F)\{S\}$ block. Its operational semantics is standard and can be defined easily, similar to [85].

3. All threads running in the work group go through the barrier bid : $\mathsf{barrier}(F)\{S\}$. This can only happen if all threads in a group are waiting to execute S. The effect on the state is that all statements in S are performed, and all threads in the group consider *bid* as performed. The operational semantics of a barrier without a body is defined in [21]. However, its extension with a body is trivial as the body is executed atomically.

Note that because barriers are labelled in KPL, any program that exhibits barrier divergence will block forever and therefore does not terminate.

Formula Language The specifications of KPL programs can be written using the following formula language:

Reserved global identifiers (constant within a thread):

gtid	Thread identifier with respect to the kernel
gid	Group identifier with respect to the kernel
ltid	Local thread identifier with respect to the work group
tc	The total number of threads in the kernel
gs	The number of threads per work group
ks	The number of groups in the kernel

Kernel language:

$$
\begin{array}{lcl}
b & ::= & \text{boolean expression over global constants and private variables} \\
e & ::= & \text{integer expression over global constants and private variables} \\
S & ::= & \mathsf{v\ :=\ e} \mid v := \mathsf{rdloc}(e) \mid v := \mathsf{rdglob}(e) \mid \mathsf{wrloc}(e_1, e_2) \\
 & \mid & \mathsf{wrglob}(e_1, e_2) \mid \mathsf{nop} \mid S_1;S_2 \mid \mathsf{if}\ b\ \mathsf{then}\ S_1\ \mathsf{else}\ S_2 \mid \mathsf{while}\ b\ \mathsf{do}\ S \\
 & \mid & \mathsf{atomic}(F)\{S\} \mid bid : \mathsf{barrier}(F)\{S\} \\
F & ::= & \mathsf{local} \mid \mathsf{global}
\end{array}
$$

Figure 8.1: Syntax for Kernel Programming Language

$$
\begin{array}{lcl}
E & ::= & \text{expressions (in first-order logic) over global constants,} \\
 & & \text{private variables, } \mathsf{rdloc}(E), \mathsf{rdglob}(E).
\end{array}
$$

$$
\begin{array}{lcl}
R & ::= & \mathsf{true} \mid E \mid \mathsf{LPerm}(E, p) \mid \mathsf{GPerm}(E, p) \mid R_1 \star R_2 \mid E \Rightarrow R \\
 & & \mid \bigstar_{v:E(v)} R(v)
\end{array}
$$

where we use $\mathsf{LPerm}(E, p)$ and $\mathsf{GPerm}(E, p)$ as explicitly different permission statements to specify accesses to local and global memories, respectively. In addition to the separating conjunction of two resource formulas, we also have guarded resource formulas, and a universal separating conjunction quantifier, which quantifies over the set of values v for which $E(v)$ is true. Formalization of the specification language and validity of the formulas are elaborated in [21].

The behavior of kernels, work groups, threads, and barriers are defined as $(K_{pre}, K_{post}, K_{rinv})$, $(G_{pre}, G_{post}, G_{rinv})$, (T_{pre}, T_{post}), and (B_{pre}, B_{post}), respectively. Note that the user only has to annotate a kernel resource invariant K_{rinv}, a group resource invariant G_{rinv} parametrized by group id, a thread's pre- and post-condition T_{pre} and T_{post} and barrier's pre- and post-condition B^{bid}_{pre} and B^{bid}_{post}. We can derive the work groups' pre- and post-conditions, *i.e.* G_{pre} and G_{post}, as the separating conjunction of the pre-

$$\frac{}{K_{rinv}, G_{rinv}(gid) \vdash \{R[v := e]\}\, \mathsf{v} \ := \ \mathsf{e}\, \{R\}} \textbf{[Assign]}$$

$$\frac{}{K_{rinv}, G_{rinv}(gid) \vdash \begin{array}{l} \{\mathsf{LPerm}(e, \pi) \star R[v := L[e]]\} \\ v \ := \ \mathsf{rdloc}(e) \\ \{\mathsf{LPerm}(e, \pi) \star R\} \end{array}} \textbf{[LRead]}$$

$$\frac{}{K_{rinv}, G_{rinv}(gid) \vdash \begin{array}{l} \{\mathsf{LPerm}(e_1, 1) \star R[L[e_1] := e_2]\} \\ \mathsf{wrloc}(e_1, e_2) \\ \{\mathsf{LPerm}(e_1, 1) \star R\} \end{array}} \textbf{[LWrite]}$$

$$\frac{\begin{array}{l} S \text{ refers to local memory only.} \\ K_{rinv} \vdash \{P(t) \star G_{rinv}(gid)\}\ S\ \{G_{rinv}(gid) \star Q(t)\} \end{array}}{K_{rinv}\ , G_{rinv}(gid) \vdash \ \{P(t)\}\ \mathsf{atomic}(\mathsf{local})\{S\}\ \{Q(t)\}} \textbf{[LAtomic]}$$

$$\frac{\begin{array}{l} S \text{ refers to global memory only.} \\ G_{rinv}(gid) \vdash \{P(t) \star K_{rinv}\}\ S\ \{K_{rinv} \star Q(t)\} \end{array}}{K_{rinv}\ , G_{rinv}(gid) \vdash \ \{P(t)\}\ \mathsf{atomic}(\mathsf{global})\{S\}\ \{Q(t)\}} \textbf{[GAtomic]}$$

Figure 8.2: Important Hoare logic rules: Read, Write and Atomic

and post-conditions of all threads belonging to the work group and the work group's resource invariant. Similarly, the kernel's pre- and post-condition, *i.e.* K_{pre} and K_{post}, can be derived automatically as the separating conjunction of the pre- and post-conditions of all work groups belonging to the kernel and the kernel's resource invariant.

Verification

Since we derive the contracts for work groups and kernels automatically, we can verify a kernel program by verifying all the threads belonging to a kernel. To verify a thread T, with body T_{body}, the following Hoare triple should be verified, using the verification rules defined in Figure 8.2 and Figure 8.3:

$$K_{rinv}, G_{rinv}(gid) \ \vdash \ \{T_{pre}\}\ T_{body}\ \{T_{post}\}$$

S, R, and E refer to local memory only.

$$\dfrac{K_{rinv} \vdash \begin{array}{c}\{\bigstar_{t\in[0..gs)} R(t) \star G_{rinv}(gid)\} \\ S \\ \{G_{rinv}(gid) \star \bigstar_{t\in[0..gs)} E(t)\}\end{array}}{K_{rinv}\ , G_{rinv}(gid) \vdash \begin{array}{l}\{P(t) \star R(t)\} \\ \textbf{barrier}(\mathsf{local}) \\ \textbf{req}\ R(t);\quad \textbf{ens}\ E(t);\ \{S\} \\ \{P(t) \star E(t)\}\end{array}}\ \textbf{[LBarrier]}$$

S, R, and E refer to global memory only.

$$\dfrac{G_{rinv}(gid) \vdash \begin{array}{c}\{\bigstar_{t\in[0..gs)} R(t) \star K_{rinv}\} \\ S \\ \{K_{rinv} \star \bigstar_{t\in[0..gs)} E(t)\}\end{array}}{K_{rinv}\ , G_{rinv}(gid) \vdash \begin{array}{l}\{P(t) \star R(t)\} \\ \textbf{barrier}(\mathsf{global}) \\ \textbf{req}\ R(t);\quad \textbf{ens}\ E(t);\ \{S\} \\ \{P(t) \star E(t)\}\end{array}}\ \textbf{[GBarrier]}$$

Figure 8.3: Important Hoare logic rules: Barriers

In addition to the standard rules for sequential compositional, conditionals, loops, and weakening, Figure 8.2 and Figure 8.3 shows the most important Hoare logic rules to reason about kernel threads. Rule **[Assign]** describes the updates to the thread's private memory. Rules **[LRead]** and **[LWrite]** specifies read and write of local memory[4]. The rules for global

[4] $L[e]$ denotes the value stored at location e in the local memory array, and substitution is as usually defined for arrays, cf. [12]:

$$L[e][L[e_1] := e_2] = (e = e_1)?e_2 : L[e]$$

memory are defined similarly, but for space reasons are not presented here. The rules **[LAtomic]** for local and **[GAtomic]** for global atomic operations are simple instances of the CSL rule using the group resource invariant and kernel resource invariant, respectively.

The rule **[LBarrier]** reflects the functionality of the barrier with a flag indicating that it synchronises local memory. It acts similar to the CSL rule for the group resource invariant but at the same time it collects resources and knowledge from all threads and redistributes these resources and knowledge. To do so it requires that the block S can be executed given the resources provided by the invariant (G_{rinv}) and all threads in the work group ($R(t)$). Moreover, it ensures that all resources are given back ($E(t)$) and the invariant is re-established (G_{rinv}). The rule also says that the effect of passing through a barrier on a thread is to give up resources $R(t)$ and get $E(t)$ in return. Note that there is a side condition that S, R and E can refer to local memory only, as this would otherwise potentially create a data race: a local barrier functions as a memory fence for local memory, thus it can exchange information about local memory without any difficulties, but no order on global memory is guaranteed. The **[GBarrier]** rule is symmetric in the use of local vs. global memory and invariants. Note that the local/global flag affects memory only. Both uses of the barrier synchronise the threads within a single work group.

Soundness

Finally, we prove soundness of our verification technique.

Theorem 8.3.1. *Given a barrier divergence free kernel, for which the thread level Hoare triples are provably correct. Then every possible execution of the kernel starting in a state that satisfies the kernel pre-condition is data race free and ends in a state that satisfies the kernel post-condition.*

Proof. We are given a finite trace of executions. In this trace every thread $t_{gid,ltid}$ makes a finite number of steps $N_{gid,ltid}$, where atomic blocks and barriers count as one step. Because a Hoare logic proof of the thread exists, we can find formulas $P^0_{gid,ltid}, \cdots, P^{N_{gid,ltid}}_{gid,ltid}$ that are valid before, between and after these steps, where $P^0_{gid,ltid}$ is the pre-condition of the thread and $P^{N_{gid,ltid}}_{gid,ltid}$ is its post-condition.

All states $\sigma_0, \cdots, \sigma_N$ in the finite global trace of N steps can be described by a function f that maps each global trace position to the positions in the local threads. We do not know in which order the steps of the threads are executed, but we know they all start in position 0, so $f(0, gid, ltid) = 0$. We also know they end in their last state, so: $f(N, gid, tid) = N_{gid,ltid}$.

We claim that before and after every step in the trace the state satisfies a specific Separation Logic formula.

$$\forall i = 0, \cdots, N : \ \sigma_i \models K_{rinv} \star \mathop{\bigstar}_{gid \in [0..ks)} \left(G_{rinv}(gid) \star \mathop{\bigstar}_{ltid \in [0..gs)} P_{gid,ltid}^{f(i,gid,ltid)} \right)$$

This claim is proven by induction on i. For $i = 0$ this is precisely the given pre-condition. Assuming that the claim is correct for $0 \leq i < N$, then there are three cases. If the step is a plain step or an atomic step, by correctness of the standard CSL Hoare triple used to prove that step, the validity for $i + 1$ follows.

The interesting case is the barrier step, in which all threads of a group are involved. The Hoare triple for each thread is valid so each thread starts knowing $P(t) \star R(t)$ and ends knowing $P(t) \star E(t)$. Because of the correctness of the standard CSL Hoare triple for the barrier statement S, the change to the state is from $\mathop{\bigstar}_{t \in [0..gs)} R(t) \star G_{rinv}(gid)$ to $\mathop{\bigstar}_{t \in [0..gs)} E(t) \star G_{rinv}(gid)$, which is precisely the change in the formulas, so $i + 1$ is established.

The last statement is precisely the kernel post-condition which proves that the end state satisfies the kernel post-condition.

A data race happens if: there is an access to a location l in step i_1 by thread t_1, followed by an access to the same location in step i_2 by thread t_2, there is no memory fence in between these accesses, and one of these accesses is a write. Suppose that t_1 used fraction p_1 for the access and thread t_2 used fraction p_2. Because one of the accesses is a write, $p_1 + p_2 > 1$. Because there is no memory fence, that is no barrier or atomic in between, at time i_1 thread t_2 must have already owned fraction p_2. Thus at time i_1, fraction $p_1 + p_2$ permission for location l existed, which leads to a contradiction. □

Tool Support

We have implemented tool support for the verification of kernels in the VerCors tool set [19]. The VerCors tool set compiles programs that are specified in a complex specification language, such as kernels, into much

simpler specified programs and then verifies the latter to prove that the former are correct. The main compilation target used for kernel programs is Silver, the intermediate language of the Viper framework [54]. Silver is a specification language designed along the lines of Implicit Dynamic Frames [78]. We can then verify these Silver programs with the Silicon tool that is part of the framework.

For the verification of kernels with atomics, two transformation passes have been added to the VerCors tool set. The first pass transforms a kernel into an intermediate form that uses the same barrier and atomic constructs as used in the kernel programming language used in this section. The second pass replaces those atomic and barrier constructs with code that mimics the conclusion of the corresponding proof rules (see Figure 8.2 and Figure 8.3) and adds code that encodes that the premises of the rule is valid. The replacement ensures that when using a barrier or atomic proof rule the program is correct. The added code verifies that the rule is used correctly.

8.4 Conclusion and Related Work

This chapter presented an approach to specify and verify GPGPU programs in the presence of atomic operations and barriers. The main characteristics of the approach are that it can be used to prove both data race freedom and functional correctness. To specify the shared memory accesses, the notion of resource invariant from CSL is lifted to the GPU memory model, distinguishing between kernel and group resource invariants. An appropriate Hoare logic is proposed and proven sound to reason about GPGPU programs using atomic operations and barriers. The approach is illustrated on some examples, and supported by an implementation in the VerCors tool set. The current version of the tool set supports full verification of the examples in this chapter, but only when implemented in PVL[5], available at [13]. It is ongoing work to update the VerCors tool set to verify these examples in OpenCL .

There is very little related work in this area, as reasoning techniques for GPU kernels are still relatively fresh. Bardsley *et al.* propose additional support in GPUVerify for reasoning about GPU kernels where warps and atomic operations are used for synchronisation [14]. In GPUVerify the user

[5] Prototypal Verification Language (PVL) is a simple language designed for verification purposes in VerCors [74].

does not need to add specifications manually, because the tool internally speculates and refines kernel specifications [17]. However, GPUVerify is not able to reason about the functional properties of kernels, it can only prove absence of data races.

Concerning verification of GPU kernels, we should also mention the work of Li and Gopalakrishnan [60]. They verify CUDA programs by symbolically encoding thread interleavings. They were the first to observe that to ensure data race freedom it was sufficient to verify the interleavings of two arbitrary threads. For each shared variable they use an array to keep track of read and write accesses, and where in the code they occur. By analysing this array, they detect possible data races. However, they do not consider atomic operations.

In the verification of (general) concurrent programs synchronised with barriers, Hobor *et al.* [45] propose a sound extension of CSL for Pthreads-style barriers. The simplicity of the OpenCL barriers makes our specification simpler. Additionally, we support barriers in the presence of atomic operations.

At the moment, the user of VerCors still has to write quite a substantial amount of annotations to make verification work. As a future work, it will be investigated how to make use of inference techniques for program annotations to reduce this annotation burden. For example, it would be interesting to investigate if GPUVerify could be used to infer some of the annotations that we need.

CHAPTER 9

Conclusions

In shared memory concurrency, unpredictable interference of threads that access shared memory makes it hard to implement a thread-safe program. The notorious difficulty of implementing correct shared memory concurrent programs makes it necessary to develop techniques that reason about thread-safety. However, reasoning about a concurrent program is challenging, because a verification technique has to consider all possible interleavings between threads. To make verification tractable, we study modular verification techniques. For such techniques, the major challenge is how global properties can be established from thread local properties. In this thesis, we study how we can efficiently describe and reason about a thread's local contribution, when a thread is used in a concurrent environment, and other threads can update shared variables accessed by the thread.

For a long period, lack of a robust logic in reasoning about shared variables has been one of the impediments to develop effective and practical axiomatic verification methods for concurrent programs. Reynolds introduced Separation Logic to reason about safety properties of imperative programs that use shared mutable data structures. Later, O'Hearn demonstrated that the core concepts of Separation Logic are also suitable to reason about concurrent shared memory programs. This triggered a new trend in the verification of concurrent programs leading to a whole collection of CSL extensions. In this trend, permission-based Separation Logic showed its potential power to be a more practical approach for the verification of programs written in a Java-like language.

This thesis demonstrates how permission-based Separation Logic can be used to specify and verify synchronisers, which are the core elements of any concurrent program. The intention of this thesis was not to propose techniques for the fully-automated verification of synchronisation mechanisms. Neither was it to develop yet another new logic for concurrent data structures verification. Instead, the permission-based Separation Logic from Hurlin and Haack [46, 41] is lifted into the specification language to specify and verify expected behaviour of the main synchronisers available in the java.util.concurrent library. This allowed us to develop the VerCors tool-set that can reason about correctness of concurrent Java programs synchronised with various synchronisers. This synchronisers can be either coarse-grained synchronisation classes like ReentrantLock, Semaphore, CoundDownLatch or RecyclicBarrier, or fine-grained synchronisation primitives like AtomicInteger and AtomicReference.

Results

This thesis identifies two main elements to reason about: 1. the threads, and 2. the synchronization constructs of a concurrent program. These elements are studied through the thesis with the following contributions:

- Our goal in Chapter 3 was to prove the correctness of a concurrent Java program with multiple join points. We explored the basics of threads synchronisations, like thread start and join. We proved the correctness of a general concurrent pipeline processing pattern with multiple join points implemented in Java.

- Our goal in Chapter 4 was to achieve an unified technique to express the behaviours of various synchronisation constructs. We have achieved this goal through the specification of various synchronisation constructs like Lock, Semaphore, CountDownLatch, CyclicBarrier. In more detail, we looked at various aspects of locks. We proposed a specification for the Lock interface which covers implementing classes like reentrant and read-write locks. We used similar structures to specify the contract of the Semaphore class. Behaviorally, a binary semaphore functions like a (single-entrant) lock. As can be seen by the specifications (see Section 4.1 and Section 4.2), if one implements a single-entrant lock the specification of the lock will be equivalent to the specification of a binary semaphore.

- Our goal in Chapter 5 and Chapter 6 was to prove the correctness of the specifications that we proposed in Chapter 4. In order to verify our proposed specifications, we tackled atomic operations, mainly from AtomicInteger, in two phases. In the first phase, presented in Chapter 5, we proposed a technique to specify atomic operations to reason about exclusive access synchronisation constructs, and then in the second phase (Chapter 6) we improved the specification to be used in the verification of both exclusive access and shared-reading synchronisation constructs. We proposed a contract for the AtomicInteger class such that one can prove correctness of both exclusive and shared-reading synchronisers. Having our final specification of the AtomicInteger opened an opportunity to improve the original specification of the synchronisation classes from Chapter 4.

- Our goals in Chapter 7 were to exploit our results from Chapter 4, Chapter 5 and Chapter 4 to reason about: 1. safety properties of concurrent pointer-based data structures, 2. reentrant locks (specified in Chapter 4), and 3. functional properties of concurrent data structures. We achieved our goals by proposing a layered verification technique where: 1. The first layer verifies the safety properties of concurrent data structures that use AtomicReference. We used VerCors to verify an implementation of ConcurrentLinkedQueue. 2. The second layer reasons about thread's local and global properties. We used VerCors to verify an implementation of ReentrantLock w.r.t. its specification from Chapter 4. 3. The third layer links our results to histories [22] to verify functional correctness of concurrent data structures. The verification is applied on verifying the correct behaviour of ConcurrentLinkedQueue.

- Our goal in Chapter 8 was to apply our approach in reasoning about atomic operations from Chapter 5 and Chapter 6 to verify the correct behaviour of atomic operations GPGPU programs. We linked our approach with [21] and demonstrated how one can reason about correct permission distributions and functional properties in the presence of both barriers and atomic operations as synchronisation constructs in GPGPU programs.

Future Directions

We have verified the implementation of locks (spin-lock and reentrant lock) using our specification for AtomicInteger. An interesting point to explore is verification of an implementation of a coupled read-write lock, where possibly two cooperating instances of an AtomicInteger are to be used for synchronisation.

Reasoning about several atomic synchronisers when they exchange their resources can lead to specify and verify Exchanger[1] from java.util.concurrent. To tackle this problem, techniques in which one can model interactions between atomic variables via message passing synchronisation [15] or channel based-based synchronisation [55] are helpful.

[1] An Exchanger is a synchronisation construct where two threads can exchange their resources with each other.

A proposal for the complete specification of the CyclicBarrier to include the second constructor with the internal barrier task (see Section 4.4) can also be done as a follow up of our study. Then, verifying the implementation of CyclicBarrier w.r.t. its new improved specification can be a goal to achieve. In this direction experiences in verification of barriers using Iris [36] can be applied in our work.

In reasoning about synchronisation constructs, *protocols* play an essential role. Any synchronisation construct needs to be instantiated with a protocol that specifies the correct behaviour of the algorithm. Current advances in Separation Logic-based specification and verification of protocols [36, 73, 22, 69] shows the potential of an unified method that verifies a custom synchronisation construct w.r.t. its specified protocol. Therefore, we believe an important step for the practical verification of complex and custom synchronisation algorithms is the unification of current logics and techniques.

Generally speaking, this unification seems necessary within the CSL-based verification community. Recently, Iris claims to have this unification in terms of a general purpose logic for concurrent programs verification. Currently, programs can be verified using Iris within an interactive theorem provers, like Coq. To support a semi-automatic verification tool, recently Caper [32] has been developed which is based on the predecessor of Iris, *i.e.* iCAP [81]. More verified case studies can demonstrate its practicality for the concurrent programming community.

All in all, recent progresses in concurrent program verification was surprising even for its pioneers when they proposed a logic for reasoning about pointer manipulating sequential programs. The Separation Logic-based sequential verification version already found its role in developing reliable industrial software [26]. Current results from the CSL-based verification of non-trivial concurrent Java and C programs feeds the hope that with an effort in unifying current techniques and enriching them with more automatic annotation generation methods we can make the dream of industrial error-free concurrent programs come true.

List of Publications

1. A. Amighi, P. de Carvalho Gomes, D. Gurov, and M. Huisman. Sound Control-Flow Graph Extraction for Java Programs with Exceptions. In *Software Engineering and Formal Methods - 10th International Conference, SEFM 2012, Thessaloniki, Greece, October 1-5, 2012. Proceedings*, pages 33–47, 2012.

2. A. Amighi, S. Blom, M. Huisman, and M. Zaharieva-Stojanovski. The VerCors Project: Setting Up Basecamp. In *Programming Languages meets Program Verification (PLPV 2012)*, pages 71–82. ACM Press, 2012.

3. A. Amighi, S. Blom, S. Darabi, M. Huisman, W. Mostowski, and M. Zaharieva-Stojanovski. Verification of Concurrent Systems with VerCors. In *Formal Methods for Executable Software Models - 14th International School on Formal Methods for the Design of Computer, Communication, and Software Systems, SFM 2014, Bertinoro, Italy, June 16-20, 2014, Advanced Lectures*, pages 172–216, 2014.

4. A. Amighi, C. Haack, M. Huisman, and C. Hurlin. Permission-based Separation Logic for multithreaded Java programs. *Logical Methods in Computer Science*, 11(1), 2015.

5. A. Amighi, S. Blom, M. Huisman, W. Mostowski, and M. Zaharieva-Stojanovski. Formal Specifications for Java's Synchronisation Classes. In *22nd Euromicro International Conference on Parallel, Distributed, and Network-Based Processing, PDP 2014, Torino, Italy, February 12-14, 2014*, pages 725–733, 2014.

6. A. Amighi, S. Blom, and M. Huisman. Resource Protection Using Atomics - Patterns and Verification. In *Programming Languages and*

Systems - 12th Asian Symposium, APLAS 2014, Singapore, November 17-19, 2014, Proceedings, pages 255–274, 2014.

7. A. Amighi, S. Darabi, S. Blom, and M. Huisman. Specification and Verification of Atomic Operations in GPGPU Programs. In *SEFM 2015*, pages 69–83, 2015.

8. A. Amighi, S. Blom, and M. Huisman. VerCors: A Layered Approach to Practical Verification of Concurrent Software. In *24th Euromicro International Conference on Parallel, Distributed, and Network-Based Processing, PDP 2016, Heraklion, Crete, Greece, February 17-19, 2016*, pages 495–503. IEEE Computer Society, 2016.

9. A. Amighi, P. de Carvalho Gomes, D. Gurov, and M. Huisman. Provably correct control flow graphs from Java bytecode programs with exceptions. *STTT*, 18(6):653–684, 2016.

Bibliography

[1] A. Amighi, S. Blom, S. Darabi, M. Huisman, W. Mostowski, and M. Zaharieva-Stojanovski. Verification of Concurrent Systems with VerCors. In *Formal Methods for Executable Software Models - 14th International School on Formal Methods for the Design of Computer, Communication, and Software Systems, SFM 2014, Bertinoro, Italy, June 16-20, 2014, Advanced Lectures*, pages 172–216, 2014.

[2] A. Amighi, S. Blom, and M. Huisman. Resource Protection Using Atomics - Patterns and Verification. In *Programming Languages and Systems - 12th Asian Symposium, APLAS 2014, Singapore, November 17-19, 2014, Proceedings*, pages 255–274, 2014.

[3] A. Amighi, S. Blom, and M. Huisman. VerCors: A Layered Approach to Practical Verification of Concurrent Software. In *24th Euromicro International Conference on Parallel, Distributed, and Network-Based Processing, PDP 2016, Heraklion, Crete, Greece, February 17-19, 2016*, pages 495–503. IEEE Computer Society, 2016.

[4] A. Amighi, S. Blom, M. Huisman, W. Mostowski, and M. Zaharieva-Stojanovski. Formal Specifications for Java's Synchronisation Classes. In *22nd Euromicro International Conference on Parallel, Distributed, and Network-Based Processing, PDP 2014, Torino, Italy, February 12-14, 2014*, pages 725–733, 2014.

[5] A. Amighi, S. Blom, M. Huisman, W. Mostowski, and M. Zaharieva-Stojanovski. Formal Specifications for Java's Synchronisation Classes. In A. L. Lafuente and E. Tuosto, editors, *22nd Euromicro International Conference on Parallel, Distributed, and Network-Based Processing*, pages 725–733. IEEE Computer Society, 2014.

[6] A. Amighi, S. Blom, M. Huisman, and M. Zaharieva-Stojanovski. The VerCors Project: Setting Up Basecamp. In *Programming Languages meets Program Verification (PLPV 2012)*, pages 71–82. ACM Press, 2012.

[7] A. Amighi, S. Darabi, S. Blom, and M. Huisman. Specification and Verification of Atomic Operations in GPGPU Programs. In *SEFM 2015*, pages 69–83, 2015.

[8] A. Amighi, P. de Carvalho Gomes, D. Gurov, and M. Huisman. Sound Control-Flow Graph Extraction for Java Programs with Exceptions. In *Software Engineering and Formal Methods - 10th International Conference, SEFM 2012, Thessaloniki, Greece, October 1-5, 2012. Proceedings*, pages 33–47, 2012.

[9] A. Amighi, P. de Carvalho Gomes, D. Gurov, and M. Huisman. Provably correct control flow graphs from Java bytecode programs with exceptions. *STTT*, 18(6):653–684, 2016.

[10] A. Amighi, C. Haack, M. Huisman, and C. Hurlin. Permission-based Separation Logic for multithreaded Java programs. *Logical Methods in Computer Science*, 11(1), 2015.

[11] A. Amighi, M. Huisman, and S. Blom. Verification of Shared-Reading Synchronisers. *SAC*, 2018. Submitted.

[12] K. R. Apt. Ten years of Hoare's logic: A survey – Part I. *ACM Trans. Program. Lang. Syst.*, 3(4):431–483, Oct. 1981.

[13] Atomic Sum in GPU: http://ctit-vm2.ewi.utwente.nl/site/example?id=38.

[14] E. Bardsley and A. Donaldson. Warps and atomics: Beyond barrier synchronization in the verification of GPU kernels. In *NASA Formal Methods*, volume 8430 of *LNCS*, pages 230–245. Springer, 2014.

[15] C. J. Bell, A. W. Appel, and D. Walker. Concurrent Separation Logic for Pipelined Parallelization. In R. Cousot and M. Martel, editors, *Static Analysis - 17th International Symposium, SAS 2010, Perpignan, France, September 14-16, 2010. Proceedings*, volume 6337 of *Lecture Notes in Computer Science*, pages 151–166. Springer, 2010.

[16] J. A. Bergstra, A. Ponse, and S. A. Smolka, editors. *Handbook of Process Algebra* . Elsevier Science, Amsterdam, 2001.

[17] A. Betts, N. Chong, A. Donaldson, S. Qadeer, and P. Thomson. GPUVerify: a verifier for GPU kernels. In *OOPSLA'12*, pages 113–132. ACM, 2012.

[18] S. Blom, S. Darabi, and M. Huisman. Verification of Loop Parallelisations. In *FASE*, pages 202–217, 2015.

[19] S. Blom and M. Huisman. The VerCors Tool for Verification of Concurrent Programs. In *FM 2014: Formal Methods - 19th International Symposium, Singapore, May 12-16, 2014. Proceedings*, pages 127–131, 2014.

[20] S. Blom, M. Huisman, and J. Kiniry. How Do Developers Use APIs? A Case Study in Concurrency, 2013. Submitted to ICECCS.

[21] S. Blom, M. Huisman, and M. Mihelčić. Specification and verification of GPGPU programs . *Science of Computer Programming*, 95(3):376 – 388, 2014.

[22] S. Blom, M. Huisman, and M. Zaharieva-Stojanovski. History-Based Verification of Functional Behaviour of Concurrent Programs. In R. Calinescu and B. Rumpe, editors, *Software Engineering and Formal Methods - 13th International Conference, SEFM 2015, York, UK, September 7-11, 2015. Proceedings*, volume 9276 of *Lecture Notes in Computer Science*, pages 84–98. Springer, 2015.

[23] R. Bornat, C. Calcagno, P. O'Hearn, and M. Parkinson. Permission accounting in separation logic. In J. Palsberg and M. Abadi, editors, *POPL*, pages 259–270. ACM, 2005.

[24] J. Boyland. Checking interference with fractional permissions. In R. Cousot, editor, *Static Analysis Symposium*, volume 2694 of *LNCS*, pages 55–72. Springer, 2003.

[25] L. Burdy, Y. Cheon, D. Cok, M. Ernst, J. Kiniry, G. Leavens, K. Leino, and E. Poll. An overview of JML tools and applications. *STTT*, 7(3):212–232, 2005.

[26] C. Calcagno, D. Distefano, J. Dubreil, D. Gabi, P. Hooimeijer, M. Luca, P. W. O'Hearn, I. Papakonstantinou, J. Purbrick, and D. Rodriguez. Moving fast with software verification. In *NASA Formal Methods - 7th International Symposium, NFM 2015, Pasadena, CA, USA, April 27-29, 2015, Proceedings*, pages 3–11, 2015.

[27] Verified ConcurrentLinkedQueue: `https://github.com/utwente-fmt/vercors/blob/master/examples/layers/LFQ.java`.

[28] Verified CountDownLatch: https://github.com/utwente-fmt/vercors/blob/master/examples/synchronizers/CountDownLatch.java.

[29] da Rocha Pinto, Pedro, Dinsdale-Young, Thomas, and P. Gardner. TaDA: A logic for time and data abstraction. In R. Jones, editor, *ECOOP 2014 - Object-Oriented Programming*, volume 8586 of *LNCS*, pages 207–231. Springer, 2014.

[30] P. da Rocha Pinto, T. Dinsdale-Young, and P. Gardner. Steps in Modular Specifications for Concurrent Modules (Invited Tutorial Paper). *Electr. Notes Theor. Comput. Sci.*, 319:3–18, 2015.

[31] Data race vs. race condition: https://blog.regehr.org/archives/490.

[32] T. Dinsdale-Young, P. da Rocha Pinto, K. J. Andersen, and L. Birkedal. Caper - Automatic Verification for Fine-Grained Concurrency. In *Programming Languages and Systems - 26th European Symposium on Programming, ESOP 2017, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2017, Uppsala, Sweden, April 22-29, 2017, Proceedings*, pages 420–447, 2017.

[33] T. Dinsdale-Young, M. Dodds, P. Gardner, M. Parkinson, and V. Vafeiadis. Concurrent abstract predicates. In *ECOOP*, pages 504–528, 2010.

[34] M. Dodds, X. Feng, M. J. Parkinson, and V. Vafeiadis. Deny-guarantee reasoning. In *Programming Languages and Systems, 18th European Symposium on Programming, ESOP 2009, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2009, York, UK, March 22-29, 2009. Proceedings*, pages 363–377, 2009.

[35] M. Dodds, S. Jagannathan, and M. J. Parkinson. Modular reasoning for deterministic parallelism. In *Proceedings of the 38th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '11, pages 259–270, New York, NY, USA, 2011. ACM.

[36] M. Dodds, S. Jagannathan, M. J. Parkinson, K. Svendsen, and L. Birkedal. Verifying custom synchronization constructs using higher-order

separation logic. *ACM Trans. Program. Lang. Syst.*, 38(2):4:1–4:72, 2016.

[37] N. Francez and A. Pnueli. A proof method for cyclic programs. *Acta Inf.*, 9:133–157, 1978.

[38] A. Gotsman, J. Berdine, and B. Cook. Interprocedural Shape Analysis with Separated Heap Abstractions. In K. Yi, editor, *Static Analysis, 13th International Symposium, SAS 2006, Seoul, Korea, August 29-31, 2006, Proceedings*, volume 4134 of *Lecture Notes in Computer Science*, pages 240–260. Springer, 2006.

[39] A. Gotsman, J. Berdine, and B. Cook. Precision and the Conjunction Rule in Concurrent Separation Logic. *Electr. Notes Theor. Comput. Sci.*, 276:171–190, 2011.

[40] A. Gotsman, J. Berdine, B. Cook, N. Rinetzky, and M. Sagiv. Local reasoning for storable locks and threads. In *Proceedings of the 5th Asian conference on Programming languages and systems*, APLAS'07, pages 19–37. Springer-Verlag, 2007.

[41] C. Haack, M. Huisman, and C. Hurlin. Reasoning about Java's reentrant locks. In G. Ramalingam, editor, *Asian Programming Languages and Systems Symposium*, volume 5356 of *LNCS*, pages 171–187. Springer, 2008.

[42] C. Haack and C. Hurlin. Separation logic contracts for a Java-like language with fork/join. In J. Meseguer and G. Rosu, editors, *Algebraic Methodology and Software Technology*, volume 5140 of *LNCS*, pages 199–215. Springer, 2008.

[43] C. Hoare. An axiomatic basis for computer programming. *Communications of the ACM*, 12(10):576–580, 1969.

[44] C. Hoare. Proof of correctness of data representations. *Acta Informatica*, 1(4):271–281, 1972.

[45] A. Hobor and C. Gherghina. Barriers in Concurrent Separation Logic. In *20th European Symposium of Programming (ESOP 2011)*, LNCS, pages 276–296. Springer, 2011.

[46] C. Hurlin. *Specification and Verification of Multithreaded Object-Oriented Programs with Separation Logic*. PhD thesis, Université Nice Sophia Antipolis, 2009.

[47] B. Jacobs and F. Piessens. Modular full functional specification and verification of lock-free data structures. CW Reports CW551, Department of Computer Science, K.U.Leuven, 2009.

[48] B. Jacobs and F. Piessens. Expressive modular fine-grained concurrency specification. In *Proceedings of the 38th annual ACM SIGPLAN-SIGACT*, POPL '11, pages 271–282, New York, NY, USA, 2011. ACM.

[49] B. Jacobs, J. Smans, P. Philippaerts, F. Vogels, W. Penninckx, and F. Piessens. Verifast: A powerful, sound, predictable, fast verifier for C and Java. In *NASA Formal Methods - Third International Symposium, NFM 2011, Pasadena, CA, USA, April 18-20, 2011. Proceedings*, pages 41–55, 2011.

[50] C. B. Jones. Specification and design of (parallel) programs. In *IFIP Congress*, pages 321–332, 1983.

[51] C. B. Jones. Tentative steps toward a development method for interfering programs. *ACM Transactions on Programming Languages and Systems*, 5(4):596–619, 1983.

[52] C. B. Jones. The role of auxiliary variables in the formal development of concurrent programs. In A. Roscoe, C. B. Jones, and K. R. Wood, editors, *Reflections on the Work of C.A.R. Hoare*, pages 167–187. Springer London, 2010.

[53] C. B. Jones, I. J. Hayes, and R. J. Colvin. Balancing expressiveness in formal approaches to concurrency. *Formal Aspects of Computing*, 27(3):475–497, 2015.

[54] U. Juhasz, I. T. Kassios, P. Müller, M. Novacek, M. Schwerhoff, and A. J. Summers. Viper: A verification infrastructure for permission-based reasoning. Technical report, ETH Zurich, 2014.

[55] R. Jung, D. Swasey, F. Sieczkowski, K. Svendsen, A. Turon, L. Birkedal, and D. Dreyer. Iris: Monoids and Invariants as an Orthogonal Basis for Concurrent Reasoning. In *Proceedings of the 42nd Annual*

ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2015, Mumbai, India, January 15-17, 2015, pages 637–650, 2015.

[56] A. Laarman, J. van de Pol, and M. Weber. Boosting multi-core reachability performance with shared hash tables. In R. Bloem and N. Sharygina, editors, *FMCAD*, pages 247–255. IEEE, 2010.

[57] D. Lea. A Java fork/join framework. In *Proceedings of the ACM 2000 conference on Java Grande*, JAVA '00, pages 36–43, New York, NY, USA, 2000. ACM.

[58] K. Leino, P. Müller, and J. Smans. Verification of concurrent programs with Chalice. In *Lecture notes of FOSAD*, volume 5705 of *LNCS*. Springer, 2009.

[59] R. Ley-Wild and A. Nanevski. Subjective auxiliary state for coarse-grained concurrency. In R. Giacobazzi and R. Cousot, editors, *POPL*, pages 561–574. ACM, 2013.

[60] G. Li and G. Gopalakrishnan. Scalable SMT-based verification of GPU kernel functions. In *SIGSOFT FSE 2010*, pages 187–196. ACM, 2010.

[61] Verified Lock: `https://github.com/utwente-fmt/vercors/blob/master/examples/synchronizers/ReentrantLock.java`.

[62] B. Meyer. Applying "design by contract". *Computer*, 25(10):40–51, Oct. 1992.

[63] M. M. Michael and M. L. Scott. Simple, fast, and practical non-blocking and blocking concurrent queue algorithms. In *Proceedings of the fifteenth annual ACM symposium on Principles of distributed computing*, PODC '96, pages 267–275, New York, NY, USA, 1996. ACM.

[64] G. Moore. Cramming more components onto integrated circuits. *Proceedings of the IEEE*, 86(1):82–85, Jan 1998.

[65] P. Müller. *Modular Specification and Verification of Object-Oriented Programs*. PhD thesis, FernUniversität Hagen, 2001.

[66] P. Müller, M. Schwerhoff, and A. J. Summers. Viper: A verification infrastructure for permission-based reasoning. In *Verification, Model Checking, and Abstract Interpretation - 17th International Conference, VMCAI 2016, St. Petersburg, FL, USA, January 17-19, 2016. Proceedings*, pages 41–62, 2016.

[67] NVIDIA Corporation. CUDA C programming guide, version 5.5, 2013.

[68] P. W. O'Hearn. Resources, concurrency and local reasoning. *Theoretical Computer Science*, 375(1–3):271–307, 2007.

[69] W. Oortwijn, S. Blom, and M. Huisman. Future-based Static Analysis of Message Passing Programs. In D. A. Orchard and N. Yoshida, editors, *Proceedings of the Ninth workshop on Programming Language Approaches to Concurrency- and Communication-cEntric Software, PLACES 2016, Eindhoven, The Netherlands, 8th April 2016.*, volume 211 of *EPTCS*, pages 65–72, 2016.

[70] S. Owicki and D. Gries. An axiomatic proof technique for parallel programs. *Acta Informatica Journal*, 6:319–340, 1975.

[71] M. Parkinson and G. Bierman. Separation logic, abstraction and inheritance. In *Principles of programming languages (POPL '08)*, pages 75–86. ACM Press, 2008.

[72] M. J. Parkinson. Local reasoning for Java. Technical Report UCAM-CL-TR-654, University of Cambridge, Computer Laboratory, Nov. 2005.

[73] W. Penninckx, B. Jacobs, and F. Piessens. Sound, Modular and Compositional Verification of the Input/Output Behavior of Programs. In J. Vitek, editor, *Programming Languages and Systems - 24th European Symposium on Programming, ESOP 2015, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2015, London, UK, April 11-18, 2015. Proceedings*, volume 9032 of *Lecture Notes in Computer Science*, pages 158–182. Springer, 2015.

[74] PVL: https://github.com/utwente-fmt/vercors/wiki/Prototypal-Verification-Language.

[75] M. Raynal. *Concurrent Programming - Algorithms, Principles, and Foundations*. Springer, 2013.

[76] J. Reynolds. Separation Logic: A Logic for Shared Mutable Data Structures. In *Logic in Computer Science*, pages 55–74. IEEE Computer Society, 2002.

[77] Verified Semaphore: https://github.com/utwente-fmt/vercors/blob/master/examples/synchronizers/Semaphore.java.

[78] J. Smans, B. Jacobs, and F. Piessens. Implicit dynamic frames. *ACM Trans. Program. Lang. Syst.*, 34(1):2:1–2:58, 2012.

[79] J. Smans, D. Vanoverberghe, D. Devriese, B. Jacobs, and F. Piessens. Shared boxes: rely-guarantee reasoning in verifast. CW Reports CW662, Department of Computer Science, KU Leuven, May 2014.

[80] J. A. Stratton, C. Rodrigues, I.-J. Sung, N. Obeid, L.-W. Chang, N. Anssari, G. D. Liu, and W.-m. Hwu. Parboil: A revised benchmark suite for scientific and commercial throughput computing. *Center for Reliable and High-Performance Computing*, 2012.

[81] K. Svendsen and L. Birkedal. Impredicative Concurrent Abstract Predicates. In *Programming Languages and Systems - 23rd European Symposium on Programming, ESOP 2014, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2014, Grenoble, France, April 5-13, 2014, Proceedings*, pages 149–168, 2014.

[82] K. Svendsen, L. Birkedal, and M. J. Parkinson. Modular reasoning about separation of concurrent data structures. In *ESOP*, pages 169–188, 2013.

[83] E. Tempero, C. Anslow, J. Dietrich, T. Han, J. Li, M. Lumpe, H. Melton, and J. Noble. Qualitas corpus: A curated collection of Java code for empirical studies. In *2010 Asia Pacific Software Engineering Conference (APSEC2010)*, Dec. 2010.

[84] A. Turon, D. Dreyer, and L. Birkedal. Unifying refinement and hoare-style reasoning in a logic for higher-order concurrency. In *ACM SIGPLAN International Conference on Functional Programming, ICFP'13, Boston, MA, USA - September 25 - 27, 2013*, pages 377–390, 2013.

[85] V. Vafeiadis. Concurrent separation logic and operational semantics. *Electr. Notes Theor. Comput. Sci.*, 276:335–351, 2011.

[86] V. Vafeiadis and M. Parkinson. A marriage of rely/guarantee and separation logic. In *CONCUR*, pages 256–271, 2007.

[87] Vercors tool set: http://ctit-vm3.ewi.utwente.nl/vercors-verifier/.

[88] H. Yang. *Local Reasoning for Stateful Programs*. PhD thesis, University of Illinois,Urbana-Champaign,Illinois, 2001.

Sumenvatting

Digitale services zijn een onmisbaar onderdeel van ons dagelijks leven geworden. Om deze services te kunnen leveren, is het belangrijk om efficiënte software te kunnen maken. *Concurrency* is een programmeertechniek die software-ontwikkelaars kunnen gebruiken om de snelheid van hun software te verbeteren. In een concurrent programma worden meerdere berekeningen tegelijkertijd uitgevoerd. Deze berekeningen hebben soms tegelijkertijd toegang nodig tot dezelfde geheugenlocaties, wat tot onverwachte fouten kan leiden. Programmeurs gebruiken daarom synchronisatietechnieken om de concurrency aan banden te leggen, en toegang tot dit gedeelde geheugen te coördineren. Om betrouwbare concurrent software te ontwikkelen is de correctheid van deze synchronisatietechnieken dus cruciaal.

In dit proefschrift gebruiken we een programmalogica, namelijk permissie-gebaseerde separatielogica, om op een statische manier te kunnen redeneren over de correctheid van synchronisatietechnieken. De gebruikte logica is krachtig genoeg om te redeneren over welke berekening toegang heeft tot een bepaalde geheugenlocatie. Een correct geïmplementeerde synchronisatietechniek zorgt er voor dat de toegang tot een gedeelde geheugenlocatie op een juiste manier wordt verdeeld. We gebruiken onze VerCors verificatietool om de correctheid van verschillende synchronisatietechnieken te demonstreren.

In Hoofdstuk 1 beschrijven we de context van het werk in dit proefschrift. Alle benodigde technische achtergrondkennis over permissie-gebaseerde separatielogica en programmasynchronisatie worden toegelicht in Hoofdstuk 2. In Hoofdstuk 3 bespreken we hoe het beginnen en samenvoegen van berekeningen gezien kan worden als synchronisatie tussen verschillende berekeningen en hoe dit gebruikt kan worden om correctheid van programma's te verifiëren.

Om correctheid van de verschillende synchronisatieklassen te laten zien, moeten we eerst specifiëren wat het verwachte gedrag is van deze klassen. Dit wordt besproken in Hoofdstuk 4. We beschrijven een uniforme aanpak om het gedrag van verschillende synchronisatietechnieken te beschrijven. Met behulp van onze specificaties is het mogelijk om te redeneren over de correctheid van programma's die de verschillende synchronisatietechnieken gebruiken om toegang tot het gedeelde geheugen te coördineren.

De *atomic* klassen van `java.util.concurrent` zijn een basisingrediënt voor elke implementatie van een synchronisatietechniek. In Hoofdstukken 5 en 6

beschrijven we een specificatie voor deze klassen, die gebruikt kan worden om te laten zien dat de implementatie van verschillende synchronisatietechnieken voldoet aan de specificaties zoals beschreven in Hoofdstuk 4. Deze specificatie is geparametriseerd met een *protocol* en een *resource invariant* en de instantiatie hiervan is afhankelijk van de context waarin de synchronisatietechniek gebruikt wordt.

In Hoofdstuk 7 beschrijven we hoe verificaties gestapeld kunnen worden, waarbij op elke laag een specifiek aspect van het te verifiëren programma gecontroleerd wordt. We laten in het bijzonder zien hoe dit gebruikt wordt voor het verifiëren van programma's waarin atomaire operaties de belangrijkste synchronisatietechniek zijn. Als concreet voorbeeld laten we zien hoe we op het laagste niveau kunnen laten zien dat een niet-blokkerende datastructuurimplementatie geen data races heeft, zodat we dit kunnen gebruiken om op een hoger niveau functionele eigenschappen van de datastructuur correct te bewijzen.

In Hoofdstuk 8 bespreken we een specificatie- en verificatietechniek om te redeneren over afwezigheid van data races en functionele correctheid van GPU kernels die atomaire operaties gebruiken als synchronisatiemechanisme.

Tenslotte eindigen we het proefschrift in Hoofdstuk 9 met ideeën voor vervolgonderzoek.

Curriculum Vitae

Afshin Amighi received his BSc degree in Software Engineering from Isfahan University of Technology, Isfahan, Iran. After receiving his bachelor's degree in 2000, he started working at the Information and Communication Technology Institute (ICTI) located in Isfahan University of Technology, as a software engineer for eight years. On September 2008, he moved to Stockholm, Sweden, where he got admitted to study in an international MSc program with specialization in Software Engineering for Distributed Systems at KTH. In 2010, he participated in a live streaming P2P project at Tradix AB, Stockholm.

On March 2011, he joined to the Formal Methods and Tools group at the EEMCS Department of University of Twente, The Netherlands as a Ph.D. candidate. His research was funded by an ERC grant under project VerCors : Verification of Concurrent Data Structures. His research led to a number of publications and this thesis. Afshin is currently a Computer Science lecturer in Rotterdam University of Applied Science.

Titles in the IPA Dissertation Series since 2015

G. Alpár. *Attribute-Based Identity Management: Bridging the Cryptographic Design of ABCs with the Real World*. Faculty of Science, Mathematics and Computer Science, RU. 2015-01

A.J. van der Ploeg. *Efficient Abstractions for Visualization and Interaction*. Faculty of Science, UvA. 2015-02

R.J.M. Theunissen. *Supervisory Control in Health Care Systems*. Faculty of Mechanical Engineering, TU/e. 2015-03

T.V. Bui. *A Software Architecture for Body Area Sensor Networks: Flexibility and Trustworthiness*. Faculty of Mathematics and Computer Science, TU/e. 2015-04

A. Guzzi. *Supporting Developers' Teamwork from within the IDE*. Faculty of Electrical Engineering, Mathematics, and Computer Science, TUD. 2015-05

T. Espinha. *Web Service Growing Pains: Understanding Services and Their Clients*. Faculty of Electrical Engineering, Mathematics, and Computer Science, TUD. 2015-06

S. Dietzel. *Resilient In-network Aggregation for Vehicular Networks*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2015-07

E. Costante. *Privacy throughout the Data Cycle*. Faculty of Mathematics and Computer Science, TU/e. 2015-08

S. Cranen. *Getting the point — Obtaining and understanding fixpoints in model checking*. Faculty of Mathematics and Computer Science, TU/e. 2015-09

R. Verdult. *The (in)security of proprietary cryptography*. Faculty of Science, Mathematics and Computer Science, RU. 2015-10

J.E.J. de Ruiter. *Lessons learned in the analysis of the EMV and TLS security protocols*. Faculty of Science, Mathematics and Computer Science, RU. 2015-11

Y. Dajsuren. *On the Design of an Architecture Framework and Quality Evaluation for Automotive Software Systems*. Faculty of Mathematics and Computer Science, TU/e. 2015-12

J. Bransen. *On the Incremental Evaluation of Higher-Order Attribute Grammars*. Faculty of Science, UU. 2015-13

S. Picek. *Applications of Evolutionary Computation to Cryptology*.

Faculty of Science, Mathematics and Computer Science, RU. 2015-14

C. Chen. *Automated Fault Localization for Service-Oriented Software Systems*. Faculty of Electrical Engineering, Mathematics, and Computer Science, TUD. 2015-15

S. te Brinke. *Developing Energy-Aware Software*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2015-16

R.W.J. Kersten. *Software Analysis Methods for Resource-Sensitive Systems*. Faculty of Science, Mathematics and Computer Science, RU. 2015-17

J.C. Rot. *Enhanced coinduction*. Faculty of Mathematics and Natural Sciences, UL. 2015-18

M. Stolikj. *Building Blocks for the Internet of Things*. Faculty of Mathematics and Computer Science, TU/e. 2015-19

D. Gebler. *Robust SOS Specifications of Probabilistic Processes*. Faculty of Sciences, Department of Computer Science, VUA. 2015-20

M. Zaharieva-Stojanovski. *Closer to Reliable Software: Verifying functional behaviour of concurrent programs*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2015-21

R.J. Krebbers. *The C standard formalized in Coq*. Faculty of Science, Mathematics and Computer Science, RU. 2015-22

R. van Vliet. *DNA Expressions – A Formal Notation for DNA*. Faculty of Mathematics and Natural Sciences, UL. 2015-23

S.-S.T.Q. Jongmans. *Automata-Theoretic Protocol Programming*. Faculty of Mathematics and Natural Sciences, UL. 2016-01

S.J.C. Joosten. *Verification of Interconnects*. Faculty of Mathematics and Computer Science, TU/e. 2016-02

M.W. Gazda. *Fixpoint Logic, Games, and Relations of Consequence*. Faculty of Mathematics and Computer Science, TU/e. 2016-03

S. Keshishzadeh. *Formal Analysis and Verification of Embedded Systems for Healthcare*. Faculty of Mathematics and Computer Science, TU/e. 2016-04

P.M. Heck. *Quality of Just-in-Time Requirements: Just-Enough and Just-in-Time*. Faculty of Electrical Engineering, Mathematics, and Computer Science, TUD. 2016-05

Y. Luo. *From Conceptual Models to Safety Assurance – Applying Model-Based Techniques to Support Safety Assurance*. Faculty of

Mathematics and Computer Science, TU/e. 2016-06

B. Ege. *Physical Security Analysis of Embedded Devices.* Faculty of Science, Mathematics and Computer Science, RU. 2016-07

A.I. van Goethem. *Algorithms for Curved Schematization.* Faculty of Mathematics and Computer Science, TU/e. 2016-08

T. van Dijk. *Sylvan: Multi-core Decision Diagrams.* Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2016-09

I. David. *Run-time resource management for component-based systems.* Faculty of Mathematics and Computer Science, TU/e. 2016-10

A.C. van Hulst. *Control Synthesis using Modal Logic and Partial Bisimilarity – A Treatise Supported by Computer Verified Proofs.* Faculty of Mechanical Engineering, TU/e. 2016-11

A. Zawedde. *Modeling the Dynamics of Requirements Process Improvement.* Faculty of Mathematics and Computer Science, TU/e. 2016-12

F.M.J. van den Broek. *Mobile Communication Security.* Faculty of Science, Mathematics and Computer Science, RU. 2016-13

J.N. van Rijn. *Massively Collaborative Machine Learning.* Faculty of Mathematics and Natural Sciences, UL. 2016-14

M.J. Steindorfer. *Efficient Immutable Collections.* Faculty of Science, UvA. 2017-01

W. Ahmad. *Green Computing: Efficient Energy Management of Multiprocessor Streaming Applications via Model Checking.* Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2017-02

D. Guck. *Reliable Systems – Fault tree analysis via Markov reward automata.* Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2017-03

H.L. Salunkhe. *Modeling and Buffer Analysis of Real-time Streaming Radio Applications Scheduled on Heterogeneous Multiprocessors.* Faculty of Mathematics and Computer Science, TU/e. 2017-04

A. Krasnova. *Smart invaders of private matters: Privacy of communication on the Internet and in the Internet of Things (IoT).* Faculty of Science, Mathematics and Computer Science, RU. 2017-05

A.D. Mehrabi. *Data Structures for Analyzing Geometric Data.* Faculty of Mathematics and Computer Science, TU/e. 2017-06

D. Landman. *Reverse Engineering Source Code: Empirical Studies of Limitations and Opportunities*. Faculty of Science, UvA. 2017-07

W. Lueks. *Security and Privacy via Cryptography – Having your cake and eating it too*. Faculty of Science, Mathematics and Computer Science, RU. 2017-08

A.M. Şutîi. *Modularity and Reuse of Domain-Specific Languages: an exploration with MetaMod*. Faculty of Mathematics and Computer Science, TU/e. 2017-09

U. Tikhonova. *Engineering the Dynamic Semantics of Domain Specific Languages*. Faculty of Mathematics and Computer Science, TU/e. 2017-10

Q.W. Bouts. *Geographic Graph Construction and Visualization*. Faculty of Mathematics and Computer Science, TU/e. 2017-11

A. Amighi. *Specification and Verification of Synchronisation Classes in Java: A Practical Approach*. Faculty of Electrical Engineering, Mathematics & Computer Science, UT. 2018-01