If software code is developed by humans, can we as users rely on its absolute correctness?

Today's software is large, complex, and prone to errors. Although many bugs are found in the process of testing, we can never claim that the delivered software is bug-free. Errors still occur when software is in use; and errors exist that will perhaps never occur. Reaching an absolute zero bug state for usable software is practically impossible.

On the other side we have mathematical logic, a very powerful machinery for reasoning and drawing conclusions based on facts. The power of mathematical logic is certainty: when a given statement is mathematically proven, it is indeed absolutely correct.

When a technique for verifying software is based on logic, it allows one to mathematically prove properties about the program. These so-called formal verification techniques are very challenging to develop, but what they promise is highly valuable, and so, they certainly deserve close research attention. This thesis shows the benefits and drawbacks of this style of reasoning, and proposes novel techniques that respond to some important verification challenges.

Still, mathematical logic is theory, and software is practice. Thus, formal verification can not guarantee absolute correctness of software, but it certainly has the potential to move us much closer to reliable software.

CLOSER TO RELIABLE SOFTWARE

Marina Zaharieva Stojanovski ■ 2015

# CLOSER TO RELIABLE SOFTWARE

## VERIFYING FUNCTIONAL BEHAVIOUR OF CONCURRENT PROGRAMS

Marina Zaharieva Stojanovski

# Closer to Reliable Software

*Verifying functional behaviour of concurrent programs*

Marina Zaharieva Stojanovski

Graduation committee:

| | | |
|---|---|---|
| Chairman: | prof.dr. Peter M.G. Apers | University of Twente |
| Promotor: | prof.dr. Marieke Huisman | University of Twente |
| Referee: | dr. Bart Jacobs | University of Leuven |
| Members: | prof.dr.ir. Arend Rensink | University of Twente |
| | dr. Job Zwiers | University of Twente |
| | prof.dr. Einar Broch Johnson | University of Oslo |
| | prof.dr. Philippa Gardner | Imperial College London |


**CTIT Ph.D. Thesis Series No. 15-375**
Centre for Telematics and Information Technology
University of Twente, The Netherlands
P.O. Box 217 – 7500 AE Enschede


**IPA Dissertation Series No. 2015-21**
The work in this thesis has been carried out under the auspices of the research school IPA (Institute for Programming research and Algorithmics).


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

ISBN 978-90-365-3924-1
ISSN 1381-3617 (CTIT Ph.D. Thesis Series No. 15-375)
Available online at `http://dx.doi.org/10.3990/1.9789036539241`

Typeset with LaTeXPrinted by Gildeprint
Cover design by Marina Zaharieva Stojanovski

# CLOSER TO RELIABLE SOFTWARE

VERIFYING FUNCTIONAL BEHAVIOUR OF CONCURRENT PROGRAMS

DISSERTATION

to obtain
the degree of doctor at the University of Twente,
on the authority of the rector magnificus,
prof.dr. H. Brinksma,
on account of the decision of the graduation committee,
to be publicly defended
on Thursday, October 1st, 2015 at 12:45 hrs.

by

Marina Zaharieva Stojanovski

born on 06 August 1985
in Kocani, Macedonia

This dissertation has been approved by:

Supervisor: prof.dr. Marieke Huisman

# Acknowledgements

The path to finishing this dissertation seems obvious now. But when I look back four years ago, when my ambitions were far greater than my research skills, choosing the right direction in the unknown seemed almost impossible. I am grateful to both Marieke Huisman and Jaco van de Pol for entrusting me with the responsibility to be part of the FMT group. I was happy to have encouraging people around me; I was inspired by them, I was learning from them, I was following their examples until I found my own way. All these people were my essential drive and support during the development of this dissertation.

Marieke, thank you for being my truly inspiring mentor. You always had a clear vision and you were here to advise me how to go towards this vision. You had trust in me, giving me space for independence and showing me the way how to lead my work. I was always admiring your courage to choose the most challenging goals. Now I see that exactly these brave choices gave me the confidence to see big challenges as something achievable. I am thankful for your support in my daily work. Every discussion that we had was a much-needed boost for me to continue. For all research skills that I have now, I am thankful to you and your always valuable feedback. The existence of this thesis is a proof for your full support in the last four years.

Stefan Blom, this thesis would not have been what it is without our long discussions. You are one of the few people I know who can grasp very complex things so easily, and can spot whether all small details will fit together. I was sometimes wondering how were you able to understand my confusing questions, on which you always had an answer and a suggestion for a further improvement. Thank you for all your support.

Dilian Gurov, thank you for showing interest in my short presentation in Leiden. Our chat there progressed in an efficient collaboration and a nice piece of work. I am thankful for your valuable help, for your constructive feedback and stimulating questions, which were crucial for improvement of our work.

Afshin Amighi and Wojciech Mostowski, you have a substantial contribution in this thesis. Thank you for all our useful discussions, for your advices and your always generous help and support.

Arend Rensink, thank you for the nice collaboration within the Advanced Logic course. You showed me useful insights in both the meaning of logic and good teaching. This was undoubtedly a welcome boost for my further work.

I would like to thank all members of the committee for their willingness to read the thesis: Philippa Gardner, Bart Jacobs, Einar Broch Johnson, Arend Rensink and Job Zwiers. I am also grateful to Europian Research Council (ERC), who funded this work via the VerCors project.

For all FMT members, thank you for being there and for being who you are: smart, wise, successful, passionate about science, talented, modest, positive, always respectful, friendly and generous. Thank you for all cosy chats and enjoyable moments, for being great colleagues and great friends. Stefano and Saeed, thank you for accepting the challenge to be my paranymphs on my defence.

I am grateful to my family and friends for their understanding and encouragement. I thank my mother Zora for giving me the love for mathematics, my father Ljupco for teaching me about diligence and endurance, and my brother Dragan for guiding me to keep a positive and practical view on all challenges in life.

Finally, I thank my husband Spase, for his unconditional support in achieving my dreams, and for always being here to toast with me all my failures and successes. Thanks!

York, UK
September 12, 2015

# Abstract

*Static formal verification* techniques are an effective method for verification of software. They exploit the advantages of formal methods to statically prove that the implementation of a program satisfies its formally written specification. This makes formal verification especially powerful: any execution of the program is guaranteed to behave correctly. Therefore, these techniques are especially attractive for safety-critical systems, where correctness of the code is a crucial requirement.

Applying formal techniques for verification of concurrent software is appealing. First, concurrent software today is omnipresent, but it is especially prone to errors. Second, finding errors in concurrent software using standard dynamic testing techniques is difficult, because of the non-deterministic behaviour of this software. Unfortunately, formal verification of concurrent software is hard and faces many challenges.

This thesis contributes with novel formal techniques for verification of multithreaded programs. We focus mainly on verification of *functional properties*, i.e., properties that describe the behaviour of the program. Concretely, we work with *axiomatic reasoning* and use *permission-based separation logic* as our basic program logic.

First, we propose a new modular technique for verification of *class invariants* in concurrent programs. This technique allows breaking of class invariants at certain safe places in the program. The technique is flexible and permissive, and thus, can be applied in a broad range of practical examples. This approach is formalised on a concurrent object-oriented language.

Second, we propose a new way of specifying and verifying functional behaviour of methods in the program. Our technique uses separation logic-based reasoning to build an abstraction of the program represented as a *process algebra term*; by reasoning about the abstract model, we prove properties about the original program. This approach allows very expressive and intuitive specifications.

It is formalised for a concurrent object-oriented language, and integrated into our verification tool *VerCors*.

Third, we propose how by using history-based reasoning, one can reason about concurrent programs with *guarded blocks*. Our technique allows proving both functional and *non-blocking* properties about these programs. Moreover, we develop also a reverse *future-based* reasoning technique that allows verification of programs with non-terminating threads. We formalise this method on a simplified procedural language.

Permission-based separation logic is a well-established and powerful logic: it ties values of shared locations with *permissions* to these locations, which is an effective way to guarantee *data race-freedom*. However, it seems that this approach is not very convenient for modular verification of functional properties.

What is common for our techniques is that we use permission-based separation logic as a basic logic to ensure data race-freedom. However, we modify this logic and allow separation between values, i.e., functional properties, and permissions. It shows that this separation is useful and can significantly increase the number of properties that we can prove about concurrent programs.

# Table of Contents

# Chapter 1

# Introduction

*If software code is developed by humans, can we as users rely on its absolute correctness?*

TODAY'S software is large, complex, and prone to errors. Moreover, the presence of concurrent code, which today is inevitable, significantly increases the number of defects in the program. While some of these bugs are too small to have any visible effect, others may cause severe problems. To improve the quality of software, standard testing techniques are normally integrated into the software verification process. Although many bugs are found in the process of testing, we can never claim that the delivered software is bug-free. Errors still occur when software is in use; and errors exist that will perhaps never occur. Reaching absolute zero bug state for a usable software is practically impossible.

On the other side we have *mathematical logic*, a very powerful machinery for reasoning and drawing conclusions based on facts. The power of mathematical logic is *certainty*: when a given statement is mathematically proven, it is indeed absolutely correct.

When a technique for verifying software is based on logic, it allows one to mathematically prove properties about the program. These so-called *formal verification techniques* are very challenging to develop, but what they promise is highly valuable, and so, they certainly deserve close research attention. This thesis is about *formal techniques for axiomatic reasoning about multithreaded programs*. We show the benefits and drawbacks of this style of reasoning, and propose novel techniques that respond to some important verification challenges.

Still, mathematical logic is theory, and software is practice. Therefore,

formal verification cannot guarantee absolute correctness of software, but it certainly has the potential to move us much *closer to reliable software*.

## 1.1 Concurrency is Attractive but Comes at a Price

**Concurrency is about speed** In 1965, G. Moore [Moo65] observed a rapid increase of hardware performance: *the number of transistors on a chip will double every 18 months*. This observation appeared to be accurate and stayed valid for the next decades. The result was a major boost in computer technology: sharp increase in computer speed and decrease in its cost.

As the growing power of hardware was physically reaching its limits, new computer architectures were needed to satisfy the need for speed. The solution was *concurrency*, i.e., executing multiple tasks at the same time. Therefore, manufacturers initiated a new trend: building processors composed of multiple *cores*.

However, adding multiple cores to a processor does not automatically increase a program's speed. To take full advantage of the multicore processors, developers have to parallelise their program as well: the program task should be divided into subtasks and delegated to parallel *threads*, i.e., primary logical units of execution. To this end, most of the programming languages today are designed to support *multithreaded programming*. The efficiency of the program highly depends on the design of the program itself and on the degree of parallelism of the code [Amd67].

Concurrent algorithms do not only contribute with increased speed, they also provide parallel behaviour of the program. This behaviour is independent of the number of physical processor cores. More cores usually means more speed, but even a multithreaded program running on a single-core processor gives an illusion to the user that different threads are working at the same moment. The operating system is responsible for associating each thread to a proper physical core and to switch the execution between threads. This is hidden from the user.

**Concurrency brings challenges** If 5 employees are given a task to write 10 reports, each of them is expected to finish 2 reports. Thus, in general the work will be finished 5 times faster than a single person was working. However, if these 5 employees are given the task to finish a single report document, an acceleration might be obtained, but only by smart division of the work and efficient collaboration. Employees must ensure that they do not interfere with each other, they should synchronise their changes when needed, and they should

not overwrite each other's work.

The same happens with concurrent software. The ideal form of parallel execution is *disjoint parallelism* [AdBO09]: every thread operates independently on its own resources and does not depend on the execution of any other thread. The implementation of such a concurrent program is no more complicated than a sequential program.

However, disjoint parallelism is rarely applicable in practice. Threads normally have to share memory resources and cooperate with each other. Sadly, concurrent programming with shared memory is difficult and leads to errors.

First, to write a parallel algorithm, one has to find the parallelism in the program: How to distribute tasks efficiently among threads? When can a thread start working or when does it have to communicate its result with another thread? Can a thread delegate work to other threads? It is not always trivial to design an efficient parallel algorithm and unfortunately, the general rule is that the more efficient the algorithm is, the more it is prone to errors.

Second, *thread interleavings* are also a reason for many errors in the code. Our brain is trained to think sequentially and thus, we expect that the code that we write behaves as it is written. However, when other threads are running in parallel, the instructions from a piece of code may interleave with instructions from another thread, which usually results in unexpected behaviour of our code. Moreover, due to thread interleavings, finding the error in a concurrent program is hard: the execution of a concurrent program is non-deterministic and thus, different executions of the same code may produce different results.

Furthermore, common errors in concurrent programs appear as a result of *data races*. Concretely, we say that: a *data race happens when a heap location (memory location accessible by multiple threads) is accessed at the same time by more than one thread, such that at least one of these threads is writing the location* (see [PGB+05] for details). For example, if the program allows a thread to execute the instruction $x = 4$ at the same time as another thread executes $y = x$, the program has a data race. A data race may lead to a "corrupted" program state, which results in a non-logical behaviour of the program. Note that multiple threads can read simultaneously the same shared location without causing a data race. This seems logical: in our writing reports example, it is completely safe if all employees are only reading the same report. However, when at least one of them is writing, the readers may expect to see some inconsistent (intermediate) results: moreover, if there exists another writer of the same report, both writers might interfere with each other's work.

To avoid data races, in a multithreaded program it is required that every access to the *heap* must be protected. This is achieved by using various *synchron-*

*isation* mechanisms, e.g., *locks*, *synchronisation statements (methods)*, *barriers* and *condition variables* [Lea99, PGB+05, Sch97].

While data race-freedom is a fundamental requirement for every multithreaded program, some applications additionally require that access to multiple heap locations is treated atomically. For example, if an application represents a coordinate pair $(x, y)$, where $x + y \geq 0$ should always hold, it should be safe for a thread to decrease the value of $x$ by 1 and increase the value of $y$ by 1. However, if both updates are not done in one step, the first update may possibly break the relation $x + y \geq 0$, and thereafter another thread may read $x$ and $y$ in an inconsistent state. This scenario is called a *high-level data race*. High-level data races can also be avoided by using appropriate synchronisation mechanisms.

While synchronisation is necessary to provide safe access to the shared memory, it brings its own drawbacks. Synchronisation causes threads to wait, and waiting might lead to *liveness problems*, i.e., it might happen that some of the threads in the program are blocked and remain waiting forever. A typical liveness problem is a *deadlock*, i.e., a state in which multiple threads are waiting for each other to obtain a lock, and none of them may proceed. Other examples of liveness problems are *starvation* or *livelock* in the program [PGB+05].

These are some of the most common problems that occur in concurrency. They show that concurrent programs are beneficial, but also suffer from errors. Therefore, efficient techniques for improving correctness of these programs are absolutely necessary.

## 1.2 Formal Verification is Attractive but Comes at a Price

**Verification is about correctness** The first computer programs were short and simple. The program code was verified by the developer, and it was not difficult to argue about its correctness. However, over time the size and complexity of modern software applications have significantly risen. As a result, they became prone to errors and harder to reason about.

Delivering software with bugs may bring high costs for a company: large amounts of time and money are spent in the maintenance phase. And it is not only about money, incorrect software may sometimes also affect people's *safety*. Correctness is a *crucial* requirement of any *safety-critical* computer system. Even a single error may lead to disastrous consequences.

Therefore, *verification* and *validation* V&V became an important phase in the software engineering process. Verification ensures that the program is correct, i.e., it implements the required specifications, while validation guarantees

that specifications indeed meet the client's intentions. In modern software engineering, it is required that this phase starts as early as possible, because this increases the chances to find bugs before the delivery of the software.

The verification phase employs various techniques to improve software correctness. Testing techniques for example, involve defining a set of representative test cases and executing the program to check whether it behaves according to the tests. As a simple and cheap technique, testing has been broadly accepted. However, a considerable disadvantage of this technique is that it ensures correctness of the program only within the set of test cases. Therefore, as stated by E. Dijkstra: [Dij70], " *Program testing can be used to show the presence of bugs, but never to show their absence!*".

Certain software applications require stronger verification techniques that give higher confidence in the correctness of the code. *Formal verification techniques* are potential candidates to address this requirement. Formal verification exploits the advantages of mathematical logics, which makes it especially powerful, because of its ability to guarantee the absence of bugs. An exhaustive testing of all possible program behaviours will give the same guarantee as formal verification; but this, however, is practically impossible for most applications.

**Formal verification brings challenges** Proving correctness of a program means proving that the program satisfies certain desirable properties. These can be properties like: "the program does not dereference null pointers", "the program is data race-free", "the program terminates", "the program always returns a sorted array", etc. A formal verification technique verifies the program statically, without executing the code. The property that we want to prove is expressed formally, and based on a special *program logic* that understands the semantics of the language, the verifier builds a proof for the desired property. This kind of analysis ensures that *any* execution of the code will preserve the verified property. However, developing a program logic that supports today's complex programming languages is quite challenging.

While some of the correctness properties are general requirements for every application, others are special requirements that describe the expected behaviour of a specific application. These are called *functional (behavioural)* properties. Functional properties are crucially important for every program: while for example data race-freedom ensures that all accesses to the shared memory are safe, functional properties will describe that what threads do is indeed what we want them to do.

Clearly, verifying functional properties requires help from the developer:

these properties must be manually specified. They are added to the program as annotations written in a dedicated *formal specification language*. Writing formal program specifications is time-consuming and difficult for developers. A good verification technique should therefore provide an expressive and intuitive language that can be accepted in practice. This is however rather challenging: a mathematically-oriented specification language is simpler for the verifier, but complicated for the user.

Furthermore, to make verification applicable in practice, it is important for a given verification technique to be *modular*. This means that the correctness of a single component (e.g., method or thread) can be verified in isolation; thus, a verified component is always correct independently of the environment in which it is used. Modularity complicates the process of specification: it requires every method in the program to be formally specified. Moreover, when specifying concurrent code, due to the possible interleavings with other parallel threads, it is especially difficult to describe the local behaviour of a method.

Verification techniques usually do not verify termination of the program. In other words, if the technique can verify that a property $X$ holds in a given program state, this means that: if the program reaches this state, then we are sure that $X$ holds. We say that such a technique verifies *partial correctness* of programs, while the properties that are verified are called *safety properties*: they ensure that that the program is safe and, "something bad will never happen".

Proving partial correctness together with termination of a program guarantees *total correctness* of the program. Proving termination is a very serious challenge for more complicated programs. In practice, normally only specific termination-like (liveness) properties are verified like: deadlock-freedom or absence of infinite loops in the program. Liveness properties describe whether "something good will ever happen".

Formal verification techniques alone still face challenges and limitations, especially in verifying concurrent software; thus, they require support from other verification techniques. However, they do have a high potential and extensive investigation in this area is certainly of great importance.

## 1.3 The Three Verification Challenges This Thesis Accepts

This thesis studies three important challenges in concurrent verification. We address them with novel verification techniques, while focusing on modularity and simplicity in order to make verification suitable for realistic multithreaded

programs. Our verification techniques are built on *permission-based separation logic* [Rey02, BCOP05, O'H07, AHHH14], a well-established program logic for verification of concurrent programs. The power of this logic is its ability to verify data race-freedom. Our techniques extend this logic to make it suitable for verifying other (mostly functional) properties, while the use of permission-based separation logic as a basis ensures that verified programs are free of data races.

Below we give only a very short explanation of our ideas, while a detailed presentation is provided later in Part II.

### 1.3.1 Verifying Concurrent Invariants

Invariants are part of the program specification that express properties that should continuously be preserved [LG86, Mey97]. An invariant typically expresses a relation between values of memory locations. In the example on page 4, the requirement $x + y \geq 0$ can be expressed as an invariant formula. As discussed, breaking of this relation should be possible, but only in a controlled manner. If a thread updates the values of $x$ and $y$ and therewith temporarily breaks the invariant, the inconsistent state must be *hidden (non-visible)* from the other threads; otherwise a *high-level data race* occurs.

Therefore, a technique for verifying invariants should define program states in which an invariant must hold, and states in which it is safe to break the invariant. Standard techniques for verification of invariants in sequential programs [MPHL06, LPX07] require invariants to hold only in the pre- and poststates of methods, and allow their breaking in the internal method states. However, in the presence of multiple threads, this concept is not appropriate, because a broken invariant in an internal method state of one thread may be visible for another thread.

This brings us to the first challenge that this thesis takes on:

> *Challenge 1: How to verify that a concurrent program is free of high-level data races?*

*An overview of our approach* We present our verification technique on an object-oriented language. Invariants are defined at the level of classes, and are called *class invariants*. We discuss our verification protocol and sketch the idea in Listing 1.1. An invariant `I` is specified in line 4, stating that the relation `x+y≥0` must always be preserved (we use `//@` to write specifications in the program).

```
1  class Point{
2    int x, y;
3    Lock lx = new Lock();
4    //@ Invariant I: x+y≥0;
5
6    void moveX(){
7      acquire lx;
8     // the invariant I holds
9     //@ unpack I{
10      // assume that I holds
11      x = x−1;
12     //the invariant I is maybe broken
13      y = y+1;
14      // prove that I holds
15    //@}
16     // the invariant I holds
17     release lx;
18   }
19  }
20
21
22  class Client{
23   void main(){
24     Point p = new Point (2, 3);
25     // prove that p.I holds
26     // the invariant p.I holds
27     p.move();
28     // the invariant p.I holds
29   }
30  }
```

Listing 1.1: Verification of class invariants

An invariant may be in one of the following two states: i) *stable*, i.e., a state in which it is preserved and cannot be broken; and ii) *unstable*, a state in which it does not necessarily hold. After a new object is created (line 25), every class invariant of the new object is verified; the invariant then enters a stable state. In this state we can verify other properties in the program while assuming that the invariant holds. Stability means that no thread may write to a location that is referred to by the invariant.

Breaking of a class invariant is allowed in explicitly specified code segments, called *unpacked segments*, see lines 9 and 15. Within such a segment, the invariant is in an unstable state. In this state, no thread can assume validity of the invariant. Unpacked segments can be considered as atomic blocks of code, they are properly synchronised such that any changes done within the segment are not visible outside of the segment. Before the segment is finished, the invariant must be re-established. Thus, we ensure that threads always break the invariant in a controlled way, and invariants always hold in a stable state.

We discuss and formalise our technique in Chapter 4. To allow modular verification, we adopt the restrictions of Dietl and Müller's ownership-based type system [DM05, DM12]. The technique explained here allows only a single thread to break the invariant at a time. However, in some scenarios it is possible to allow multiple threads to work individually on the same class invariant, without harming its state. For example, for the invariant in Listing 1.1, it is safe if a

thread increases x only, while a second thread increases y. When both threads join, we know that the invariant holds.

Therefore, in Section 5.7 we suggest an improvement of our verification technique from Chapter 4, such that we allow multiple threads to break the same invariant simultaneously. Each thread can break the invariant temporarily, but must promise that this breaking occurs on a safe places only. When all threads finish their updates, we can guarantee that the invariant holds. This improvement makes the approach much more permissive and applicable in practical scenarios.

### 1.3.2 History-based Verification of Functional Properties

While invariants are used to specify functional properties that are constant, to describe progress of the program we need to provide *method contracts*, i.e., a *pre-* and *postcondition* of a method that describe respectively the pre- and poststate of the method. However, in a multithreaded program, due to thread interleavings, describing the precise behaviour of a method can be a serious challenge.

For example, consider the method incr(), which increases the value of x by 1:

```
void incr(){
  acquire lx;
    x=x+1;
  release lx;
}
```

If the method was sequential, we could specify its behaviour via a method postcondition: x = \old(x)+1 (where \old(x) refers to the value of x in the prestate of the method). In a multithreaded environment, this is not an acceptable postcondition. In particular, x is protected by the lock lx and thus, outside of the synchronised segment it might have *any* value. We can always define the trivial pre-and postcondition expression, i.e., the formula **true**, but this does not provide any useful information to the client (the caller method). If for example a client initialises the value of x to 0 and then calls two parallel threads, each of them executing incr(), we want to be able to use the information from the method contract to prove that after both threads finish, the value of x is 2.

Therefore, this thesis responds to the following challenge:

*Challenge 2: How to specify and verify expressive specifications that describe the functional behaviour of multithreaded programs?*

*An overview of our approach* We propose an approach to reason about functional behaviour, based on the notion of *histories*. Concretely, a history is a *process algebra* term used to trace the behaviour of a chosen set of shared locations $L$. When the client has some initial knowledge about the values of the locations in $L$, it initialises an empty *global* history over $L$. The global history can be split into *local* histories and each split can be distributed to a different thread. One can specify the local thread behaviour in terms of *abstract actions* that are recorded in the local history. When threads join, local histories are merged into a global history, from which the possible new values of the locations in $L$ can be derived. Therefore, a local history remembers what a single thread has done, and allows one to postpone the reasoning about the current state until no thread uses the history.

Every action from the history is an instance of a predefined specification action, which has a contract only and no body. For example, to specify the incr() method, we first specify an action a, describing the update of the location x (see the code below). The behaviour of the method incr() is then specified as an extension of a local history H with the action a(1). This local history is used only by the current thread, which makes history-based specifications stable.

```
//@ precondition true;
//@ postcondition x =\old(x)+k;
action a(int k);
```

```
//@ precondition H;
//@ postcondition  H · a(1),
  void incr(){...};
```

We reason about the client code as follows. Initially, the only knowledge is x=0. After execution of both parallel threads, a history is obtained represented as a process algebra term H=a(1) ‖ a(1). We can then calculate all traces in H and conclude that the value of x is 2. Note that each trace is a sequence of actions, each with a pre- and postcondition; thus this boils down to reasoning about sequential programs.

We discuss this technique thoroughly and present its complete formalisation in Chapter 5. We use the same object-oriented language and formalisation from Chapter 4, extended with the new history-based mechanism. The technique has also been implemented in our verification tool Vercors [BH14].

### 1.3.3 Verifying Non-blocking Properties

Usually in concurrent applications, threads also need to synchronise on the value of some data. An efficient mechanism to achieve this is by using *guarded blocks* (or the *wait/notify* mechanism) [Lea99]. A guarded block is a block of code,

which can be entered by a given thread only if a certain condition, i.e., *a guard*, holds. If the guard is not satisfied, the thread has to wait until being notified by another thread. For example, the method m below contains a guarded block: a thread can enter the block (line 3) only if the value of x is different from 1.

```
   void m(){
2    synchronised (lx){
       wait (x≠1, lx);
4        x = 1;
       notifyAll(lx);
6    }
   }
```

Verification of programs with guarded blocks is difficult. First, reasoning about the functional behaviour of the code becomes more difficult, because guarded blocks restrict certain interleavings in the program. Second, guarded blocks cause liveness problems: they often bring threads into a state in which they wait forever.

What is especially challenging is that reasoning about non-blocking of the program cannot be done in isolation, because it depends on the functional behaviour of the threads: whether a thread will terminate depends on the values of certain shared locations; on the other side, the shared state is dependent on the guarded blocks, which determine the possible thread interleavings.

Therefore, this thesis accepts the following challenge:

*Challenge 3: How to verify functional and non-blocking properties of programs with guarded blocks?*

*An overview of our approach* To address this challenge, we propose a verification technique that allows verifying functional properties while considering the possible interleavings only, and moreover, that allows one to prove non-blocking of the program.

Our approach uses a history-based concept, similar to the one explained above. First, we use local reasoning to build a history, i.e., an abstract model in the form of a process algebra term, which captures the synchronisation and blocking behaviour of the program. Every action in the process algebra term represents a concrete synchronised block in the program, which contains all information needed to reason about non-blocking. Second, non-blocking of the original program is proven by proving non-blocking of the abstract model.

Alternatively, we could *predict* in the beginning of the client program the future behaviour of the program by specifying a specific abstract model, and then use local reasoning to show that the blocking behaviour of the program can indeed be abstracted by this model. In this case, we call the model a *future* because it is predicted in advance. This future-based reasoning has the advantage that it can also be used to reason about programs that contain non-terminating threads. An example of such a program is a producer that infinitely often adds a new item to a shared queue, while a consumer consumes an item infinitely often.

We present and formalise our technique on a simplified, procedural language, abstracting away all unnecessary details that are irrelevant to the approach itself. Chapter 6 presents a detailed explanation of the technique.

## 1.4 Contributions of the Thesis

In summary, this thesis contributes with novel techniques for verification of concurrent, multithreaded programs. Mainly, it studies the problem of verification of functional behaviour in concurrency. We list the following contributions.

- The thesis proposes a novel modular technique for verification of class invariants in multithreaded programs. The technique allows breaking of class invariants at safe places in the program. We provide a complete formalisation using an object-oriented language.

- The thesis proposes an idea that allows multiple parallel threads to break the same class invariant simultaneously. This makes the technique for verification of class invariants much more permissive and applicable in practice.

- The thesis proposes a novel verification technique that allows modular specifications that describe the functional behaviour of methods. The technique abstracts the behaviour of parts of the program in a model, and by reasoning about the model, we prove properties about the original program. This approach allows both expressive and intuitive specifications. The technique is formalised on an object-oriented language and implemented in our VerCors tool.

- The thesis proposes a new technique that allows reasoning about functional behaviour as well as proving non-blocking about programs with guarded blocks. It also proposes a method to reason about programs

where threads may have infinite (or non-deterministic) executions. We illustrate and formalise the approach on a simplified procedural language.

## 1.5 Outline of the Thesis

The thesis is structured as follows:

- Part I gives background on formal verification:
  - Chapter 2 describes the concepts for reasoning about sequential programs;
  - Chapter 3 studies verification of multithreaded programs, focusing mainly on permission-based separation logic.
- Part II describes our newly proposed verification techniques and compares them with existing related work:
  - Chapter 4 describes our method for verifying class invariants in multithreaded programs (see [ZSH14] for the origins of this chapter);
  - Chapter 5 presents the history-based technique for verifying functional behaviour (see [BHZS15, BHZ15] for the origins of this chapter); and
  - Chapter 6 presents the technique for reasoning about programs with guarded blocks (see [ZSBGH] for the origins of this chapter);
- Finally, in Chapter 7, we review the thesis, identify further challenges, and discuss our views about formal verification in general.

# Part I

# Background: Concepts of Verification

# Chapter 2

# Verification of Sequential Programs

Formal verification means mathematically proving that a program is correct, i.e., the implementation of the program satisfies the requirements (specification) of the program. Concretely in this thesis we are interested in *static* formal verification techniques based on *axiomatic (Hoare-style) reasoning* [Hoa69]. Such a verification technique relies on three basic components: i) *the program implementation*, i.e., the source code; ii) *the program specification*; and iii) *a program logic*.

The specification of the program describes its individual requirements; it can describe a specific property that we want to check or the entire *functional behaviour* of the program. It is written in a special mathematical language called *specification language* and is normally provided by the developer of the program. The program logic is an extension of predicate logic with rules that describe the behaviour of each construct of the programming language. The goal of the verification technique is to prove that the implementation matches the specification by deriving a proof in the given program logic.

A basic component of a specification language is the *assertion*, i.e., a boolean formula added at a certain place in the code, in which this formula is expected to hold. The idea of using assertions was introduced by Floyd [Flo67]. More precisely, the assertion placed at a given control point should hold every time an execution of the program reaches that point. An assertion is expected to describe some *functional properties* over the program state: it typically does

not describe the concrete values of variables, but rather some general relation between these values. For example, if the program receives two input values $x$ and $y$, and prints the value $r$ on the screen, which should be the sum of $x$ and $y$, an assertion can be added at the end of the program as a boolean formula $r == x + y$.

Once the program is annotated with assertions, the next step is verifying that these assertions are indeed satisfied in the required program states. This is done by analysing the code, without execution of the program and thus, it is called *static verification*. (In contrast to this, *dynamic verification* techniques [AGVY11, Kan14] involve checking the requirements of the program by executing the program.) The advantage of the static verification is that once the program is verified, we are sure that for any execution of the program and any input values, every specified assertion will be satisfied in the state in which this is required. As mentioned above, the form of static verification we use in this thesis is *axiomatic reasoning*. There do exist another approaches of static verification like *abstract interpretation* [CC14] or *symbolic model checking* [HJMS03] but these, however, are not of interest to this thesis.

**Outline** In this chapter we present the basic concepts of the verification process, and give a brief insight of the program logic that we use. We start with a discussion of the fundamental concepts of the axiomatic reasoning about simple imperative programs in Section 2.1: here we present the basic rules in the logic and show how to derive a proof in this logic to verify correctness of a simple program. Furthermore, in Section 2.1.2 we show the ideas of reasoning about a program in a modular way. In Section 2.2 we move to object-oriented sequential programs, discussing the challenges present in verification of these programs. In Section 2.3 we discuss more extensively the features of a real specification language, and in Section 2.4 we conclude.

## 2.1 Axiomatic Reasoning about Imperative Programs

Historically speaking, the first ideas for axiomatic reasoning about programs can be attributed to R. Floyd and C. A. R. Hoare [Flo67, Hoa69]. Floyd proposed a method for reasoning about flowcharts, which later was adjusted by Hoare, to a method for reasoning about simple sequential programs. Their work attracted a great deal of attention in the subsequent years.

The axiomatic reasoning (or often called Hoare-style reasoning) suggests specifying the program in terms of its *pre-* and *postcondition*. A precondition is

```
// precondition: true
 if (a >b){
   r = a−b;
 }
 else{
   r = b−a;
 }
//postcondition:  r ≥0
```

Listing 2.1: Pre- and postcondition of a program

an assertion that we assume to be true at the beginning of the program, and the postcondition is the assertion formula that we want to be verified to hold at the end of the program. The precondition of the program might be the trivial default expression true: this indicates that we do not have any assumptions or restrictions on the initial state of the program. Later we will see that preconditions are generally used to specify the prestate of a component of the program (e.g., a method), rather than the prestate of the program itself.

For example, consider the program (let us name it $S$) in Listing 2.1, which assigns to the variable r the absolute difference of a and b. As a precondition $P$, we specify the expression true: there are no initial restrictions for the program to execute. If we are interested to verify that the value of r at the end of the program is non-negative, as a postcondition $Q$ we add the formula r≥0. Of course, we could also specify another stronger expression that describes more precisely the behaviour of the program, such as r≥0 ∧ (r==a-b ∨ r==b-a).

### 2.1.1 Hoare Logic

Verifying the program $S$ with respect to its pre- and postconditions $P$ and $Q$ means that we should prove the following: for any execution of the program statement $S$, if the precondition $P$ holds in the prestate of the execution, and if the execution terminates, the postcondition $Q$ will hold in the poststate of the execution. To express this, Hoare introduced the following triple, which was later called a *Hoare triple*:

$$\{P\}\ S\ \{Q\}$$

Note that we assume termination of the program statement $S$ and therefore, the Hoare triple expresses *partial correctness* of the statement $S$ with respect to the pre- and postconditions $P$ and $Q$ respectively.

To derive a correctness proof for the triple $\{P\}\ S\ \{Q\}$, a standard mathematical logic (such as *predicate logic*) is not sufficient: such a logic allows one to prove properties over a given *stable* state, while a state in a program is variable. Thus, to reason about programs, we additionally need rules that describe how the instructions in the program change the state of the program.

Therefore, Hoare logic is a *program logic*, a *deductive system* that extends the predicate logic with a set of *axioms* and *inference rules*, each of them describing the behaviour of a certain construct in the programming language. In particular, the rules of inference decompose triples of composed statements into triples of their substatements. An inference rule or axiom has to be introduced for every construct in the language. The soundness of these rules can be derived from the semantics of the programming language. To give the intuition behind the reasoning system, below we present a few of these rules. For more details about Hoare logic refer to [HW73, LGH$^+$78, Apt81, Apt83].

*Assignment axiom* The assignment axiom describes that: to prove that a formula $P$ holds in the postcondition of the assignment instruction, one should prove that in the prestate of the instruction the formula $P[v/x]$ holds, i.e., the formula $P$ in which every free variable $x$ is replaced by $v$.

$$\textit{[Assignment]}\ \frac{}{\{P[v/x]\}\ x = v\ \{P\}}$$

*Sequential composition rule* This rule can be understood as follows: to show that if $P$ holds in the initial state, then $Q$ holds after the sequential execution of two program statements $(S_1; S_2)$, it is sufficient to find an intermediate formula $R$ that holds after execution of $S_1$, and from which $S_2$ will establish $Q$.

$$\textit{[Sequential Composition]}\ \frac{\{P\}\ S_1\ \{R\} \qquad \{R\}\ S_2\ \{Q\}}{\{P\}\ S_1; S_2\ \{Q\}}$$

*Conditional rule* This rule states that: if $P$ holds in the prestate of the conditional construct $\mathsf{if}\ b\ \mathsf{then}\ S_1\ \mathsf{else}\ S_2$, we can conclude that $Q$ holds as a postcondition if we are able to prove both triples $\{b \wedge P\}\ S_1\ \{Q\}$ and $\{\neg b \wedge P\}\ S_2\ \{Q\}$.

$$\textit{[If-then-else]}\ \frac{\{b \wedge P\}\ S_1\ \{Q\} \qquad \{\neg b \wedge P\}\ S_2\ \{Q\} \quad \text{and } b \text{ has no side-effect}}{\{P\}\ \mathsf{if}\ b\ \mathsf{then}\ S_1\ \mathsf{else}\ S_2\ \{Q\}}$$

*Rule of consequence* The rule of consequence describes that during the reasoning, the inferred assertion statements can be simplified: a precondition $P_1$ can be replaced by a stronger formula $P$ ($P \Rightarrow P_1$), while the postcondition can be replaced by a weaker expression.

$$[Consequence]\ \frac{P \Rightarrow P_1 \qquad \{P_1\}\ S\ \{Q_1\} \qquad Q_1 \Rightarrow Q}{\{P\}\ S\ \{Q\}}$$

*While-loop rule* Reasoning about loops in a program is more challenging, because it requires help from the user. In particular, the user needs to explicitly specify a predicate for every loop that is expected to hold during the execution of the loop. This predicate is called *loop invariant*. Recently, different techniques have been developed that deal with automated generation of loop invariants [Wei11, LL05, RK07]. The *while-loop rule* in the Hoare system is the following, where $P$ is the loop invariant:

$$[While]\ \frac{\{P \wedge b\}\ S\ \{P\} \quad \text{and } b \text{ has no side-effect}}{\{P\}\ \mathsf{while}\ b\ \{S\}\ \{\neg b \wedge P\}}$$

**Example 2.1.** *To give a better understanding of how one can derive a proof in the Hoare deductive system, we illustrate the proof of the example in Listing 2.1. The goal is to prove (partial) correctness of the triple*

$$\{\mathsf{true}\} S \{r \geq 0\},$$

*where $S$ is the initial program in Listing 2.1. We apply rules from the logic to decompose the program into smaller sub-programs, until we obtain triples that are axioms in the logic, or are expressions provable in predicate logic. Figure 2.1 presents the complete proof.*

$$\dfrac{\dfrac{a > b \wedge true \Rightarrow a - b \geq 0 \qquad \dfrac{}{\{a - b \geq 0\}\ r = a - b\ \{r \geq 0\}}\,[\text{Assign}]}{\{a > b \wedge true\}\ r = a - b\ \{r \geq 0\}}\,[\text{Conseq}] \qquad \text{T}}{\{true\}\ S\ \{r \geq 0\}}\,[\text{Cond}]$$

$$\text{T:}\ \dfrac{\neg(a > b) \wedge true \Rightarrow b - a \geq 0 \qquad \dfrac{}{\{b - a \geq 0\}\ r = b - a\ \{r \geq 0\}}\,[\text{Assign}]}{\{\neg(a > b) \wedge true\}\ r = b - a\ \{r \geq 0\}}\,[\text{Conseq}]$$

Figure 2.1: Deductive reasoning

**Automated reasoning** Several years after Hoare's system was developed, in 1976, Dijkstra made the next step towards automatic reasoning [Dij76]. He proposed an alternative formulation of the Hoare system, by means of a *predicate transformer semantics*, which performs a *symbolic evaluation* of statements into predicates: for every statement $S$ in the programming language, the semantics defines a function that transforms a predicate Q to a predicate $P$ that is the *weakest precondition* such that the postcondition Q will be established by the statement $\{P\}\ S\ \{Q\}$. Examples of the weakest precondition computation rule are the following:

$$wp(x = v, Q) = Q[v/x]$$
$$wp(S1; S2, Q) = wp(S1, wp(S2, Q))$$

Dijkstra's semantics gives an effective strategy for axiomatic reasoning. We start from the postcondition of the program and reason about every individual statement in the program going backwards. After every passed statement in the program, the weakest precondition is computed and inserted in the new state. Basically, we are simulating an execution of the program in the backwards direction. The proof is successful if at the end, in the prestate of the program, we get an assertion that can be deduced from the specified program precondition. Further in the thesis, the proof outlines of the examples are presented by inserting assertions in the program itself, instead of using the notion of natural deductive reasoning, as shown in Figure 2.1.

In contrast to backwards reasoning, there also exists a reverse strategy, namely forward reasoning, where instead of the weakest precondition, the predicate transformer computes the *strongest postcondition*. Both the backward and the forward strategies provide the basis for automated reasoning about sequential programs. Based on this theory, several powerful tools have appeared over the last years, *e.g.*, OpenJML [Cok11], Dafny [KL12, Lei12], Key [BHS07], ESC/Java [CK04], VCC [CDH$^+$09a], Spec# [BLS05], and KIV [RSSB98].

### 2.1.2 Modular Verification

So far we have discussed verification of simple imperative programs. In short, given a program $S$, we specify its pre- and postconditions, i.e., formulas $P$ and $Q$ respectively, and we derive a correctness proof of the triple $\{P\}\ S\ \{Q\}$ in the Hoare system.

In practice, however, programs are large and complex. To increase the clarity and the maintainability of the program, normally the code is structured into *modules*. As here we are speaking about imperative programs, with modules we

refer to methods (procedures) in the program. Modularity is not only important to tackle the complexity of the code in the development process, but it is also a crucial requirement for the verification phase.

Modular verification means that we prove correctness of each module of the program separately, from which we can later deduce correctness of the whole program. Once a method is verified to be correct, we can rely on its correctness regardless of where this method is invoked from. Modularity in the verification process brings the following advantages: i) as the program is developed in modules, each module can be verified before it is delivered; this is especially important when modules are developed by different parties, ii) when the same module $m$ is used by different clients i.e., modules where $m$ is invoked, there is no need to reverify $m$, every time when we verify one of its clients; iii) a change in the code that does not affect the module contract does not require that the whole program is reverified; instead it is sufficient to reverify only the module in which the change has been made.

**Method contracts** The techniques for verifying modular software use the concepts of *design by contract*, a regulation for designing software by providing the software components with formal specifications. These standards were set by Meyer [Mey92], when he introduced them on his object-oriented programming language Eiffel.

Design by contract basically extends the assertion mechanism, such that it requires every module in the program to be provided by a specification, also called a *contract*. This approach views the program as a construction of modules, which communicate with each other via their contracts. In particular, when a client calls a certain method, the contract of the called method is the agreement between both parties:

- the *precondition* of the method is the obligation for the client: the method may be called only when the precondition holds;
- the *postcondition* of the method is what the method promises to the client: if the precondition holds at the beginning of the method, the postcondition will be satisfied when the method terminates.

The contract of a method allows one to verify the method in isolation. We say that a method is correct with respect to its pre- and postcondition, i.e., $P$ and $Q$ respectively, if we can derive a correctness proof for the tuple $\{P\}\ S\ \{Q\}$, where $S$ is the body of the method. Once a method is verified, its contract can be used by the client: before invoking the method, the client is obliged to

prove the validity of the method precondition, while after the execution of the method, the client can assume that the method postcondition holds.

More precisely, Hoare logic includes the *method call rule*, where $P$ and $Q$ are respectively the pre- and postcondition of the method $m$, while the function $\mathsf{mbody}(m, v_1, .., v_n)$ represents the body of the method $m$, such that every occurrence of a parameter $x_i$ from the declaration of $m$ is replaced by the corresponding passed argument $v_i$.

$$\frac{\{P\}\ \mathsf{mbody}(m, v_1, .., v_i)\ \{Q\}}{\{P\}m(v_1, ..., v_k)\{Q\}}$$

Intuitively, the rule states that: if in a state in which $P$ holds, we invoke a method $m$ that has previously been verified with respect to pre- and postconditions $P$ and $Q$ respectively, we can deduce that $Q$ holds after the execution of the method.

**Invariants** Besides the pre- and postconditions of the methods, an important part of the program specification are *invariants*. As already discussed in Section 1.3, an invariant (not to be confused with a loop invariant) is a boolean formula that expresses a property expected to continuously hold during the entire program execution. Normally, an invariant represents a relation between certain variables in the program that should always be preserved.

Stating that an invariant should *always* be preserved literally means that the invariant should hold in every program state. In practice, however, this is often impossible to achieve. In particular, preserving constantly the validity of the invariant might happen when the invariant refers to read-only variables only, i.e., variables whose values do not change within the program. Otherwise, any change of the value of a variable might temporarily break the invariant.

Therefore, a more appropriate definition of an invariant is: an *invariant is an expression that holds in every visible state of the program.* In a sequential program, visible state is defined as a state that is a pre- or poststate of a method [LPC+07, HH07]. Thus, within a method it is allowed for an invariant to be broken, as long as at the end of the method and before a method call the invariant is re-established.

For example, consider a program that contains a sorted list of integers. A general requirement is that the list is sorted during the execution of the program and thus, this property may be specified as an invariant over the values from the list. However, normally the program will additionally contain methods for management of the list, e.g., adding an element to the list, removing an element

from the list. During the execution of such a method, the invariant may probably be broken, but it is important that the broken state is just a temporary state and the poststate of the method is a valid state in which the list is again sorted.

Therefore, a verification technique for sequential programs should provide a strategy that allows an invariant to be broken within a method, but guarantees that the invariant holds in every visible state of the program. To prove correctness of an invariant $I$ in the program, for every method $m$ with a body $S$, a precondition $P$ and a postcondition $Q$, one should prove the triple:

$$\{P \wedge I\}\ S\ \{Q \wedge I\}$$

The rule states that a method is correct if it satisfies its contract and it maintains the invariant $I$. Moreover, before every method call, we must prove that the invariant holds, while after every method call, we can assume its validity.

## 2.2 Axiomatic Reasoning about Object-Oriented Programs

Object-oriented programming makes one step further towards modular and maintainable software. However, as this paradigm provides considerable benefits to the process of building software, it also brings new challenges in the development of suitable verification techniques.

In general, an object-oriented program is a collection of classes (templates for creating objects), each containing: fields (attributes), constructors and methods. Consequently, correctness of the program means that every class in the program is correct. Importantly, to make verification modular, we should be able to prove correctness of a given class without having knowledge about the other classes that are part of the program.

**Method contracts** Defining correctness of a class is analogous to correctness of a procedural program. Concretely, methods in the class are specified with contracts, that should be satisfied by the implementation of the method. Once a method is verified to be correct, the client class (the class from where the method is invoked) can use its contract. Verification of constructors, which can be seen as a special category of methods, does not differ from verification of standard methods. Constructors are also equipped with a contract: the precondition must hold when the constructor is invoked, i.e., when the client creates the object; the postcondition holds at the end of the object creation.

**Class invariants** Invariants in an object-oriented program are defined on the level of classes and thus are called *class invariants*. Analogous to invariants in procedural programs, a class invariant defined in a class $C$ expresses a property that must hold in every visible state throughout the life cycle of every object from the class $C$.

Therefore, to say that a class is correct, an additional requirement is that the invariants defined in the class are preserved in all visible states. Concretely, a visible state in an object-oriented program is defined as: a poststate of a constructor, or a pre- or poststate of a *non-helper* (*non-private*) method [LPC+07]. A helper method is viewed as part of another method that carries out the real execution (a public method); thus the pre- and poststates of a helper method are basically internal states of a public method, and the class invariants do not necessarily need to hold in these states.

A class invariant is also not expected to hold in the prestate of the constructor. This is logical, since in the prestate of the object creation, the object fields are still not initialised. Thus, for every invariant $I$, and every constructor with a body $S_c$ and a contract $P_c$ and $Q_c$, we must prove the triple:

$$\{P_c\}\ S_c\ \{Q_c \wedge I\}$$

**Class invariants and the modularity problem** Unfortunately, verifying (class) invariants in a modular way already becomes a challenge. In particular, as discussed above, a technique for modular verification of object-oriented program should ideally be able to prove correctness of a single class, without being aware of which other classes exist in the program.

If an invariant $I$ is defined in the class $C$, proving correctness of $C$ means that we should prove that $I$ is maintained by all non-helper methods in the program. We can of course prove that all methods in $C$ maintain this invariant (we are aware of their existence); however, the invariant might also be broken by a method $m'$ in another class $C'$, of which we do not know that it exists. For example, $I$ may depend on fields from $C'$, which are updated in $m$. Therefore, proving correctness of $C$ in isolation becomes impossible.

The problem of modular verification of class invariants is addressed in more detail in Section 4.2.2. At this point, we just mention that there exist several solutions that deal with this problem [MPHL06, BDF+04, LPX07, DM12]. In general, their common approach is to impose certain restrictions in the definition of the invariants as well as in the program itself. In this way, breaking of a class invariant happens in a more controlled way, i.e., only in the context where the invariant is defined.

**Inheritance** Inheritance, as one of the core principles in object-oriented programming brings additional challenges in the verification phase. This thesis does not deal with inheritance as this is not directly relevant for the techniques that we propose in Sections 4, 5 and 6. Therefore, here we just give a brief intuition of the standard concepts that answer the inheritance question.

Inheritance allows a method $m$ defined in a class $C$ to have different implementations in classes that extend $C$. When the client calls a method $o.m()$, the *dynamic* type of the *receiver object*, i.e., the object $o$, will determine which implementation of $m$ is going to be executed; this is known as *dynamic dispatch*. The dynamic type, however, is determined at run-time, while with static verification, we know only the contract of the method found in the *static* type of the receiver object. This complicates the modular verification of programs that support inheritance.

To this end, techniques for verification of object-oriented software are based on the concept of *behavioural subtypes* [LW94]. The main idea of this principle is that classes that inherit from each other must also inherit their specifications. Concretely, if a method $m$ overrides another method $m'$ then: the precondition of $m$ must be weaker than (implied by) the precondition of $m'$, and the postcondition of $m$ must be stronger than the postcondition of $m'$. Similarly, a class invariant defined in the subclass must always be stronger than a class invariant defined in the superclass.

In other words, behavioural subtyping always guarantees that the subclass preserves the behaviour of the superclass. In this way, when the method $o.m()$ is called, the client can safely use the contract of the method found in the static type of $o$. Therefore, this approach allows modular verification.

## 2.3 Java Modeling Language

We have presented the fundamental principles behind reasoning about sequential programs. In this section, we give a deeper insight into the well-known *Java Modeling Language (JML)* [LPC+07], a specification language suitable for Hoare-style reasoning, which we use as a basis in this thesis. JML is a language developed for specifying the behaviour of Java programs and built in accordance with the design by contract approaches. The strong side of this language is that it provides an interface with a syntax very close to a programming language, which makes it acceptable and easy to use by software developers with only modest mathematical skills.

JML specifications are added directly to the program in the form of annota-

| | |
|---|---|
| **requires** exp | defines exp as a precondition of the method |
| **ensures** exp | defines exp as a postcondition of the method |
| **invariant** exp | defines a class invariant exp |
| **loop_invariant** exp | defines a loop invariant exp |
| **assert** exp | defines an assertion statement exp |
| **also** | combines multiple contracts of a method |
| **spec_public** | defines a public specification visibility of a field |
| **assignable** | defines a set of fields that a method is allowed to assign |
| **pure** | defines that the method has no side-effects |
| **result** | refers to the returned value of a method |
| **old**(exp) | refers to the value of exp in the prestate of a method |
| ∀ x; exp; exp | the universal quantifier |
| ∃ x; exp; exp | the existential quantifier |
| exp ⇒ exp | implication |

Figure 2.2: Basic syntax of JML

tion comments /*@ ... @*/ or //@. Basically, the "@" symbol is used to distinguish JML comments from standard program comments. Figure 2.2 presents the basic elements (this is just a small subset) of the JML syntax. The meaning of most of the terms should be clear from the discussion above. The figure also contains the keywords: **also**, which is used to combine multiple method specifications (pre- and postconditions), such that they are all satisfied by the method implementation; and **spec_public** is used to increase the visibility of a field from the specification point of view: a private or protected field annotated with **spec_public** becomes publicly accessible in the specification. The keywords **assignable** and **pure** are explained later.

In Listing 2.2 we give an example of a simple Java program specified with JML annotations. The program represents an array of positive integer elements. Two class invariants are specified to describe that: the number of elements in the program (size) is in the range [0, array.length]; and every element in the array has a value greater than 0. The contract of the method add(data) describes that: the client can invoke the method only if the passed parameter (data) has a positive value; the method then ensures that the element will be added to the end of the array, and the size of the array will be increased by 1.

**Framing in JML** To provide modular method contracts that will be useful for the client class, sometimes specifying the functional behaviour of the method

```
class ArrayList {
  private /*@ spec_public @*/ int[] array = new int[10];
  private /*@ spec_public @*/ int size = 0;

/*@ invariant 0 ≤ size ∧ size ≤ array.length;
  @ invariant ∀ int i; 0 ≤ i ∧ i < size; array[i] >0;
  @*/

/*@ requires data > 0 ∧ size < array.length;
  @ ensures size == \old(size) + 1 ∧ array[size - 1] == data;
  @ ensures ∀ int i; 0 ≤ i ∧ i < size - 1; array[i] == \old(array[i]);
  @*/
 public void add(int data){
    array[size]=data;
    size = size+1;
 }
}
```

Listing 2.2: Java program with JML specification

only is not sufficient. We give an example in Listing 2.3. The method **set**() assigns the value 5 to x, and invokes the method empty(), which has no behaviour. Proving the postcondition x==5 of the method **set**() is not possible, because the contract of the method empty() does not provide any information about whether it has made any change to x. One way to solve this is to specify explicitly in the method postcondition that the method preserves the values of certain locations. In our example, we could specify a postcondition x==**old**(x) for the method empty(). However, this approach would bring a huge specification overhead, and more importantly, might break the modularity requirements if locations that are to be mentioned are outside of the current context. This problem is known as *the frame problem* [BMR95], and has been studied in [Kas11, Lei98, MÖ2].

JML allows one to specify the *frame* of a method, i.e., the set of locations that are assigned within the method. The frame of a method is also known as the *footprint* of the method. To this end, JML proposes the **assignable** F clause, where F is the method frame. For convenience, instead of listing all concrete locations, JML provides also some additional keywords, e.g., **everything** (for all locations in the program), **nothing** (for no location) or o. * (all fields of the object o). Instead of using **assignable** **nothing** as part of the method contract, one can also use the more convenient form of the keyword **pure** attached to the

```
    //@ requires true;
2   //@ ensures x==5;  // the postcondition can not be proved
    void set () {
4     x=5;
      empty();
6   }
    //@ requires true;
8   //@ ensures true;
    void empty () {
10  }
```

Listing 2.3: The frame problem

definition of the method. For our example in Listing 2.3, we could specify that the method empty() is *pure*, by giving the definition **void** /*@pure@*/ empty ().

**Abstraction** We discussed that modularity is the fundamental concept of software construction: it ensures that the code is structured into smaller modules that are easy to maintain. Modularity goes together with *abstraction*. The concept of abstraction allows components in the system to interact with each other on an abstract level. This means that the module encapsulates all implementation details to make them hidden from the external clients, and exposes only an interface with a restricted set of publicly accessible information. In object-oriented programming this is known as *encapsulation* or *information hiding*. To establish abstraction, an object-oriented language normally provides: interfaces and abstract classes, inheritance, or access modifiers (**private**, **protected**, **public**) that define the access level of a certain field.

In verification of object-oriented software, the language for communication between modules are method contracts. To maintain also abstraction in the specification, it is important that the method contracts adhere to the abstraction regulations: they should hide as much as possible the unnecessary implementation details from the client.

In Listing 2.2, the contract of the method add(data) expresses properties over fields declared as private in the program code. To make them visible by the specification in the client class, their visibility is increased by using the **spec_public** modifier. The drawback of exposing private data via a method contract is that: if the implementation of the method changes, its contract should also be changed, and this will also probably affect the client class. Moreover, it really becomes a problem when abstract methods need to be specified.

To this end, JML provides so called *model variables*. A model variable is a specification-only variable, i.e., visible only by the specification and hidden from the compiler, which is associated with a *relation* function to specify how this variable maps to the real program implementation. The model variable is normally publicly accessible and exposed to the client, while its representation function refers to private fields. JML provides the keyword **model** to declare a model variable, and the keyword **represents** to define the representation function.

The concept of model variables is used by Hoare [Hoa02], where he uses the term *abstract variables*. The idea for these abstract variables is to allow a program to be formally specified during the design phase. Therefore, methods in an interface can be specified with a model variable declared in the interface, and whose representation is specified later in the concrete classes.

In addition to model variables, JML also allows to define model methods. One of the important contributions of JML is also the presence of *model types* [LCC+05]: JML offers a package of model types that describe mathematical data structures, e.g., bags, sets, sequences. These can be used to specify different data structures in the language.

For example, we can add a model variable in the example in Listing 2.2 of type JMLObjectSequence as:

```
public model JMLObjectSequence jmlarray;
```

and relate this variable to the implementation of the array program variable using the expression:

```
represents jmlarray <- JMLObjectSequence.convertFrom(array,size);
```

The specification of the add(data) method can be expressed in terms of the model variable; in this way we can avoid exposing the private data of the class ArrayList to the client side.

Similar to model variables, to provide data abstraction, JML provides also *ghost variables*, declared using the keyword **ghost**. These are also specification only variables that, in contrast to the model variables, are assignable. We assign a ghost variable using the keyword **set**. For example, to declare a ghost variable x and assign 0 to it, we write:

```
//@ ghost x;
//@ set x=0;
```

Basically, a ghost variable is seen as a normal (program) variable by the verifier, but is invisible for the compiler.

## 2.4 Conclusions and Discussions

We have presented the first revolutionary steps in axiomatic reasoning about programs, introduced in the late sixties by Floyd and Hoare. The intention of Hoare-style reasoning is to equip the program with a pre- and postcondition and to derive a correctness proof using the rules of Hoare logic. The proof ensures that the behaviour of a program satisfies the program specifications under any circumstances. Hoare presented his approach on simple imperative programs. His work received a great deal of attention; in the later decades, extensive research has been conducted on applying axiomatic reasoning to real software and more comprehensive programming languages.

To verify real and complex software, modularity was seen as one of the core requirements. Meyer set the standards of design by contract, an approach which requires components of the program to be provided with formal specifications. This approach seems appealing, as it suggests a natural and elegant way to address the problem of modularity; thus, it has been broadly accepted by the software verification community.

However, the design by contract approach brings its own drawbacks because of the need to provide formal specifications. In particular, specifying every method in the program with a formal language is not a trivial task: it is time-consuming and requires expertise. What becomes even more problematic is the fact that to prove correctness of a newly developed component that uses a library, it is necessary that this library is also formally specified.

This critique, however, can be addressed by the fact that documenting the code is a critical part of the design and implementation phase of software. If we try to initiate a new trend of specifying the code with formal specifications instead of specifications in a natural language, this would increase the quality of the documentation. Formal specifications are strict and precise logical formulas, in contrast to the vague and often ambiguous natural language.

Furthermore, designing a suitable specification language is a challenging task. The specification language lies between the programming language and the mathematical language, i.e., the verification logic. On the one hand, a mathematically-oriented specification language is hardly acceptable in practice (it requires mathematical skills); on the other hand, a programming-oriented specification language complicates the verification process.

We discussed JML as a broadly accepted specification language that provides a nice interface that is easily acceptable by programmers. Integrating the language in a verification tool is nevertheless not a trivial task. In [LLM07] Leavens at al., discuss their experience with Spec#, i.e., a JML-based verification system

that extends the C# language; here they discuss all challenges they have faced while developing the system and point out many remaining open questions. In any case, many existing tools today for verification of sequential software are found efficient and useful.

To conclude, we will state that formal verification is a rather new, but very promising software verification technique. It is usually considered as expensive and useful only for safely-critical systems. However, experience shows that the benefit of formal verification is substantial in many different software systems. It considerably improves the quality of software; and importantly, the total costs of software are not significantly increased, but rather shifted from the maintenance phase to the design and implementation phase.

# Chapter 3

# Verification of Concurrent Programs

This chapter gives background on verification of multithreaded programs, i.e., programs that consist of a variable set of threads that run simultaneously and operate on a shared memory.

In Chapter 2 we explained that Hoare triples are fundamental components for axiomatic reasoning. To prove correctness of a given module $S$, one should describe the behaviour of $S$ via a pre- and a postcondition, P and Q respectively, and prove the triple $\{P\}\ S\ \{Q\}$. A successful proof guarantees that $S$ satisfies its specification in any environment where it is used.

This perspective changes in a multithreaded program because of one important fact: *we have to think about the other threads.* In particular, if we want to prove that $\{P\}\ S\ \{Q\}$ is correct in any environment, we have to take into consideration that other threads might be running in parallel, interfering with the current thread, and thus, they might have an impact on the behaviour of $S$. Therefore, to prove $\{P\}\ S\ \{Q\}$, we must show that $P$, $Q$, and any other intermediate assertion used in the proof outline of this triple is a stable predicate, i.e., it can not be invalidated by any external thread. Basically, in a multithreaded program, we can state that a predicate holds in a given state, only if we are sure that in that state the predicate is stable.

This observation shows that the Hoare triples used to reason about sequential programs can not be taken for granted to reason also about multithreaded programs. Verification of multithreaded programs requires a different attitude

and logic that has a control over the interference between program threads. Concretely, in this thesis we use *permission-based separation logic* [BCOP05, Rey02], which allows modular verification of concurrent program and where the control of thread interference is established by the use of permissions.

**Outline** We start with Section 3.1 where we introduce the Owicki-Gries approach, i.e., the first method for reasoning about multithreaded programs. This method is not directly used in this thesis, but it is important as it gives the common intuition for reasoning about multiple threads. Then in Section 3.2 we discuss the logic used in this thesis, i.e., *permission-based separation logic*: in Section 3.2.1 we show the main concepts of the standard separation logic; then in Section 3.2.2 we explain permissions in separation logic, and in Section 3.2.3 we explain how one can reason about programs with synchronisation using this logic. Note that it is important to understand Section 3.2, as it provides background that is essential to follow the rest of the thesis. Next in Section 3.3 we discuss briefly some other common approaches for reasoning about multithreaded programs and in Section 3.4 we conclude.

## 3.1 The First Technique for Concurrent Reasoning

Concurrent reasoning started with Susan S. Owicki and David Gries. In 1976, they were the first to introduce a Hoare-based axiomatic technique for reasoning about multithreaded programs [OG76b, OG76a]. They extended the Hoare sequential language with the parallel construct $\|$, where $S_1 \parallel \ldots \parallel S_n$ describes parallel execution of $n$ threads, for which they defined the following Hoare triple:

$$\frac{\{P_1\}\ S_1\ \{Q_1\} \ldots \{P_n\}\ S_n\ \{Q_n\}\ \textit{hold and proof outlines are interference-free}}{\{P_1 \wedge \cdots \wedge P_n\}\ S_1 \parallel \ldots \parallel S_n\ \{Q_1 \wedge \cdots \wedge Q_n\}}$$

At first glance, this looks like an ideal rule for modular reasoning: to prove correctness of the parallel construct $S_1 \parallel \ldots \parallel S_n$, it is required to prove in isolation that each thread $S_i$ is correct with respect to its local specification (pre- and postcondition $P_i$ and $Q_i$ respectively). However, the modularity is broken because of the premise of the rule, which requires one to show interference-freedom between all triples $\{P_i\}\ S_i\ \{Q_i\}$.

Basically, interference-freedom means stability of the assertions used in the proof rules: to show that $\{P_i\}\ S_i\ \{Q_i\}$ is interference-free from the other threads, one has to show that every (intermediate) assertion $P_i^k$ used in the proof outline of $\{P_i\}\ S_i\ \{Q_i\}$ can not be invalidated by any other thread. To prove stability,

Owicki and Gries propose to inspect all atomic steps of the other threads, and show that none of these steps can invalidate $P_i^k$. More precisely, for any atomic sub-statement $s_i$ from $S_i$, and $s_j$ from $S_j$, where $1 \leq j \leq n, j \neq i$, preceded by intermediate assertions $\mathsf{pre}(s_i)$ and $\mathsf{pre}(s_j)$ respectively, we have to prove the following obligation: $\{\mathsf{pre}(s_i) \wedge \mathsf{pre}(s_j)\}\ s_j\ \{\mathsf{pre}(s_i)\}$.

The interference-freedom requirement makes this approach non-modular and non-scalable. This means that local specifications chosen to describe one thread depend on the behaviour of the other threads. For example, if we want to specify the behaviour of $S_1$, we must also take into consideration the behaviour of $S_2, \ldots, S_n$; the pre- and postcondition of $S_1$ ($P_1$ and $Q_1$) must be chosen such that they stay invariant to the execution of the other threads. Providing such a specification is usually a challenging task (see some examples in [OG76a]). Moreover, if the same code $S_1$ is used in another environment in the presence of different threads, the specification of $S_1$ might not be valid anymore.

Another important observation is that proving stability of all intermediate assertions quickly results in an great number of proof obligations, which makes the Owicki-Gries approach hardly applicable in practice. Nevertheless, the approach does have a high theoretical value, as it provides insight in the nature of concurrent programs and interference between threads. Later in Section 3.3 we review also some other approaches for concurrent reasoning.

## 3.2 Separation Logic for Concurrent Reasoning

A modular verification technique should be able to derive a proof for the triple $\{P\}\ S\ \{Q\}$ locally: we must assume that other threads are running in parallel with the current thread, but we do not have any knowledge what these external threads are actually doing.

The theory of Owicki-Gries shows that deriving such a proof locally is difficult, because any assertion $P_k$ that appears in the proof outline of $\{P\}\ S\ \{Q\}$ (including $P$ and $Q$) may be invalidated by another (unknown) thread that runs in parallel. This invalidation basically happens if this second thread modifies a variable $x$ for which $P_k$ expresses something, i.e., if $x$ appears as a free variable in $P_k$. Therefore, the question raised is: How to ensure that the assertions used in the proof outline are stable and resistant to the external threads?

### 3.2.1 The Basic Concepts of Separation Logic

The question above has been addressed by *separation logic*, a logic initially invented by John C. Reynolds [Rey02] in 2002. Reynolds introduced this logic

to allow reasoning about programs with pointers, for which he showed that Hoare logic was not applicable. Short time after, Peter W. O'Hearn realised that separation logic is also suitable to deal with parallelism. Therefore, he made a small adjustment to this logic and introduced its new version, called *concurrent separation logic* [O'H07], which was broadly accepted as a logic for reasoning about multithreaded programs.

Two main flavours of separation logic can be found in the literature: *intuitionistic separation logic* [IO01, Par05], which we use in this thesis, and *classical separation logic* [JP11, BCO05]. Later in this section we discuss the main intention of both logics and the differences between them.

**Separation** Separation logic aims to provide local reasoning. This is achieved by separating the memory (from where comes the name separation logic). In particular, the shared memory can be separated into *disjoint* partitions, such that reasoning about a triple $\{P\}\ S\ \{Q\}$ is done over a single partition only, rather than over the whole global memory. Over which partition we reason, is defined by the precondition $P$. Basically, the predicate $P$ describes properties over certain memory locations, and these locations define the precise local partition (or the local state). $P$ gives the right to the program statement $S$ to access the local partition only: everything that is mentioned in $P$ is local and might be accessed (read or written) in $S$; everything that is not mentioned in $P$ belongs to somebody else, and must not be touched in $S$.

To allow separating and merging memory partitions, separation logic introduces the *separating conjunction* operator $*$. The expression $P*Q$ describes that the current memory contains two disjoint partitions, such that $P$ holds over the first, while $Q$ over the second partition. This operator is closely related to the standard conjunction operator $\wedge$, but is stronger: $P*Q$ requires that both $P$ and $Q$ hold, and additionally requires that both predicates do not refer to common memory locations.

In addition to the sequential conjunction, two other binary operators are used (although less frequently) in separation logic:

- the *separating implication* operator $-\!\!*$, also known as *magic wand*; the expression $P-\!\!*Q$ asserts that: if we extend the current state with a disjoint partition on which $P$ holds, the result is a partition on which $Q$ holds;

- the *separating equivalence* $*\!\!-\!\!*$ denotes a two-way magic wand: $P*\!\!-\!\!*Q$ states that both $P-\!\!*Q$ and $Q-\!\!*P$ hold.

Having introduced the concept of memory partitioning, a consequent rule to present is the *frame rule*, one of the fundamental elements in separation logic:

$$[\textit{Frame}]\ \frac{\{P\}\ \ S\ \ \{Q\}}{\{P{*}R\}\ \ S\ \ \{Q{*}R\}}$$

The frame rule explains the principle of local reasoning: if we can prove $\{P\}\ S\ \{Q\}$ locally, on a smaller memory partition, we can conclude that the triple $\{P{*}R\}\ S\ \{Q{*}R\}$ holds on an extended memory, i.e., $S$ has not changed anything in the extension $R$.

**Concurrency** Separation of the memory and local reasoning has shown to be the right direction to handle programs with parallel threads. In principle, parallel threads can safely operate on disjoint memory partitions, without causing any interference. Thus, if we say that we distribute disjoint memory partitions to different parallel threads, then we can reason about a single thread in isolation, ensured that this thread does not interfere with any other parallel thread.

Therefore, concurrent separation logic introduces the following rule:

$$[\textit{Parallel}]\ \frac{\{P_1\}\ \ S_1\ \ \{Q_1\} \qquad \{P_2\}\ \ S_2\ \ \{Q_2\}}{\{P_1{*}P_2\}\ \ S_1 \parallel S_2\ \ \{Q_1{*}Q_2\}}$$

The rule explains the following: if the initial memory can be separated into two partitions, described by predicates $P_1$ and $P_2$, such that we can distribute each partition to a separate parallel thread, then we can reason locally about each thread, and at the end combine the two postconditions to $Q_1{*}Q_2$.

**The *points-to* predicate** In Section 2.1, we presented Hoare logic, which is suitable to reason about a simple language that does not contain pointers. The program state in this logic is represented by a set of variables that can be assigned to arbitrary values. In contrast, separation logic was invented to reason about programs with pointers. Thus, in contrast to Hoare logic, separation logic explicitly distinguishes between:

- the *heap* $h$, which represents the shared memory; it is a mapping $\mathsf{Loc} \rightharpoonup \mathsf{Value}$ from memory addresses (locations) to values.
- the *store* $s$, which represents the local memory; it is a mapping $\mathsf{Var} \rightharpoonup \mathsf{Value}$ from variables to values (including memory addresses, $\mathsf{Loc} \subseteq \mathsf{Value}$);

| | |
|---|---|
| $x \mapsto 2 * y \mapsto 2$ | $x$ and $y$ are pointers that point to two *different* memory locations, both of them with value 2. |
| $x \mapsto 2 \wedge y \mapsto 2$ | $x$ and $y$ are pointers that point to a single memory location with value 2, or to two different memory locations, both with value 2. |
| $x \mapsto 2 * y == x$ | $x$ and $y$ are pointers that point to the same memory locations with value 2. |
| $x \mapsto \_ * x \mapsto \_$ | this expression is equivalent to false. |

Figure 3.1: Assertions in separation logic

When we talk about partitioning of the memory in separation logic, we always refer to the shared memory, i.e., partitioning of the heap.

A core ingredient in separation logic is the predicate $x \mapsto v$, known as the *points-to* predicate. The meaning of this predicate is that a pointer $x$ points to a location at which the value $v$ is stored. More concretely, the store $s$ maps $x$ to a memory address, and the heap $h$ maps this address to $v$, i.e., $h(s(x)) = v$. Importantly, in addition to this, the predicate $x \mapsto v$ serves as a *permission (access ticket)*, i.e., it gives the current thread the right to access the location of $x$. When the value $v$ is unknown or not relevant, we write $x \mapsto \_$. Furthermore, we use the notation $x \mapsto ?v$ when the value of $x$ is unknown, but we name it $v$.

To give some more understanding of the predicates in separation logic, in Figure 3.1 we present a few examples of assertions and their meaning. The last expression deserves some attention. An expression $x \mapsto \_ * x \mapsto \_$ always leads to a contradiction, because both the left and right operand refer to the same memory location and thus, they can never hold on two disjoint memory partitions. In particular, the whole meaning of having the $x \mapsto v$ predicate in the logic is that this predicate is *unique*: there can always exist at most one predicate for a single location: when a thread owns this predicate, it is the only thread in the program allowed to access this location.

**Hoare triples** Below we illustrate a few Hoare triples that describe the basic concepts of separation logic (see [Rey02] for more details). Note that we use the notation $[x]$ for pointer dereferencing: $[x]$ denotes the value stored in the heap on the location where $x$ points to, i.e., $h(s(x))$. In contrary, $x$ denotes the value that is directly assigned to $x$ in the store, i.e., $s(x)$.

*Local assignment* The instruction $x = v$ assigns the value $v$ to the variable $x$. There is no access to the heap, and therefore, no permission for $x$ is necessary. The related Hoare triple is defined as:

$$\textit{[Local Set]}\ \frac{}{\{\mathsf{true}\}\ x = v\ \{x == v\}}$$

*Allocation* The instruction $x = \mathsf{cons}(v)$ denotes allocating a new memory location with a value $v$ and associating the pointer $x$ to this location. The Hoare triple for this construct describes that this is the instruction (and the only one) that produces the permission to access the location where $x$ points to:

$$\textit{[Allocation]}\ \frac{}{\{\mathsf{true}\}\ x = \mathsf{cons}(v)\ \{x\ \mapsto\ v\}}$$

The semantics of the $\mathsf{cons}(v)$ instruction takes care to reserve always a *free* memory address for the newly allocated location. This ensures that there is a single permission predicate for each memory location.

*Read heap access* The construct $y = [x]$ represents assigning to $y$ the value $[x]$. Therefore, to access this heap location, a permission is required, i.e., the predicate $x\ \mapsto\ v$:

$$\textit{[Read]}\ \frac{}{\{x\ \mapsto\ v\}\ y = [x]\ \{x\ \mapsto\ v\ *\ y == v\}}$$

*Write heap access* Similarly, to assign to a heap location $[x]$, it is required that the current thread has a permission to access this location, i.e., the $x\ \mapsto\ v$ predicate should hold:

$$\textit{[Write]}\ \frac{}{\{x\ \mapsto\ \_\}\ [x] = v\ \{x\ \mapsto\ v\}}$$

**Intuitionistic v.s. classical separation logic** As mentioned above, apart from intuitionistic separation logic, which is used in this thesis, another common alternative is classical separation logic. Below we make a short comparison between these two logics. Primarily, they differ in the semantics of the $x\ \mapsto\ v$ predicate. In the intuitionistic version, $x\ \mapsto\ v$ defines the heap only partially: it states that $x$ points to a heap location with value $v$, but we do not know whether there also exist other allocated heap locations. In contrast, the same predicate in classical separation logic entirely defines the heap: $x\ \mapsto\ v$ means that there is precisely one location on the heap with value $v$, and this location is referred to by $x$.

This means that intuitionistic separation logic allows weakening. For example, if the predicate $x \mapsto v * y \mapsto w$ holds over a state, it can be replaced by a weaker predicate $x \mapsto v$, i.e., we can simply forget part of the state. This is not possible in classical separation logic. This distinction makes the intuitionistic version appropriate for languages that support garbage-collection, for example Java or C#; in contrast, the classical version is normally used for languages with manual memory deallocation, like C or C++.

**Ownership transfer** Separation logic provides a different perspective on reasoning. In particular, the presence of permissions classifies expressions written in separation logic into two categories:

- *pure expressions*, which do not contain permissions; these expressions basically express some knowledge over the store only, and this knowledge is allowed to be duplicated and shared between threads. For example, $x == 1$ is equivalent to $x == 1 * x == 1$;

- *resource assertions*, which express properties over the heap, i.e., they contain permissions; such an expression can not be shared between threads: $x \mapsto 1$ is *not* equivalent to $x \mapsto 1 * x \mapsto 1$.

Practically, assertions in separation logic represent some kind of *ownership*. An expression $e$ expresses some properties over the state and this expression is *owned* by the current thread. The concept of reasoning is then based on transferring ownership. The thread can duplicate this expression (in case it is pure), and transfer the copy to another thread; or it can give up its ownership and transfer the expression to another thread (in case of a resource expression).

How this transfer happens is controlled by the specifications of the threads. If a caller thread starts a new thread that requires some resources (stated by its precondition), the caller must provide the new thread with these resources. When the caller thread joins another thread, the postcondition of the terminating thread defines the resources that will be transferred to the caller. This transferring of resources is shown in the example below.

**Example 3.1.** *Listing 3.1 gives an example of reasoning using separation logic. Initially, two new heap locations are created in the program,* x *and* y*, both with values* 0 *(lines 4 and 6). Thereafter, two parallel threads start to execute in parallel (line 8): the first one increments the value of* x *by* 1 *(method* incrX()*),*

```
    //@ requires true;
 2  //@ ensures  x ↦ 1  *  y ↦ 1;
    void main(){
 4    x=cons(0);
    {x ↦ 0}
 6    y=cons(0);
    {x ↦ 0  *  y ↦ 0}
 8    incrX() || incrY();
    {x ↦ 1  *  y ↦ 1}
10  }

12  //@ requires x ↦ v;
    //@ ensures x ↦ v+1;
14  void incrX(){
       l=[x];
16  {x ↦ v  *  l==v}
       l′=l+1;
18  {x ↦ v  *  l′==v+1}
       [x]=l′;
20  {x ↦ v+1}
    }
22
    //@ requires y ↦ v;
24  //@ ensures y ↦ v+1;
    void incrY(){
26     l=[y];
    {y ↦ v  *  l==v}
28     l′=l+1;
    {y ↦ v  *  l′==v+1}
30     [y]=l′;
    {y ↦ v+1}
32  }
```

Listing 3.1: Reasoning with separation logic

*and the second thread increments the value of* y *by* 1 *(method* incrY()*)* [1]*. Thus, both threads do not share any common memory. We want to prove that after both threads finish their executions, the value of both* x *and* y *is* 1*.*

*The proof outline shows that reasoning with separation logic can be viewed from the perspective of transferring access tickets. After both locations are created, two access tickets are produced:* $x \mapsto 0$ *and* $y \mapsto 0$ *(lines 5 and 7). When the two threads are forked (line 8), the client must distribute both tickets to the threads, as they require in their preconditions. If the client did not hold all required tickets, the program could not be verified. When threads are joined, the client obtains the tickets back, as both threads ensure that they will return the borrowed ticket back in their postconditions.*

**Framing** We introduced in Section 2.3 the framing problem and explained why is it important for the contract of a method to define explicitly which locations the method assigns (the **assignable** clause). When using separation logic, specifying the assignable clause is not necessary. In particular, framing

[1] Note that the value $v$ in the requires clause is not a specific value, but an unknown value that we name $v$. In particular, we could use $x \mapsto ?v$, but we avoid this representation in the examples for a simpler presentation.

here is done implicitly, via the points-to predicates. In general, the locations that may be assigned in the method are described via permissions specified in the method's precondition. We say that expressions in separation logic are *self-framing.*

**Locality and data race-freedom** What is important to remember about separation logic is that it allows local reasoning. Methods are equipped with local specifications that are valid in any environment. Basically, the concept of using permissions makes expressions in separation logic stable and resistant to external threads; thus a proof outline is always free of interferences.

Importantly, permissions ensure that at a certain moment, at most one thread might access a given heap location. Therefore, *a program verified using separation logic is guaranteed to be data race-free.*

**Abstract predicates** Method specifications must express permissions to all locations that are accessed in the method. When using only standard specification expressions, method specifications might become too complex and unclear. Moreover, specifying the method might even become impossible. For example, for a recursive list data structure, it is not possible to express permissions for all nodes in the list.

To this end, in 2005 Matthew J. Parkinson and Gavin M. Bierman extended separation logic with a new abstraction mechanism - *abstract predicates* [PB05]. They added abstract predicates to an object-oriented Java-like language, but the same concept is applicable to any modular language. In their work they discuss their improved technique, which is suited for object-oriented programs with inheritance.

A predicate is abstract when it is associated with a *name* and a *scope*; the scope is normally the class (the module) where the predicate is defined. Swapping from name to definition and vice versa is known as *opening (unfolding)* and *closing (folding)* the predicate, respectively. Within its scope, the predicate can be folded or unfolded, and therefore, it can be used both with its name or its full definition. Outside its scope, the predicate is always closed, which means that clients can access the predicate only by its name, but the definition of the predicate is not accessible. The problem of specifying recursive list data structure mentioned above, can easily be solved by defining a recursive abstract predicate, and expressing the method specification via the name of this predicate.

**Example 3.2.** *Listing 3.2 illustrates the use of abstract predicates. The class* `Point` *contains two fields* `x` *and* `y`*. We define an abstract predicate with the*

```
class Point{
  int x, y;
  pred state(int vx, int vy):
    x ↦ vx * y ↦ vy;
  //. . . other methods

  //@ requires state(vx, vy);
  //@ ensures state(0, 0);
  void setToCenter(){
  {this.state(vx, vy)}
    // unfold state predicate;
  {x ↦ vx * y ↦ vy}
    x=0;
    y=0;
  {x ↦ 0 * y ↦ 0}
    // fold state predicate;
  {this.state(0,0)}
  }
}
class Client{
  Point p;
  //. . . other methods

  //@ requires p.state(vx, vy);
  //@ ensures p.state(0, 0);
  void movePoint(){
    p.setToCenter();
    //unfold p.state is not possible
  }
}
```

Listing 3.2: Abstract predicates in separation logic

*name state parametrised by two integers; the definition of the predicate describes permissions to x and y. The specification of the class Point is defined in terms of the abstract predicate; therefore, no information about the internal fields x and y is exposed to the client. Within the context of the class Point, the predicate may be folded or unfolded, and therefore, a thread may use permissions to x and y to access the fields (see method setToCenter()). The client, however, can access only the name of the predicate, but not its definition.*

### 3.2.2 Extending Separation Logic with Permissions

Separation logic allows *disjoint* separation only, but this is too restrictive. In particular, the logic always forbids two threads to access the same location, without making a distinction between read and write accesses. In practice, however, sharing a location between several threads is safe and does not cause a data race, as long as these threads are just reading the shared location. Thus, a program with parallel reads, which is evidently data race-free, cannot be verified by standard concurrent separation logic.

**Fractional permissions** In 2003 John T. Boyland identified this problem, and addressed it with a very clever idea based on *splitting permissions* [Boy03]. He presented his idea on a new type system for checking interference of parallel

threads, but suggested that fractions would also be applicable to separation logic to allow parallel reads.

A central point of Boyland's approach is the *fractional permission*, i.e., a rational number $\pi$ in the interval $(0, 1]$. We mentioned that in standard separation logic, the $x \mapsto \_$ predicate can be considered as a permission (an access ticket) to a location $x$, which represents a full ownership of the location $x$; a thread either has a *full access* or *no access* to $x$. In contrast, in Boyland's system, a thread may have a *full access* to a location, a *partial access* or *no access*. A full access is represented by a permission $\pi = 1$, we call it a *write* permission. A write permission ensures exclusive ownership and allows a thread to both read and write to a given location. Any permission $\pi$ may be split into two permissions (fractions) $\pi_1$ and $\pi$-$\pi_1$. Similarly, permissions $\pi_1$ and $\pi_2$ may be merged into a new permission $\pi_1 + \pi_2$, provided $\pi_1 + \pi_2 \leq 1$. A permission in the interval $(0, 1)$ is called a *read* permission; it gives a thread the right to only read the location.

The amount of the read permission does not make any difference in what it allows the program to do. The fraction 1/10 gives the same right as the fraction 9/10. However, keeping track of these values is important: if the write permission is split into fractions, we need to collect all pieces if we want to obtain back the write permission. It might happen that the fraction is lost (for example, if the thread is specified such that it ensures to return a smaller amount of permissions than it required). This means that thereafter, no thread will have the right to write to this location. Note that this is typically used to specify that a location becomes *read-only*.

Importantly, the concept of fractional permissions ensures data race-freedom because the sum of permissions of all threads for the same location does not exceed 1. Therefore, parallel readings at the same location $x$ are allowed; however, when a thread is writing on $x$, it fully owns this location, and no other thread has the right in that moment to access $x$, neither for writing nor for reading.

**Fractional permissions in separation logic** Following Boyland's idea, Bornat et al. extend standard separation logic with fractional permissions [BCOP05]. The assertion $x \overset{\pi}{\mapsto} v$ is introduced to express that $x$ points to a location on the heap containing the value $v$, while the current thread holds at least permission $\pi$ for this location. We call $x \overset{\pi}{\mapsto} v$ a *permission predicate*.

Figure 3.2 presents the Hoare triples already introduced in Section 3.2, but now modified according to the requirements of permission-based separation logic: a write permission is produced when a new memory address is allocated

$$\textit{[Local Set]}\ \frac{}{\{\mathsf{true}\}\ x = v\ \{x == v\}}$$

$$\textit{[Allocation]}\ \frac{}{\{\mathsf{true}\}\ x = \mathsf{cons}(v)\ \{x \overset{1}{\mapsto} v\}}$$

$$\textit{[Read Access]}\ \frac{}{\{x \overset{\pi}{\mapsto} v\}\ y = [x]\ \{x \overset{\pi}{\mapsto} v\ *\ y == v\}}$$

$$\textit{[Write Access]}\ \frac{}{\{x \overset{1}{\mapsto} \_\}\ [x] = v\ \{x \overset{1}{\mapsto} v\}}$$

$$\textit{[Split/Merge]}\quad x \overset{\pi_1+\pi_2}{\mapsto} v_1 * v_1 == v_2 \mathrel{*\!\!-\!\!*} x \overset{\pi_1}{\mapsto} v_1 * x \overset{\pi_2}{\mapsto} v_2$$

Figure 3.2: Hoare triples in permission-based separation logic

(rule *[Allocation]*); any permission $\pi$ is sufficient for reading a shared location (rule *[Read Access]*); while for writing a shared location, a write permission is necessary (rule *[Write Access]*).

In addition, Figure 3.2 presents the *[Split/Merge]* axiom, which describes splitting points-to predicates: the $*\!\!-\!\!*$ operator can be read as "splitting" (from left to right) or "merging" (from right to left). Predicates that have this property to be split or merged over permissions are called *groups*. In general, a group is a predicate, parametrised by a permission variable $P(\pi)$ for which the following holds: $P(\pi_1 + \pi_2) \mathrel{*\!\!-\!\!*} P(\pi_1) * P(\pi_2)$.

**Example 3.3.** *Listing 3.3 illustrates the idea of splitting and merging access tickets. Initially the client allocates three shared locations:* x *and* y *with values* 0*, and* z *with value* 1 *(lines 4, 6 and 8). Thereafter, it forks two threads (line 11): the first thread increments* x *by the value of* z *(method* incrX()*) and the second thread increments* y *by* z *(method* incrY()*). Thus, both threads share a read access to* z*. To this end, when the client distributes the access tickets, the ticket for the location* z*, i.e.,* $z \overset{1}{\mapsto} 1$ *is split into two fractions, and each fraction is transferred to one thread. In this way, both threads can write to the location on which they have full ownership (*x *and* y *respectively), and they can simultaneously read the value of the location* z*.*

Notice that when specifying the program, the user does not need to calculate and specify the concrete amount of read permissions, but it is sufficient to specify the program with an abstract permission, represented by a specification-

```
    //@ requires true;
 2  //@ ensures x ↦^1 1 * y ↦^1 1 * z ↦^1 1;
    void main(){
 4    x=cons(0);
    {x ↦^1 0}
 6    y=cons(0);
    {x ↦^1 0 * y↦^1 0}
 8    z=cons(1);
    {x ↦^1 0 * y ↦^1 0 * z ↦^1 1}
10  {x ↦^1 0 * y ↦^1 0 * z ↦^1/2 1 * z ↦^1/2 1}
     incrX() || incrY();
12  {x ↦^1 1 * y ↦^1 1 * z ↦^1/2 1 * z ↦^1/2 1}
    {x ↦^1 1 * y ↦^1 1 * z ↦^1 1}
14  }

16  //@ requires x ↦^1 v * z ↦^π w;
    //@ ensures x ↦^1 v+w * z ↦^π w;
18  void incrX(){
      lx=[x];
20    lz=[z];
      l'=lz+lx;
22    [x]=l';
    }
24

    //@ requires y ↦^1 v * z ↦^π w;
26  //@ ensures y ↦^1 v+w * z ↦^π w;
    void incrY(){
28    ly=[y];
      lz=[z];
30    l'=lz+ly;
      [y]=l';
32  }
```

Listing 3.3: Reasoning with separation logic

only variable (see the specification of methods incrX() and incrY() in Listing 3.3). In principle, this abstract permission is encoded as a method parameter. This is rather flexible for the caller, because the concrete value of the abstract permission is decided by the caller side. If the called method requires an abstract read permission for a given location, then any positive permission $\pi$ owned by the client would be sufficient to serve the caller: concretely the called method then receives $\pi/2$.

From the perspective of specifications, it is important that when specifying a method, we take care of whether we want the same amount of a given required permission to be given back to the caller. In that case, the **ensures** clause must express the same abstract permission as stated by the **requires** clause, see the specifications of methods incrX() and incrY() in Listing 3.3. Otherwise, there is a risk that the permission is lost (or leaked), as we already discussed above.

**Accounting permissions** The management of permissions basically defines some protocol that the program is expected to obey. Fractional permission are only one way to achieve this. The main advantage of fractional permissions is that every permission can be split an infinite number of times. This approach

is useful for programs where splitting is symmetrical; for example divide-and-conquer where all subtasks are reading from some shared memory. A fractional permission, however, knows only about the current thread, and it does not keep track where it has been borrowed from.

Fractional permissions are not suitable for all kind of applications. To this end, other mechanisms for accounting permissions can also be found in the literature. A well-known alternative are so-called *counting permissions* [BCOP05]. This technique is suitable for problems where the number of possible readers is important: for example, when permissions are protected by a semaphore that counts the number of possible readers. With the counting permission method, the distribution of permission is controlled by one "central" thread. When a thread requires a partial permission from the central thread, the splitting is not done symmetrically, but the requesting thread gets only one permission unit $\epsilon$, which is further not splittable. The central thread is therefore able to count all given permissions, and to recombine them all in a complete write permission.

Other permission models are for example the *symbolic approach* for permission accounting introduced by Huisman and Mostowski [HM15], where the representation form of a permission gives a whole history view of where exactly this permission originates from, or the *tree-based permissions* [DHA09], which distinguish only between the current thread and all other threads. Furthermore, Heule et al. define a permission model based on abstract definition of fractions [HLMS11], where the user specifies the program with abstract values, while permission operations are defined to enable splitting and recombining permissions.

### 3.2.3 Synchronisation and Separation Logic

So far, we have shown that permissions might be transferred when starting a new thread (from the caller to the new thread) or when joining a thread (from the terminating thread to the caller). In a concurrent program, however, threads are not expected to work on the same shared memory during their whole life cycle. Instead, a thread is expected to give up or gain access to shared resources during its execution. To ensure that at most one thread accesses a shared resource at a time, threads might have to wait for each other at certain points in the program. To this end, one needs to use a synchronisation mechanism in the program.

The term synchronisation normally refers to *lock-based synchronisation*. A lock (also known as a *monitor*) is used to synchronise threads in the program in the following way. Initially, when a lock is created in the program, its state is *free*. Thereafter, threads may try to *acquire* the lock. The first thread that wins the lock becomes its *owner*, the lock is therefore in an *owned* state. Any

thread that tries to acquire a lock that is owned by another thread, has to wait until the lock is *released*.

**Reasoning about locks** Verification techniques that allow reasoning about locks are based on so called *resource invariants* [O'H04]. A resource-invariant based method was already presented in the work of Owicki and Gries [OG76b]. They defined a resource invariant as a specification expression (a predicate) associated to a lock (they call a lock a *resource*) that expresses a property over locations protected by the lock and is expected to hold every time when the lock is free.

In a separation logic setting, the concept of using resource invariants is rather similar to the one presented by Owicki and Gries. A resource invariant is an expression that describes the state that is protected by the associated lock. Practically, by specifying a resource invariant for a given lock, we express how we intend to use the lock. For example, if we aim our lock *lockX* to give a thread a full (write) access to $x$, we can associate an invariant to this lock, defined as:

$$I_{lockX} : \exists v.x \overset{1}{\mapsto} v * v > 0$$

The invariant $I_{lockX}$ defines a write permission to $x$, and additionally describes a functional property over $x$, i.e, the value of $v$ is positive. Permissions in the resource invariant are useful for the verifier to check data race-freedom in the program. A functional property may additionally be specified to describe the expected behaviour of the program. Concretely, a functional property expressed via the resource invariant is expected to hold every time when the lock is free, while it can be invalidated when a thread owns the lock.

When the lock is free, we say that the resource invariant, and thus, all permissions that it defines, are owned by the lock. The functional property expressed via the resource invariant holds over the shared state, and since permissions to these locations are protected by the lock, no thread can invalidate it. When a thread acquires the lock, the resource invariant is transferred from the lock to the thread. The thread can then use these permissions to access protected shared memory and possibly invalidate temporarily the properties expressed via the invariant. When the thread releases the lock, these properties must hold over the shared state, and the resource invariant is transferred from the thread back to the lock.

When creating a lock, the thread must provide the lock with the associated invariant. This means that all locations that the lock protects must be already allocated, and the caller thread must hold all these permissions (folded in the

$$[Commit]\ \frac{}{\{I_{lock}\}\ \mathsf{commit}\ lock\ \{\mathsf{true}\}}$$

$$[Acquire]\ \frac{//lock\ has\ been\ committed}{\{\mathsf{true}\}\ \mathsf{acquire}\ lock\ \{I_{lock}\}}$$

$$[Release]\ \frac{}{\{I_{lock}\}\ \mathsf{release}\ lock\ \{\mathsf{true}\}}$$

Figure 3.3: Hoare triples for non-reentrant locks

form of a resource invariant predicate). Transferring the invariant from the caller to the lock at creation is called *committing* the lock.

Therefore, when using locks, the ownership of resources is transferred between the existing threads and the locks in the program. This is analogous to the *Separation property* defined by O'Hearn in [O'H07]: "at any time, the state can be partitioned into that *owned* by each process and each mutual exclusion group". In our context, a mutual exclusion group is equivalent to a lock.

Figure 3.3 presents the Hoare triples for lock-related programming constructs (we explain later the meaning of *non-reentrant* locks used in the title of the figure). The rules basically explain how the resource invariant is transferred between threads in the program and a lock: i) to commit a lock, the thread must hold the related resource invariant, which is then transferred to the lock (*[Commit]* rule); ii) after a thread acquires the lock, it obtains the resource invariant from the lock (*[Acquire]* rule); and iii) after releasing the lock, the resource invariant is transferred back from the thread to the lock (*[Release]* rule).

**Example 3.4.** *The example in Listing 3.4 illustrates how one can reason about a program with locks. The program is a modified version of the example in Listing 3.3: here we use locks to protect the accesses to the shared locations* x, y *and* z.

*The first thread writes on a location* x *and reads on* z *(method* incrX()*), while in the same time another thread writes on a location* y *and reads on* z *(method* incrY()*). We define the following protocol. We create two locks: lockX protects write access to* x *and read access to* z*, while lockY protects write access to* y *and read access to* z*. If a thread holds both lockX and lockY , it has write access to* z*. Accordingly, two resource invariants are specified (lines 1 and 2). The resource invariant for* lockX *stores permission* 1 *for* x *and a permission* 1/2 *for* z*, while*

```
   // lockX_Inv: x ↦^1 _ * z ↦^1/2 _
2  // lockY_Inv: y ↦^1 _ * z ↦^1/2 _

4  void main(){
     x=cons(0);
6  {x ↦^1 0}
     y=cons(0);
8  {x ↦^1 0 * y ↦^1 0}
     z=cons(1);
10 {x ↦^1 0 * y ↦^1 0 * z ↦^1 1}
   {x ↦^1 0 * y ↦^1 0 * z ↦^1/2 1 * z ↦^1/2 1}
12   commit lockX;
   {y ↦^1 0  *  z ↦^1/2 1}
14   commit lockY;
   {true}
16   incrX() || incrY();
   }
18

20

   void incrX(){
22   acquire lockX;
   {x ↦^1 v * z ↦^1/2 w}
24     lx=[x];
       lz=[z];
26     l'=lz+lx;
       [x]=l';
28 {x ↦^1 v+1 * z ↦^1/2 w}
     release lockX;
30 }

32 void incrY(){
     acquire lockY;
34 {y ↦^1 v * z ↦^1/2 w}
       ly=[y];
36     lz=[z];
       l'=lz+ly;
38     [lx]=l';
   {y ↦^1 v+1 * z ↦^1/2 w}
40   release lockY;
   }
```

Listing 3.4: Reasoning with resource invariants

*the invariant for lockY stores permission* 1 *for* y *and a* 1/2 *for* z.

*Initially, the main thread allocates the new locations* x, y *and* z *for which it obtains write permissions. Thereafter, it distributes the permissions to both locks, lines 12 and 14. To access locations* x *and* z *in method incrX() the thread needs to acquire the lock lockX and obtain permissions. Analogously in incrY() the access to* y *and* z *is allowed after obtaining lock lockY.*

**Locks and specifying program's behaviour** This scenario in Listing 3.4 shows an example of *internal synchronisation*. Both methods incrX() and incrY() are synchronised internally. For example, the program segment between acquiring and releasing the lock (lines 22 and 29) is synchronised, but not the whole method incrX(). Because the program is multithreaded, the prestate of the **acquire** lockX instruction is not equivalent to the prestate of the method, and

the poststate of the **release** lockX is not equivalent to the poststate of the method. Any other thread might change the values of x or z in the non-synchronised part.

Internal synchronisation causes difficulties in providing behavioural specifications of the program. In particular, the program in Listing 3.4 is properly equipped with permission specifications, so that a verifier can verify data race-freedom of the program. However, specifying a pre- and postcondition of both methods incrX() and incrY() is troublesome, unless we define the trivial expressions **true** that do not express any behaviour. In contrast to this example, the program in Listing 3.3 does not suffer from this problem: the two methods there are entirely synchronised and thus, there it is trivial to provide expressive method specifications. The problem of internal synchronisation and behavioural specifications is addressed extensively later in Chapter 5.

**Reentrant locks** In some applications, it may happen that a thread acquires the same lock multiple times, without releasing the lock in between. Consider for example a recursive method, in which a lock is acquired in every iteration. Locks that can be acquired multiple times are called *re-entrant locks*. For a thread to release (completely) an owned lock, it is important that it releases the lock the same number of times as it has acquired the lock.

If we now take a closer look at the rules in Figure 3.3, we can conclude that they are not sound for reentrant locks. Consider that a thread acquires twice the same lock with an associated invariant $x \overset{1}{\mapsto} \_$, then the thread will obtain the same resource invariant twice, as shown below:

$\{$**true**$\}$
**acquire** lock;
$\{x \overset{1}{\mapsto} \_\}$
**acquire** lock;
$\{x \overset{1}{\mapsto} \_ * x \overset{1}{\mapsto} \_\}$ //invalid

Therefore, when a thread reacquires a lock, it might end up with a permission for a shared location that is greater than 1, which of course is not sound in permission-based separation logic. Therefore, the rules in Figure 3.3 are valid for non-reentrant locks only.

In [HHH08] Haack et al. explain their idea for reasoning about programs with re-entrant locks. They extended the standard technique for reasoning about locks, with a mechanism that counts the number of times that a thread has acquired a lock. Concretely, each thread maintains a multi-set of locks that it currently owns. The number of occurrences of a lock in this set indicates "how many times the thread owns the lock". When this number is $n > 0$, the

thread has to execute $n$ times the release instruction, in order to completely release the lock. In this way, we can distinguish between first acquiring and re-acquiring a lock: a thread obtains the resource invariant from the lock, only when it acquires the lock for the first time. Similarly, we can identify when is the final releasing of the lock: only at the final release, the resource invariant is transferred back from the thread to the lock.

**Other synchronisation mechanisms** Apart from locks, there also exist other synchronisation mechanisms, but reasoning about them is based on a similar concept. A resource invariant is specified to describe which permissions the synchroniser stores, and permissions are transferred between the threads and the synchroniser via the operations that the synchroniser offers. Examples of other synchronisers are *a semaphore* or *a barrier*, which we shortly describe below. In our work in [ABH+14b], we present different synchronisers from the Java API, and discuss their formal specification and verification in more details.

*Semaphore* A semaphore is a synchronising mechanism that controls access of threads to a certain shared memory state. Each semaphore is provided by an integer property $n$, that defines the maximal number of threads that can access the shared state simultaneously. If $n = 1$ the semaphore is equivalent to a lock. For $n > 1$, the semaphore is normally used to allow reading access to multiple threads, with a maximum number of threads restricted to $n$.

To verify a program with a semaphore, the semaphore synchroniser should be specified with a resource invariant that describes the permissions protected by the synchroniser. The resource invariant has to be a group, $I(\pi)$, as it is intended to be divided among multiple readers. Exceptionally, in case the number of permits for the semaphore is $n = 1$, the resource invariant does not necessarily need to be splittable. In that case, a single thread (a writer) is allowed to use the resource invariant.

When a thread creates the semaphore, the resource invariant $I(1)$ is transferred from the thread to the semaphore. To access the shared state, the thread should acquire the semaphore, and therewith obtain $1/n^{th}$ part of the resource invariant, i.e. $I(1/n)$. Releasing the semaphore returns this part of the resource back to the semaphore. Therefore, the semaphore may distribute maximum $n$ tickets, to $n$ different readers.

A representative synchroniser class in Java is the class Semaphore. More details about specification and verification of this class can be found in [ABH+14b].

*Barrier* Another synchronisation alternative is a barrier. A barrier is always associated to a group of threads. When one of these threads reaches the barrier, it should stop and wait for all other threads to reach the barrier. When all associated threads have reached the barrier state, they all can proceed with their execution.

From a verification point of view, a barrier is modelled as a redistribution of permissions among the synchronised threads. Threads that hit the barrier give up the permissions they hold, and get a new set of permissions when they leave the barrier. In general, the sum of the permissions of the threads entering the barrier, should be equal to the sum of the permissions that threads hold after the barrier. How this redistribution is done, is defined by a specified resource invariant associated to the barrier.

This verification approach has been introduced by Hobor and Gherghina [HG12] for verifying barriers in PThreads programs. Furthermore, in [BHM14] Blom et al. proposed a verification technique based on the same concept for reasoning about barriers in GPU programs. In Java, a barrier is represented by the CyclicBarrier class; in [ABH+14b] we give more details of how one can specify and verify this class.

**Non-blocking synchronisation** Lock-based synchronisation is a common way to make code safe; however, it is not the most efficient way. Synchronisation basically means forcing threads to wait at certain points in the program. Therefore, this *blocking* or *coarse-grained* synchronisation affects a program's performance. To make optimal use of the multiple processor cores, it is preferred that all of them are busy as much as possible.

A more efficient way to synchronise the code is by using *atomic variables*, i.e., variables that are accessed atomically. Protecting such a variable with a lock-based synchroniser is not needed, because the variable itself is an *atomic synchroniser*: accesses to an atomic variable are treated as if they are protected by an internal lock. This way of synchronisation is called *non-blocking* or *fine-grained*. The programming language Java provides *atomic classes*: an atomic class wraps an atomic variable and exposes three operations to the clients (atomic read, write and compare-and set). Verification of synchronisers using atomics in a permission-based separation logic environment has been investigated by Amighi [ABH14a]. The negative side of non-blocking algorithms is that they are more difficult to implement, and therefore, the performance of the code usually comes at the cost of error-proneness.

This thesis is mainly concerned with verification of programs with blocking

synchronisation.

## 3.3 Some Other Approaches

In addition, we discuss a few other approaches that are commonly known in the area of verification of concurrent programs. These are not directly related with the rest of the thesis; however, at certain points the thesis refers to some of them and compares them with our proposed techniques.

**Rely-guarantee** *Rely-guarantee* logic was developed by Cliff B. Jones [Jon83], before separation logic was introduced. The aim was to tackle the modularity problem of the Owicki-Gries approach. The idea of rely-guarantee reasoning is the following. In addition to the pre- and postconditions $P$ and $Q$, for each thread, one should define a *rely* predicate $R$ to express the transitions that the environment of the thread is allowed to do, and a *guarantee* predicate $G$ that expresses which transitions the thread may execute.

Therefore, a specification of a thread is a quadruple $(P, R, G, Q)$. To prove that a statement $S$ satisfies its specification, written $R, G : \{P\}\ S\ \{Q\}$, one needs to prove that: if $P$ is satisfied in the prestate of $S$, and if we can assume that the environmental threads satisfy $R$, then every atomic execution in $S$ satisfies $G$ and $Q$ holds in the poststate of $S$. To prove parallel composition of threads, each thread should be verified in isolation, and moreover, the rely condition of each thread must be implied by the guarantee conditions of the other threads. The rule for composition of parallel threads is defined as:

$$\frac{(R \vee G_2, G_1) : \{P_1\} S_1 \{Q_1\} \quad (R \vee G_1, G_2) : \{P_2\} S_2 \{Q_2\}}{(R, G_1 \vee G_2) : \{P_1 \wedge P_2\} S_1 || S_2 \{Q_1 \wedge Q_2\}}$$

**Rely-guarantee separation logic** Both rely-guarantee reasoning and separation logic are powerful techniques that take different approaches in reasoning about concurrent programs. With the rely-guarantee technique, the interference between threads might be precisely defined; however, specifications become complicated as reasoning is global, and interference between any two state updates must be considered. On the other hand, the power of separation logic is its local reasoning, but it lacks the advantage of expressing thread interference.

Vafeiadis and Parkinson combine the strength of these two logics in a unique technique, *rely-guarantee separation logic (RGSep)*[VP07]. The state in this logic is split into a *local state*, and a *shared state*. As in rely-guarantee logic, they use rely and guarantee conditions to express how the shared state might

change by describing the interference between threads. However, when there is no thread interference, they reason locally, as done in separation logic.

Another similar approach is the work of Feng et al. [FFS07], who combined both logics into a new *Separated A-G Logic (SAGL)* [2].

**Implicit dynamic frames** Very close to separation logic is the concept of *Implicit Dynamic Frames (IDL)* [SJP09, SJP12]. A predecessor of IDL was *dynamic frames* [Kas06], a method used for verifying object-oriented programs and which aims at first-order logic tool support. The dynamic frames method requires explicit specification of frames of methods. Inspired by separation logic and its *implicit framing*, Smans et al. introduce IDL, an extension of dynamic frames, where the explicit frame was replaced by implicit access tickets.

Later Leino et al., extended the work on implicit dynamic frames with fractional permissions to handle multithreaded programs. Their methodology is implemented in Chalice, a first-order automatic verification tool [LMS09, LM09] for verifying concurrent, object-based programs, written in a specially designed verification language called Chalice. The input Chalice source code is translated into the Boogie intermediate language and given as an input of the Boogie [ByECD$^+$06] back-end.

Both separation logic and implicit dynamic frames have very similar concepts of ownership and permission transfer. They differ in their semantics, as they model the heap in a different way. In [PS12] Parkinson and Summers compare both logics; they provide an equivalent modification of the standard separation logic semantics that also captures the semantics of implicit dynamic frames. Their result shows that a separation logic proof might be encoded in a first-order tool.

**Type systems for safe programming** In 2002 Boyapati et al. [BLR02] develop a new static type system to prevent data races and deadlocks in a multithreaded object-oriented program. Classes in this system are parametrised with an appropriate *protecting mechanism*. For example, such a parameter may express a mutual exclusion lock that protects parallel accesses to the object. A well-typed program, which properly uses the specified mechanism, is data race-free.

To handle deadlocks, the type system imposes each lock in the program to be associated with a *lock level*. Moreover, a partial order should be specified

[2] The abbreviation A-G comes from *assume-guarantee*, which is another name for rely-guarantee

among lock levels. Locks within a lock level may further be ordered using a tree data structure. The type checker verifies that threads acquires locks in a descending order, which ensures deadlock-freedom.

**A programming model for concurrent programs** Bart Jacobs et al. provide another modular verification methodology [JPLS05, JPS+08]. Their technique extends the sequential Spec# programming system [BLS05] and verifies programs written in the Spec# language, which is an extension of C#. The system verifies properties like data race-freedom and deadlock-freedom, and validity of class invariants.

To enable reasoning about data race-freedom, for each object one should explicitly specify whether it is *shared* or *unshared*. Each thread maintains an *access set*, containing all unshared objects that the thread has created, and all shared objects, locked by the thread. A thread is only allowed to access (read or write) fields of objects contained in the set. Data race-freedom is guaranteed by ensuring that access sets of two different threads may not intersect.

Some other approaches for verifying concurrent programs are *Concurrent Abstract Predicates (CAP)* [DYDG+10], iCAP (Impredicative CAP) [SB14] and *linearisability* [Vaf07, Vaf10]. We discussed them briefly in Section 5.8.

## 3.4 Conclusions and Discussions

We showed that multithreaded programs are much more difficult to verify than sequential programs. Primarily, when reasoning about a multithreaded program, one should *think differently*, always taking into account whether other parallel threads may *interfere* with the thread that we reason about. Thread interference is the central problem in concurrency. This has been shown by Owicki and Gries and their first technique for concurrent verification. They extended the Hoare system with a single rule to support parallelism; however, exactly this rule breaks modularity of the whole system and degrades the logic because it requires one to prove interference-freedom.

Furthermore, we presented *separation logic*, a commonly accepted logic for modular reasoning about multithreaded programs. Central element in this logic are *permissions* (or access tickets) tightly coupled with every heap location in the program. This permissions-based concept in separation logic allows one to reason *locally*, avoiding to some extent the problem with thread interference. An improvement of the standard separation logic is *permission-based separation logic*, which allows splitting of permissions and thus, allowing multiple threads

to read the same location simultaneously.

A great asset of separation logic is its ability to guarantee *data race-freedom*. However, this guarantee does not come free of charge. For a program to be verified, it is necessary that it is annotated with permissions, which leads to a significant specification overhead. The positive side, however, is that infering permissions to the program is something that could be done automatically. Ferrara and Müller have addressed this drawback; in [FM12] they propose a static analysis technique, based on abstract interpretation, for automatic inference of access permissions. Their static analyser Sample automatically infers permissions in method pre-and postconditions, as well as in resource invariants.

Multithreaded programming comes together with *synchronisation*. When verifying a program that uses a synchroniser, it is important that the synchroniser is also decorated with specifications. These specifications normally describe the protocol of transferring permissions between the client threads and the synchroniser. They should be general enough so that the synchroniser could be used by client threads in different applications. In [ABH+14b] we address the problem of lack of specified synchronisers; we provide specifications and discuss verification of several most commonly used synchroniser classes in Java.

Separation logic provides an elegant way to solve the problem of thread interference and prove data race-freedom. However, this problem is not entirely avoided when we want to reason about *functional properties* of programs. We showed (see Example 3.4) that specifying the functional behaviour of a method in a concurrent program is troublesome if the method is synchronised internally. This problem is a consequence of thread interference. We discuss this problem extensively and propose its solution later in Chapter 5.

Furthermore, what we have not mentioned in this chapter but is also a considerable challenge in concurrent verification is *reasoning about concurrent invariants*. We leave this discussion for Chapter 4, where we address the problem in more detail.

Finally, we can conclude that verification of multithreaded programs is indeed challenging, but the existing verification techniques give promising results and show that concurrent verification is definitely feasible and worthwhile.

# Part II

# Novel Techniques for Verification of Concurrent Programs

# Chapter 4

# Verification of Concurrent Class Invariants

Invariants are a highly useful feature for the verification of programs. While method contracts are used to describe local program states only (the pre- and poststate of a method), with an invariant we specify a property over the shared state that should be preserved during the entire execution of the program. A verification technique should be able to inspect whether the program behaves correctly such that it does not break the validity of the specified invariants.

In object-oriented programs, invariants are called *class invariants*; they are typically used to express properties about the object's state that should hold throughout the object's life cycle. In the literature, they can sometimes be found under the name *object invariants* [LM04, DFMS08, SDM09][1].

In a sequential setting, the theory about invariants validity is well-developed [Mey97]. There already exist successful techniques for modular verification of class invariants in object-oriented programs [MPHL06, LPX07, DFMS08]. Unfortunately, as shown in Chapter 1, verifying *concurrent invariants*, i.e., invariants in multithreaded programs, is still a challenge. We discussed in Chapter 1 that invalidating an invariant in a state where it is expected to hold is called a *high-level data race*, and formulated the following challenge:

[1] Leino and Müller use the term *object invariants* for invariants that refer to instance fields only [LM04], and the term *static class invariants* for invariants that refer to static class fields [LM05]

*Challenge 1: How to verify that a concurrent program is free of high-level data races?*

In this chapter we address this question with a novel technique for modular verification of concurrent class invariants. Our technique guarantees that a verified program is free of high-level data races. We build on a variant of permission-based separation logic, which ensures that a verified program is also data-race free. To allow modular verification, we adopt the restrictions of Dietl and Müller's ownership-based type system [DM05, DM12]. The technique is illustrated on a suitable object-oriented language. We give a complete formalisation of the verification system, and prove its soundness.

**Outline** The chapter is organised as follows. First in Section 4.1 we revisit the basic technique for verification of class invariants indicating why this technique breaks in a concurrent setting. In Section 4.2 we discuss informally our verification technique: first in Section 4.2.1 we present the class invariant protocol only, without considering modularity; and later in Section 4.2.2 we explain the concepts of the *ownership-based model* and show how we integrate it in our verification system to achieve modularity. In Section 4.3 we provide a complete formalisation of the system: Section 4.3.1 presents the syntax of our language; Section 4.3.2 illustrates the semantics; Section 4.3.3 gives the list of proof rules for reasoning, and Section 4.4 discusses soundness of our system. Concretely, we prove that verified programs are: *partially correct*, *free of data races* and *free of high-level data races*. Finally in Section 4.5 we conclude, discuss related work, and suggest further improvements.

## 4.1 Why does the Basic Theory for Class Invariants Break in a Concurrent Environment?

While in general an invariant is a property that *always* holds, in practice it is often infeasible to maintain the invariant continuously. A practical verification technique should allow breaking of the invariant (otherwise the technique will be too restrictive), as long as this breaking is internal and not publicly visible. Therefore, it is important to distinguish precisely between *visible states*, i.e., states in which it is really important for an invariant to hold; and *hidden states*, i.e., states in which it might be allowed for an invariant to break. Thus, a verifier should monitor the validity of the invariants in the visible states only.

The main principles of the theory for class invariants in sequential programs (as already discussed in Chapter 2) is based on the following rules:

- *(Visible states)* Every pre- and poststate of a non-helper method, or a poststate of a constructor in the program is a visible program state;
- *(Assumption)* In a prestate of a method call (excluding constructors), invariants may be assumed by the callee. In a poststate of a method call (including constructors), invariants may be assumed by the caller.
- *(Obligation)* In a prestate of a method call (excluding constructors), invariants must be proved by the caller. In a poststate of a method call (including constructors), invariants must be proved by the callee.

This strategy guarantees that class invariants always hold in any visible program state. Further in this chapter, we avoid the distinction between helper and non-helper methods. The simplified language that we use does not contain explicit visibility modifiers (**private**, **public**, **protected**); thus, we consider every method in the program to be a non-helper method.

Unfortunately, this theory for verifying sequential invariants does not satisfy our needs because it suffers from the following two drawbacks: i) it does not allow modular verification; and ii) it breaks in the presence of multiple threads.

*Modularity problem* The problem with class invariants and modularity was already discussed in Chapter 2, page 26. In short, the obligation stated above can not be established by a modular verification technique: it requires one to prove validity of every class invariant in the poststate of a method $m$; however, in the context (class) where the method $m$ is defined, we do not know *all* class invariants. Therefore, it might happen that the method invalidates an invariant defined in a class of which we are not aware that it exists.

The modularity problem arises because of the nature of modular software and thus, it is also relevant for sequential programs. This problem has already been investigated and answered by several sequential techniques for modular verification of object-oriented software [MPHL06, LPX07, DFMS08].

*Concurrency problem* What is more interesting for us is that the standard verification technique can not be carried over directly to the setting of multi-threaded programs. Due to possible interference between parallel threads, any state in the program might be *visible*. More concretely, if the program state is an internal method state for one thread, it might be a pre- or poststate of a method

for another parallel thread. Therefore, a technique for verification of concurrent invariants should take a different approach with respect to determining visible states.

*Our solution* Our verification technique addresses both problems explained above. The main novelty of the technique is the way how we tackle the problem of concurrency and thread interleavings. We define a protocol that allows a thread to break a class invariant, under the condition that this invalidating is only temporary and is not visible for any other parallel thread. To answer the modularity problem, we integrate the laws from an existing ownership-based type system [DM05, DM12]. Using this type system, only slight modifications were needed to adapt our verification technique and make it modular.

## 4.2 The Concepts of Our Methodology

This section gives only a conceptual (informal) understanding of our technique, while the system is formalised later in Section 4.3. Here, we present our approach from two different aspects. First, in Section 4.2.1 we address the concurrency problem. We discuss the strategy for verifying concurrent invariants without worrying about modularity. We describe our *invariant protocol*, which defines: i) when can a class invariant be assumed to hold? ii) when is a class invariant allowed to be broken? or iii) when must a class invariant be proven to hold? Second, in Section 4.2.2 we discuss how to integrate the ownership-based type system to allow modular verification.

### 4.2.1 Class Invariant Protocol

We consider that class invariants do not express properties over static class fields, but only over instance fields. In an object-oriented program, the instance fields are stored in the heap memory, we refer to them as shared locations. A field $x$ of an object instance $o$ is denoted with $o.x$. When $o$ is the current object, i.e., *this*, it is usually omitted. A class invariant $I$ defined in a class $C$ is always associated to a particular object $o$ of class $C$, written $o.I$. We call the set of locations referred to by an invariant $o.I$ the *footprint of* $o.I$, denoted $fp(o.I)$. This is formally defined in Section 4.3.

**Permissions apart** In Section 1.3.1, we gave the intuition of a high-level data race, via the example of the class Point. Now we take a closer look at this

```
class Point{
 int x;  int y;
 Lock lx;  //lx_Inv:  x ↦¹ _
 Lock ly; //ly_Inv:  y ↦¹ _
 Invariant I: x+y≥0;
//...

                        void move(){
                          acquire lx;
                            x = x - 1;
                          release lx;
                          acquire ly;
                            y = y + 1;
                          release ly;
                        }
                       }
```

Listing 4.1: Locks and class invariants

example from the point of view of separation logic. Listing 4.1 presents a class Point that contains two instance fields x and y. Write permissions to x and y are protected by two separate locks, lx and ly, respectively. A class invariant I describes that the relation x+y≥0 should constantly be maintained.

Note that the class invariant is not a valid expression in standard separation logic: it contains heap locations (x and y) that are not framed by a positive permission. We could rewrite the invariant into a framed expression: $x \overset{\pi}{\mapsto} vx * y \overset{\pi}{\mapsto} vy \Rightarrow vy+vy \geq 0$ which describes: if the thread that we reason about has positive permissions to x and y, the relation $x+y \geq 0$ holds. However, this seems nonintuitive, because we specify a class invariant as a general property that should hold over the global state, and thus, we do not think in terms of reasoning about local threads.

Another alternative is to express the invariant properties via resource invariants: thus, a resource invariant will express both permissions to locations and a functional property over these locations that must hold when the lock is free. This alternative, however, is not always suitable. First, a resource invariant is always associated to a synchroniser. Second, it requires that the same lock protects all footprint locations of the invariant. For example, for the program in Listing 4.1, this would mean that the resource invariant associated to a lock lx (or ly) must store permissions for both x and y. Therefore, if a thread increases the value of x only, it has to obtain permissions for both x and y.

The discussion above shows that there is a very close relation between locks and class invariants. We find that a convenient way to deal with this is to set permissions apart from the functional properties: class invariants can express functional properties only, while resource invariants are still responsible for expressing permissions to locations. In particular, we force class invariants to be non-framed expressions: they can contain locations, but are not allowed to con-

tain permissions. This means that we deviate from standard separation logic, but still ensure data race-freedom in the program.

**Assuming a class invariant** In sequential programs, a class invariant may be assumed to hold in the prestate of a method and must be proved to hold in the poststate of a method. Obviously, the standard verification technique does not apply here. When we reason about a method, we reason thread-locally; thus, if in the method's pre- and poststate the current thread does not hold permissions to the footprint locations of the invariant, neither can we assume that the invariant holds in the prestate, nor can we prove that it holds in the method's poststate.

To control whether a class invariant is expected to hold in a given state, to every invariant $o.I$, we associate a special abstract predicate $\mathsf{holds}(o.I, 1)$. When a thread holds this predicate, it can assume that the invariant is valid in that state. The predicate is a group, i.e., a splittable predicate. Therefore, it might be split and distributed among different threads by using the following equivalence:

$$\mathsf{holds}(o.I, \pi) \mathrel{*\!\!-\!\!*} \mathsf{holds}(o.I, \pi/2) * \mathsf{holds}(o.I, \pi/2)$$

Any thread that holds a fraction of the predicate $\mathsf{holds}(o.I, \pi), \pi > 0$ can assume the validity of the invariant. Additionally, when a thread holds the full predicate $\mathsf{holds}(o.I, 1)$, it has the right also to break the class invariant.

In a way, the $\mathsf{holds}(o.I, \pi)$ predicate serves as an additional protection level that controls how threads access the shared state that the class invariant refers to. The predicate $o.f \overset{\pi}{\mapsto} \_$ is still required to give a thread read or write permission to access a shared location $o.f$, and therefore takes care to ensure data race-freedom. Additionally the $\mathsf{holds}(o.I, \pi)$ predicate controls that updates to multiple footprint locations are done in one step, such that a high-level data race does not occur.

In particular, the $\mathsf{holds}(o.I, \pi)$ predicate guarantees that: i) a class invariant $o.I$ is stable and all threads that hold a predicate $\mathsf{holds}(o.I, \pi)$ may rely on $o.I$'s correctness; or ii) *at most one* thread has the predicate $\mathsf{holds}(o.I, 1)$ and therewith, the right to break the invariant, while no other thread may assume correctness of $o.I$.

**Breaking a class invariant** When a thread $t$ assigns a location $p.f$ that belongs to the footprint of a class invariant $o.I$, i.e., $p.f \in \mathit{fp}(o.I)$, the class invariant may possibly be broken. At that moment, no other thread has access

```
    {holds(I, 1) * x ↦¹ vx * y ↦¹ vy}
2   {holds(I, 1) ∧ I *  x ↦¹ vx * y ↦¹ vy} // the validity of I may be assumed
    {holds(I, 1) ∧ x+y ≥ 0 * x ↦¹ vx * y ↦¹ vy}
4       //@ unpack I {        // trades holds predicate for unpacked predicate
    {break(I, 1) ∧ x+y ≥ 0 * x ↦¹ vx * y ↦¹ vy}
6         x = x − 1;       // the invariant I is broken
    {break(I, 1) ∧ x+y ≥ -1 * x ↦¹ vx-1 * y ↦¹ vy}
8         y = y + 1;       // the invariant I can now be reestablished
    {break(I, 1) ∧ x+y≥0 * x ↦¹ vx-1 * y ↦¹ vy+1}
10  {break(I, 1) ∧ I * x ↦¹ _ *  y ↦¹ _}
      //@ }           // trades unpacked predicate for holds predicate
12  {holds(I, 1) * x ↦¹ _ * y ↦¹ _}
```

Listing 4.2: Unpacked segment of a class invariant

to the location $p.f$, since the thread $t$ holds the complete write permission. Therefore, we can say that at this state, the invariant is not publicly visible, but is accessible only internally by the thread $t$. To avoid high-level data races, it is important that the invariant stays non-visible until the thread $t$ makes all other necessary updates and ensures that the invariant is again re-established.

We require one to explicitly specify the segment in the program where a class invariant property might be violated. To this end we introduce the specification command unpack $o.I\{c\}$, which marks the program code $c$ as an *unpacked segment of* $o.I$. Within the unpacked segment of $o.I$, the predicate holds$(o.I, 1)$ is exchanged for a new predicate break$(o.I, 1)$ called a *breaking predicate*. The thread that unpacks the invariant becomes a *local thread* to the unpacked segment. Holding the break predicate, this thread has a license to break the class invariant $o.I$, i.e., to assign to the locations from the invariant's footprint. Before the end of the unpacked segment, the thread must re-establish the validity of the class invariant $o.I$. The break$(o.I, 1)$ predicate is then exchanged again for the holds$(o.I, 1)$ predicate .

**Example 4.1.** *Listing 4.2 illustrates the use of an unpacked segment. The program code is a segment from the* move() *method from the class* Point *(see Listing 4.1), modified in a way such that permissions to both* x *and* y *are already obtained before the unpacked segment and not released within the segment. This change is important to be able to verify the program.*

```
    {holds(I, 1) * y ↦¹ vy}
2   {holds(I, 1) ∧ x+y ≥ 0 * y ↦¹ vy}
      //@ unpack I {
4   {break(I, 1) ∧ x+y ≥ 0 * y ↦¹ vy}
        y = y + 1;      // the invariant I can now be reestablished
6   {break(I, 1) ∧ x+y ≥ 1 * y↦¹ vy+1}
    {break(I, 1) ∧ I * y ↦¹ vy+1}
8     //@ }
    {holds(I, 1) * y ↦¹ vy+1}
```

Listing 4.3: Accessing only part of the invariant footprint

*At line 1, the current thread holds both a complete* holds *predicate for the class invariant I and write permissions to* x *and* y*. The* holds *predicate gives the thread right to assume that the invariant is correct, and to start an unpacked segment (line 4); thereafter it trades this predicate for the* break *predicate. Thus, within the unpacked segment, no other thread may hold a fraction of the* holds *predicate and assume the validity of the class invariant. Using the write permissions, together with the* break *predicate, the thread updates the values of* x *and* y *(lines 6 and 8). After both updates, the class invariant is again re-established, and the unpacked segment may end. Thereafter, the thread obtains back the* holds *predicate, which indicates that the invariant is again stable.*

**When only part of the footprint is accessed** As mentioned above, the holds$(o.I, \pi)$ predicate controls the access to the locations from the footprint of $o.I$. When the current thread holds a positive fraction of holds, no thread might hold a fraction of the break predicate, which is necessary to write on any of these footprint locations. All $o.I$'s footprint locations are then stable. Similarly, a complete break predicate guarantees stability of the footprint locations.

Therefore, an expression in our language is framed when for every occurring location $p.f$ it contains: a positive permission to $p.f$, a positive fraction of the holds or a complete break predicate of every class invariant that refers to $p.f$. This makes our approach flexible and permissive, because it allows a thread to break an invariant without holding all permissions associated to the invariant property. This is shown on the example below.

**Example 4.2.** *Assume we want to increase only the value of* y *from the class* Point*, see Listing 4.3. As* y *is a footprint location of a class invariant, it is*

*necessary that the invariant is first unpacked. Before the unpacked segment, the current thread holds the* holds(I, 1) *predicate, which gives the right to assume that the invariant* x+y≥0 *holds, line 1. The location* x *in line 1 is not framed by a positive permission to* x*, but is stable because it is framed by the* holds(I, 1) *predicate. This allows us to prove that the invariant holds at the end of the unpacked segment (line 7), as we can assume that nobody has changed the value of* x*.*

**Restrictions to an unpacked segment** As mentioned above, within the unpacked segment of a class invariant, no other thread may obtain the holds predicate and assume that the invariant holds. However, this is not sufficient to ensure high-level data race-freedom. What we really want is to forbid the other threads to observe the changes that are made within the unpacked segment before the invariant is reestablished. We say that a location that is modified within the unpacked segment is in a critical state. More precisely:

**Definition 4.1.** *(Critical state of a location) Let* $o.I$ *be an invariant,* $p.f$ *a location, such that* $p.f \in fp(o.I)$*, and let* $p.f$ *be assigned inside an unpacked segment of* $o.I$*. Then, any program execution state between the assignment and the end of the unpacked segment is a* critical state for $p.f$ with respect to this unpacked segment*.*

**Definition 4.2.** *(High-level data race) A high-level data race occurs in a state that is critical for a location* $p.f$ *with respect to an unpacked segment, and in which a non-local thread of this segment accesses* $p.f$*.*

To prevent high-level data races, a location in a critical state must not be publicly exposed. Therefore, within an unpacked segment we forbid the running thread to release permissions and make them accessible to other threads. Concretely, within an unpacked segment, we allow only so-called *safe commands*, i.e., commands that exclude any lock-related operation (e.g., acquiring or releasing a lock). This means that all permissions used in the unpacked segment must be obtained before the segment begins.

We also define the notion of a *safe method*, a method composed of safe commands only. Such a method is specified with the optional modifier **safe**. A call to a safe method is also a safe command and thus is allowed within the unpacked segment. Threads with a safe run() method are also called *safe threads*. Forking a safe thread within the unpacked segment is allowed under the condition that the thread must be also joined within the unpacked segment. We

```
    void move(){
2    acquire lx;
   //@ unpack I{  //I must be unpacked before assigning to x
4      x=x-1;
     release lx; //invalid call, must happen after the unpacked segment
6    acquire ly; //invalid call, permissions must be obtained before unpacking
       y = y+1;
8  //@ }
     release ly;
10   }
```

Listing 4.4: Restrictions to an unpacked segment

say that these threads are also *local to the segment*. For the formal definition of safe commands we refer to Section 4.3.

The breaking predicate might be shared among all local threads of the unpacked segment, and thus, they might all update different locations of the invariant footprint in parallel. This is why the break predicate is a group, and thus the following axiom holds:

$$\mathsf{break}(o.I, \pi) \mathrel{*\!\!-\!\!*} \mathsf{break}(o.I, \pi/2) * \mathsf{break}(o.I, \pi/2)$$

**Example 4.3.** *Listing 4.4 presents the move() method from Listing 4.1. It shows that the method cannot be verified, because both acquiring and releasing a lock must not occur within the unpacked segment. It is important that permissions for all locations that are assigned in the unpacked segment are obtained before the segment starts and are not released until the end of the segment.*

**Object initialisation** Initialisation of a new object $o$ should end in a state in which the object $o$ is valid, which means that all class invariants $o.I$ hold. In our language, object initialisation, i.e., the execution of the constructor $o = new\ C(v_1, ...v_n)$, is divided into three steps:

i) *object construction*, i.e., executing the command $o = new\ C$, which creates an empty object $o$ where: all instance fields of $o$ have a default value and all invariants $o.I$ are unpacked. Thus, after the object is created, the current thread obtains a write permission for each field of $o$, and the $\mathsf{break}(o.I, 1)$ predicate for each class invariant $o.I$;

ii) *object initialisation*, i.e., executing the method $o.C(v_1, ...v_n)$. Importantly,

the constructor methods are special methods that must be composed of safe commands only. Normally, they are used to initialise objects fields only.

iii) *object packing*, i.e., executing the command pack $o$. This is a specification-only command used to pack all class invariants of the object $o$. For every class invariant $o.I$ it exchanges the break$(o.I, 1)$ predicate for a predicate holds$(o.I, 1)$. Therefore, packing of the object ensures that the newly initialised object $o$ is valid.

**Lock management** In our language, any object may be used as a lock. However, when the new object is created, it is still not ready to be acquired. We say that the new object is a *fresh* lock . A thread must first *commit* the lock (we use a command commit $o$); thereafter the lock becomes *initialised* and any thread may acquire it.

We discussed in Section 3.2.3 that locks are associated to a resource invariant, a special predicate that describes the locations that are protected by the lock. The resource invariant is transferred: i) on *commit*: from the current thread to the lock; ii) on *acquire*: from the lock to the thread; and iii) on *release*: from the thread back to the lock. When acquiring a lock $o$, the thread also obtains a predicate locked$(o)$ which indicates that it is the owner of the lock. When releasing the lock, the predicate is returned back to the lock.

**Example 4.4.** *Listing 4.5 presents the complete proof outline of the class Point. The definitions of the state predicate (line 4) and the resource invariant predicate (line 5) describe that the Point object will also be used as a lock that protects both locations x and y. The lock also stores the predicate holds(I, 1). Of course, this is not necessary, but it is a natural choice, because we need this predicate only when we own the lock. Initialisation of the fields x and y is done within the constructor (lines 11 and 13).*

*The Client class in Listing 4.6 initialises a new Point object p. We illustrate separately the three steps of the initialisation: i) an empty object p is created in line 3, which gives the current thread write permissions to p.x and p.y, a predicate fresh(p), and the* break *predicate for the invariant p.I; ii) the constructor is executed in line 5; and iii) the object is packed in line 7, which exchanges the* break *for the* holds *predicate. After initialisation, the object is a fresh lock. In line 9, the object p is committed. Thereafter, the resource invariant is consumed by the lock, and the fresh(p) predicate is converted into a predicate initialised(p), which is necessary for the client to acquire the lock.*

*It is important to note that we define the initialised(p) predicate such that it can be created once, i.e., after committing the object p, but can later be duplicated*

```
class Point{
 int x;   int y;
//@ Invariant I: x+y≥0;
//@ pred state (int vx, int vy) = x ↦¹ vx * y ↦¹ vy;
//@ pred res_inv = holds(I, 1)  * state(_, _);

//@ requires state(_, _);
//@ ensures state(vx, vy);
Point(int vx, int vy){
{x ↦¹ _ * y ↦¹ _}
  x = vx;
{x ↦¹ vx * y ↦¹ _}
  y = vy;
{x ↦¹ vx * y ↦¹ vy}
{state(vx, vy)}
}

//@ requires initialised(this)
//@ ensures  true;
void move(){
  acquire this;
{locked(this) * holds(I, 1) * state(vx, vy)}
{locked(this) * holds(I, 1) ∧ I * x ↦¹ vx * y ↦¹ vy}
{locked(this) * holds(I, 1) ∧ x+y≥0 * x ↦¹ vx * y ↦¹ vy}
    //@ unpack I {
{locked(this) * break(I, 1) ∧ x+y≥0 * x ↦¹ vx  * y ↦¹ vy}
      x = x -1;
{locked(this) * break(I, 1) ∧ x+y≥-1 * x ↦¹ vx-1  * y ↦¹ vy}
      y = y + 1;
{locked(this) * break(I, 1) ∧ x+y≥0 * x ↦¹ vx-1  * y ↦¹ vy+1}
{locked(this) * break(I, 1) ∧ I * x ↦¹ _  * y ↦¹ _}
    //@ }
{locked(this) * holds(I, 1) * state(_, _)}
  release this;
 }
}
```

Listing 4.5: Complete proof outline of the Point class

```
   class Client{
2    void createPoint(){
       Point p = new Point;
4   {p.state(0, 0) * fresh(p) * break(p.I, 1)}
       p.Point(2,3);
6   {p.state(2, 3) * fresh(p) * break(p.I, 1) ∧ p.x+p.y ≥ 0}
       pack p;
8   {p.state(2, 3) * fresh(p) * holds(p.I, 1)}
       commit p;
10  {initialised(p)}
    {initialised(p) * initialised(p)}
12     p.move();
    {initialised(p)}
14    }
    }
```

Listing 4.6: Creation of a *Point* object

*(this is explained later on page 99). This is why there is no need for the move() method in class Point to explicitly return this predicate in the postcondition. Instead the client can copy the predicate any time (see line 11), and the copy can be used after the call to the method p.move().*

**Using a class invariant** Any positive fraction of the holds$(o.I, \pi)$ predicate is sufficient to ensure that the invariant $o.I$ holds. Therefore, when a client thread holds this fraction it can use the validity of the invariant expression to verify other properties of the client class. We illustrate this on an example.

**Example 4.5.** *Listing 4.7 presents a client class DrawPoints. The client thread creates initially a Point object p for which the invariant p.I holds (p.x + p.y ≥0), and obtains the* holds *predicate at line 5. The thread then forks two parallel threads (lines 9 and 14), passing each of them a reference to p and a fraction 1/2 of the* holds *predicate.*

*The thread class Task is presented in Listing 4.8. The run method shows that each thread has a task to create a sequence of new points at specific locations calculated from the location of the initial point p. This class also contains its own class invariant I. The use of a class invariant is shown at line 24. The thread holds fractions of the* holds *predicate for both invariants p.I and* **this**.*I. This guarantees that the invariants hold: p.x + p.y ≥0, k1≥ 0 and k2≥0. These expressions are then used to verify that the location of the new point (x, y) is*

```
   class DrawPoints {
2
    void createPoint(){
4    Point p = new Point (0,0);
   {p.state(0,0) * holds(p.I, 1)}
6
     Task t1 = new Task(p,1,0);
8  {p.state(0,0) * holds(p.I, 1) * t1.state(p,1,0) * holds(t1.I, 1)}
     fork t1;
10 {p.state(0,0) * holds(p.I, 1/2)}

12   Task t2 = new Task(p,0,1);
   {p.state(0,0) * holds(p.I, 1/2) * t2.state(p,0,1) * holds(t2.I, 1)}
14   fork t2;
   {p.state(0,0)}
16   join t1;
   {p.state(0,0) * holds(p.I, 1/2) * t1.state(p,1,0) * holds(t1.I, 1)}
18   join t2;
   {p.state(0,0) * holds(p.I, 1) * t1.state(p,1,0)
20       * holds(t1.I, 1) * t2.state(p,0,1) * holds(t2.I, 1)}
   }
```

Listing 4.7: Splitting and distributing the *holds* predicate

*also valid, i.e., x+y≥0. Therefore, the initialisation of the new object p1 can be successfully verified.*

*The example shows that several parallel threads can use the validity of a given class invariant at the same time. None of the two threads in the example holds positive permissions to the locations p.x and p.y, but they both hold a fraction of the holds(p.I, pi) predicate, and this ensures that the relation p.x + p.y ≥0 holds and is stable.*

**Summary** We presented a new technique for verifying class invariants in concurrent programs. The class invariant protocol can be summarised via the following four rules:

**R1** (*Assuming*) A thread may assume (use) a class invariant $o.I$ if it holds the predicate $\mathsf{holds}(o.I, \pi), \pi > 0$.

**R2** (*Breaking*) A thread may write on a location $p.f$ if apart from holding a write permission to $p.f$, it holds a breaking predicate $\mathsf{break}(o.I, \pi), \pi > 0$

```
class Task {
  Point p;
  int k1; int k2;
//@ Invariant I: k1 ≥0 ∧ k2 ≥0;
//@ pred state(vp, vk1, vk2) = p ↦¹ vp * k1 ↦¹ vk1 * k2 ↦¹ vk2;

//@ requires state(_, _, _);
//@ ensures state(p, k1, k2);
  Task(Point p, int k1, k2){
    this.p = p;
    this.k1=k1;
    this.k2=k2;
  }

//@ requires state(_, _, _) * holds(p.I, π) * holds(this.I, π1);
//@ ensures state(_, _, _) * holds(p.I, π) * holds(this.I, π1);
 void run(){
  int i = 0;
  Point p1;
  while (i ≤ 10) {
{state(_, _, _) * holds(p.I, π) ∧ p.x+p.y ≥0 * holds(this.I, π1) ∧ k1≥0 ∧ k2≥0}
    int x = p.x+k1·i;
    int y = p.y+k2·i;
{state(_, _, _) * holds(p.I, π) * holds(this.I, π1) * x+y ≥0}
    p1 = new Point(x,y);  // the invariant of p1 can be verified
     i = i+1;
   }
  }
}
```

Listing 4.8: Using a class invariant for verifying a client class

for each class invariant $o.I$ that refers to $p.f$, *i.e.*, $p.f \in fp(o.I)$.

**R3** (*Reestablishing*) A class invariant $o.I$ must have been reestablished when pack $o.I$ is executed.

**R4** (*Exchanging predicates*) The pack $o$ command produces a predicate holds$(o.I, 1)$ for every class invariant $o.I$; within an unpacked segment of $o.I$, the holds$(o.I, 1)$ predicate is exchanged for break$(o.I, 1)$.

The presented verification technique is flexible and permissive, because it allows a thread to break a class invariant without holding all permissions associated to the invariant property. Moreover, it allows a thread to use the correctness of the class invariant without holding any permissions to these locations. Importantly, the technique guarantees that:

i) a verified program is free of high-level data races; and

ii) a class invariant holds any time when it is in a packed state.

Later in Section 4.3 we formalise these properties and prove them sound.

### 4.2.2 Modular Verification

The approach discussed so far is not modular. In particular, in the prestate of the assignment to a location $p.f$, rule **R2** requires the break predicate for all invariants that refer to $p.f$. This is where the modularity breaks, because in the context (class) where the assignment happens, not all invariants in the program are known. The assignment of $p.f$ may therefore break a class invariant that is defined in another class, of which we do not know that it exists.

To address this problem, we employ the well-known *ownership model* [DM05], which is often used for modular verification of class invariants [MPHL06, BDF+04, LPX07, DM12]. Below we discuss the concepts of the ownership model and we explain how we integrate this model in our reasoning system.

#### Ownership model

The idea of the ownership model is to increase control over object references in the program. It forces all objects to be organised in a structural way and it applies certain restrictions to the operations applicable to each object reference.

In particular, each object is required to respect the concept of *ownership topology*, where objects are organised in a hierarchy. Each object has exactly one *owner*, either another existing object on the heap, or the *root* of the tree (no real object). We say that object $o_1$ is a *transitive owner* of object $o_2$ if $o_1$ is an owner of $o_2$ or $o_1$ is an owner of another $o_2$'s transitive owner.

Such an ownership topology can be used to define different encapsulation disciplines that set some restrictions on the object references and interactions between objects. We use the *owner-as-modifier* discipline, which restricts write accesses (while read-only access is always allowed), requiring that: any object modification must be initiated by the object's owner. The ownership type system imposes the following rule:

**RO** When an object $o$ is modified within a method $m$, for each transitive owner $p$ of the object $o$, the path to $m$'s invocation includes a method invocation with $p$ as a receiver object.

### Universe type system

To enforce the ownership model in a program, several type systems have been implemented [AKC02, VB99, DM05]. A well-typed program satisfies the requirement **RO**. Another alternative is to enforce this model dynamically, as done in the work of Leino and Müller [LM04]. A dynamic approach has advantages over a type system: while a type system statically determines the ownership hierarchy, a dynamic approach allows this hierarchy to be dynamically changed. In contrast, a dynamic approach requires more specifications in the program than a type system. We build our verification technique on the *Universe type system* [DM05], which is discussed below.

*Reference types* An object reference in the universe type system has a type: *rtype* $C$; where $C$ is the class of the referred object, and *rtype* is a special *reference type*, which represents the relationship of the referred object with the current this object. Concretely, *rtype* is a modifier from the set {rep, peer, self, readonly}, where:

- peer indicates that the referred object has the same owner as the current this object;
- rep specifies that the referred object is owned by the this object;
- self stands for references that point to the this object; and
- readonly means any other relation, or the relation is unknown.

A newly created object may be either owned by the current object, or have the same owner as the current object. Thus, we can create a new object assigning it either a modifier rep or peer. For example, if within the context of an object $o$ of class $C$, a new object $p$ is created,

```
rep Point p;
p = new rep Point;
```

$p$ will be added to the objects hierarchy with the current object ($o$) as an owner. The modifier peer is a default modifier; thus if the program is implemented without using any reference types, all heap objects will be structured as peer objects with a common owner, i.e., the root of the tree.

*Assignment* To assign to a variable, the standard type rule also applies: *both the left and the right-hand side of the assignment have to agree on their types.* For example, if in addition to the code above, we add:

```
peer Point p1;
p1 = p;    // invalid, types do not match
```

the type rule is broken, because the types of $p1$ and $p$ in the context of the $o$ object are different.

*Viewpoint adaptation* When an object changes its context via accessing object field, or transfer as a method parameter, the type of the new reference needs to be adapted in the new context. This recalculation is done by applying the *viewpoint adaptation* function, $\triangleright : (rtype, rtype) \mapsto rtype$, defined below:

$$r_1 \triangleright r_2 = \begin{cases} r_2 & \text{if } r_1 = \mathsf{self} \\ \mathsf{rep} & \text{if } r_1 = \mathsf{rep},\ r_2 = \mathsf{peer} \\ \mathsf{peer} & \text{if } r_1 = r_2 = \mathsf{peer} \\ \mathsf{readonly} & \text{otherwise} \end{cases}$$

For example, if $o$ has the reference type rep (in the context of the this object), and the reference type of $f$ is peer in the context of $o$, then the reference type of $o.f$ (in the context of this) is $\mathsf{rep} \triangleright \mathsf{peer} = \mathsf{rep}$. Therefore, the object that $o.f$ refers to is also owned by the current this object.

*Restrictions to read-only references* The idea of using reference types for every object reference, is to identify the position of the referred object in the object hierarchy and to impose certain restrictions in the way in which this object may interact with other objects. Expectedly, read-only references allow read access only. Concretely, the following rules are defined:

- assigning to a field $o.f$ is allowed only when the reference type of $o$ is different from readonly.
- a method call $o.m(\overline{v})$ is allowed when the reference type of $o$ is different from readonly, or $m(\overline{v})$ is a *pure* method, i.e., a method without side-effects.

In this way, each object has the control over all updates that happen of all transitively owned objects. A well-typed program in the Universe type system satisfies the rule **RO** defined above.

### Ownership-based verification technique

The Universe type system, and the ownership-based concepts in general, provide a common and natural way of building *layered* software. For example, the *Point* class may be used for building a *Line* data structure, and a *Line* may be used as an element of a *LineSet*, i.e., a more complex set data structure. Naturally, every *Line* object will be owner of the *Point* objects it refers to, and each *LineSet* object will be owner of each *Line* it is built of.

Relying on the ownership-based concept, we can adapt the rules of our verification technique to provide modularity. First, we allow assignments to fields of the *this* object only. This is not a real restriction, but a requirement for a proper encapsulation policy. Then, we restrict the definition of a class invariant $o.I$ such that we allow it to express properties only over locations that are *owned by* $o$, i.e., locations of the same object, $o.f$, or locations of objects transitively owned by $o$. Therefore, a class invariant in the *Line* class can specify some requirements about the owned *Point* objects, but not vice versa. We call these *admissible class invariants*.

**Definition 4.3.** *(Admissible class invariant) A class invariant $o.I$ is admissible if it expresses properties over fields $p_1.p_2...p_n.f$, where $n \geq 1$, $o == p_1$ and $p_i$ is a rep field in the class of $p_{i-1}$ $(i = 2..n)$.*

From Definition 4.3, we observe the following: given a location $p.f$, a class invariant $o.I$ may refer to $p.f$ only if $o == p$ or $o$ is a transitive owner of $p$. Therefore, before assigning to a location $p.f$ we must ensure that:

i) any class invariant $p.I$ that refers to $p.f$ is unpacked; and

ii) any class invariant $o.I$ that refers to $p.f$ such that $o$ is a transitive owner of $p$ is unpacked;

The first requirement is easy to check. Before the assignment to $p.f$ it is sufficient to inspect whether all class invariant from the class of $p$ are unpacked. The second requirement is not trivial to check, because in the context where the assignment happens we are not aware of the transitive owners of $p$. However, having the ownership model, as stated by the rule **RO**, we know that an assignment to $p.f$ must be preceded by a method call where $o$ is a receiver. To allow modular verification, the check that the invariant $o.I$ is unpacked should therefore be a requirement of the method call where object $o$ is a receiver. More precisely, we replace the rule **R2** (see page 76) with the following two rules:

**R2'** To write on a location $p.f$, apart from holding a write permission to $p.f$, the thread has to show that it holds a predicate $\mathsf{break}(p.I, \pi)$ $(\pi > 0)$ for every class invariant $p.I$ that refers to $p.f$, i.e., $p.f \in fp(p.I)$;

**R2"** To invoke a method $o.m(\overline{v})$ that assigns to a field $p.f$, (apart from the requirement that the precondition of the method holds), the thread has to show that for every class invariant $this.I$ that refers to $p.f$, it holds the predicate $\mathsf{break}(this.I, \pi)(\pi > 0)$.

To establish **R2"**, the contract of the called method $m$ should provide information to the caller about the locations it assigns to. Since we use permission-based separation logic we can identify the locations assignable by $m$ from the precondition expression of method $m$, i.e., $m_{pre}$; this is the set of locations for which $m_{pre}$ requires a write permission, denoted by $writeLocs(m_{pre})$ (see Appendix B). The permission $\pi$ might also be obtained by acquiring a lock within the method $m$. However, in practice this should not happen, as explained below. Assume that within $m$ a lock is acquired to obtain permission to $p.f$, then the location $p.f$ is modified and therewith the validity of the class invariant $\mathsf{this}.I$ is broken. This means that the invariant $\mathsf{this}.I$ must be unpacked. The object $o$ cannot unpack an invariant of its owner: thus, $\mathsf{this}.I$ must have been unpacked before the method call. However, in the unpacked segment (and thus in $m$) lock-related commands are not allowed, but all permissions required in the segment must be obtained before the unpacked segment starts. This contradicts the assumption.

Therefore, it is reasonable to set a rule in the language that forbids a class invariant $o.I$ to refer to locations that are protected by locks transitively owned by $o$. In practice, as explained above, this rule does not bring a notable restriction: even if there is such a lock, it must be acquired at the layer of the object $o$. Concretely, we define the following rule:

**TR1** $\forall I \in inv(C);\ \forall f \in relFld(C);\ fld(C, I) \cap fldResInv(classOf(C, f)) = \emptyset$

The rule is translated as: for any class invariant $I$ from the set of invariants defined in a class $C$, i.e., $inv(C)$, and any field $f$ from the set of *relevant* fields of $C$, i.e., $relFld(C)$, the set of fields that appear in $I$, i.e., $fld(C, I)$, is disjoint from the set of fields that appear in the resource invariant definition in the class of $f$, i.e., $fldResInv(classOf(C, f))$. A field $f$ is *relevant* to a class $C$ if it may be expressed as a $p_1.p_2.,...p_n.f$, where $p_1$ is a $\mathsf{rep}$ field defined in $C$, and $p_i$ is a $\mathsf{rep}$ or $\mathsf{peer}$ field in the class of $p_{i-1}, i = 2..n, n >= 1$. The auxiliary functions used in the rule are defined in Appendix B.

As discussed above, the Universe type system restricts assignments to fields $o.f$ and calls to non-pure methods $o.m(\overline{v})$ if the reference type of $o$ is readonly. In addition to this, we add the following typing rule, which is intuitive and important for the soundness of our system:

> **TR2** i) commands acquire $o$, release $o$, commit $o$, pack $o.I$, unpack $o.I$ are allowed only when the reference type of $o$ is different than readonly. ii) assignment to a field $o.f$ is allowed only when the receiver $o$ is the *this* object.

**Example 4.6.** *We illustrate the modular verification on the example in Listings 4.9 and 4.10. The class* Line *represents a line segment between two points,* p1 *and* p2*. A class invariant* I *is defined stating that a* Line *object is valid when the absolute difference* |p1.x - p2.x| *is greater than* 5*. The invariant is admissible, because both* p1 *and* p2 *references have* **rep** *reference type.*

*To satisfy rule* ***TR1****, it is important that locations* p1.x *and* p2.x *are not protected by locks that are transitively owned by the* Line *object. We choose for example the following strategy: a* Line *object is used as a lock, which protects the fields* x *and* y *of both owned objects* p1 *and* p2*. The class of the objects* p1 *and* p2 *is* LinePoint*, see Listing 4.10. It mainly differs from the class* Point *in Listing 4.5 in the locking policy: the* move() *method in class* LinePoint *uses* external *synchronisation (permissions to* x *and* y *are obtained before the method is called); while in the* Point *class, permissions to* x *and* y *are protected by the* Point *object, which allows* internal *synchronisation of the method* move()*.*

*The calls to the methods* p1.move() *and* p2.move() *in the class* Line *(see lines 28 and 29) may potentially break the invariant* I *of a* Line *object. Therefore, the rule* ***R2"*** *requires that the invariant is unpacked before the method calls. After the execution of the methods, the invariant is reestablished and packed again.*

*Locking policy and class invariants* The discussion above shows that the invariants that can be maintained strongly depend on the locking strategy used. If a class invariant is defined on an upper layer that express properties about locations from the lower layer, the upper layer should also decide how these locations will be protected.

In Example 4.6, protection of the locations p1.x and p2.x is decided on the level of the Line class. Note that it was not necessary for a Line object to protect all four locations: p1.x, p1.y, p2.x and p2.y. Locations p1.y and p.y are not referred to by the invariant of a Line object and thus, they may be protected by the Point object or any other lock object from the lower layer. Thus, if a Point object p moves, such that only its p.y coordinate is changed, it is sufficient to obtain only

```
class Line {
  rep LinePoint p1;
  rep LinePoint p2;
//@ Invariant I:  p1.x-p2.x>5 ∨ p2.x-p1.x>5;

//@ pred state = p1 ↦^1 _ * p2 ↦^1 _;
//@ pred res_inv = state * p1.state(_, _) * p2.state(_, _) *
                   holds(p1.I, 1) * holds(p2.I, 1) * holds(I, 1);

//@ requires this.state;
//@ ensures this.state * p1.state(6,6) * p2.state(0,0) *
                  holds(p1.I, 1) * holds(p2.I, 1);
 Line(){
      p1 = new rep LinePoint(6,6);
      p2 = new rep LinePoint(0,0);
  }

//@ requires true;
//@ ensures true;
   void move() {
    acquire this;
{res_inv * locked{this}}
{state * p1.state(vx1, vy1) * p2.state(vx2, vy2) * holds(p1.I, 1) * holds(p2.I, 1)
                  * holds(I, 1) * locked{this}}
//@ unpack I {
{state * p1.state(vx1, vy1) * p2.state(vx2, vy2) * holds(p1.I, 1) * holds(p2.I, 1)
                  * break(I, 1) * locked{this}}
  p1.move();
  p2.move();
{state * p1.state(vx1-1, vy1+1) * p2.state(vx2-1, vy2+1) * holds(p1.I, 1)
                * holds(p2.I, 1) * break(I, 1) ∧ this.I * locked{this}}
//@ }
{state * p1.state(_, _) * p2.state(_,_) * holds(p1.I, 1) * holds(p2.I, 1)
                  * holds(I, 1) * locked{this}}
{res_inv * locked{this}}
    release this;
   }
}
```

Listing 4.9: Modular verification, the Line class

```
class LinePoint {
  int x; int y;
//@ Invariant I: x+y≥0;
//@ pred state(int vx, vy) = x ↦¹ vx * x ↦¹ vy;

//@ requires this.state(_, _);
//@ ensures this.state(vx, vy);
LinePoint(int vx, int vy){
   x=vx;
   y=vy;
}
//@ requires this.state(vx, vy) * holds(I,1);
//@ ensures this.state(vx-1, vy+1) * holds(I,1);
   void move() {
//@ unpack I {
        x = x - 1;
        y = y + 1;
//@  }
   }
}
```

Listing 4.10: Modular verification, the LinePoint class

the lock that protects p.y: the invariant p.I can be reestablished even without holding a positive permission to p.x, while the invariant of a Line object will not be broken because it does not refer to p.y.

**Summary** We discussed how to integrate the rules from the ownership model to provide modularity. The essential change in the verification technique presented in Section 4.2.1 is in the rule **R2**, where a new requirement is added in the prestate of every method call, i.e., rule **R2"**. This makes our verification technique sound for programs: i) that are well-typed in the Universe type system that satisfy the type rules **TR1** and **TR2**; and ii) in which all specified class invariants are admissible.

## 4.3 Formalisation

This section gives a complete formalisation of our approach. We define a simplified object-oriented concurrent language (Section 4.3.1), discuss its semantics

| | | | | | |
|---|---|---|---|---|---|
| Variables | $x, y, z, r$ | $\in$ | Var | | |
| Values | $v, w, u$ | $\in$ | Value | ::= | null $\mid n \mid b \mid o \mid$ this $\mid r \mid \pi$ |
| Integers | $n$ | $\in$ | Int | = | $\{..., -1, 0, 1, 2...\}$ |
| Booleans | $b$ | $\in$ | Bool | = | {true, false} |
| Permissions | $\pi$ | $\in$ | Permission | ::= | $1 \mid$ split$(\pi)$ |
| Object identifiers | $o$ | $\in$ | ObjId | | |
| Thread identifiers | $t$ | $\in$ | ThrId | $\subseteq$ | ObjId |
| Field identifiers | $f$ | $\in$ | FieldId | | |
| Invariant identifiers | $I$ | $\in$ | InvId | | |

Figure 4.1: Variables, values, identifiers

(Section 4.3.2) and show the proof rules of our reasoning system (Section 4.3.3). In Section 4.4 we prove soundness of our system. The language presented here is also used in Chapter 5, extended with the new history-based mechanism.

In general, the core ideas of our formalisation are taken from the verification system developed by Christian Haack et al. [AHHH14] for reasoning about concurrent Java-like programs. In contrast to their work, our language does not support re-entrant locks and class inheritance. We made these simplifications for better presentation of the thesis, because these concepts are irrelevant with the main ideas of this thesis. We also allow joining of a thread only once; this restriction was needed for soundness of our techniques. The concept of using *resources* (which we discuss later in Section 4.3.2) and the soundness of our system also follow from [AHHH14], but with certain modifications, to allow better presentation of the thesis, and of course to support our new techniques.

### 4.3.1 Language Syntax

We denote variables in our language with $x$, $y$, $z$, $r$; specifically, with $r$ we refer to *read-only* variables, see Figure 4.1. Variables may have the following values: null, integers, booleans, object identifiers, permissions. Read-only variables may also be used as values.

Figure 4.2 presents the syntax of our language. With $\overline{x}$ we define sequences of $x$, while $x?$ represents an optional $x$. Specification is added to the program using the JML annotation comments /*@ ... @*/ or //@. A program *pgm* is composed of a set of classes. Each class *cl* is defined by a set of fields ($\overline{fd}$), methods ($\overline{md}$),

| | | | |
|---|---|---|---|
| Program definition | $pgm$ | ::= | $\overline{cl}$ |
| Class definition | $cl$ | ::= | $\mathsf{class}\ C\ \langle \overline{T}\ \overline{v} \rangle\ \{\overline{fd}\ \overline{md}\ \overline{inv}\ \overline{pd}\}$ |
| Field definition | $fd$ | ::= | $T\ f$ |
| Method definition | $md$ | ::= | $spec\ \mathsf{void}\ m(\overline{T}\ \overline{x})\{c\} \mid spec\ C(\overline{T}\ \overline{x})\{sc\}$ |
| Commands | $c$ | ::= | $T\ x \mid x = v \mid x = op(\overline{v}) \mid T\ r = x$ |
| | | | $\mid \mathsf{if}\ v\ \mathsf{then}\ c\ \mathsf{else}\ c \mid \mathsf{while}\ e\ \{c\} \mid c; c \mid v.m(\overline{v})$ |
| | | | $\mid x = \mathsf{new}\ rtype\ C\ \langle \overline{v} \rangle \mid x = v.f \mid v.f = v$ |
| | | | $\mid \mathsf{fork}\ v \mid \mathsf{join}\ v \mid \mathsf{fork}\ v \parallel sc$ |
| | | | $\mid \mathsf{acquire}\ v \mid \mathsf{release}\ v \mid \mathsf{commit}\ v$ |
| | | | $\mid \mathsf{unpack}\ v.I\{sc\} \mid \mathsf{pack}\ v \mid \mathsf{assert}\ F$ |
| Expressions | $e$ | ::= | $v \mid x \mid op(\overline{e})$ |
| Operators | $op$ | $\in$ | $\{==, >, <, \wedge, \vee, \Rightarrow, ...+, -, ...\}$ |

Figure 4.2: Language syntax

predicates ($\overline{pd}$) and class invariants ($\overline{inv}$).

*Class parameters* A class may also be specified with class parameters $\langle \overline{T}\ \overline{v} \rangle$. (Note that for convenience in Figure 4.2 and Figure 4.3 we avoid using the annotations /*@ ... @*/ or //@.) Class parameters are passed at type declaration and are treated as part of the object type. A typical use of such a class parameter is the Lock class defined as:

```
class Lock/*@⟨pred inv⟩@*/{
  pred res_inv = inv;
}
```

In this way, the client class may decide which locations are protected by the lock object, and may send the resource invariant to the lock object via a class parameter.

```
pred inv = x↦¹v;
Lock/*@⟨inv⟩@*/ lock = new Lock/*@⟨inv⟩@*/();
```

**Commands** We denote commands with $c$. Most of them are standard, thus we discuss here only the most interesting ones.

To assign a variable, the right-hand side may contain only read-only variables. This requirement is added to simplify the formal treatment. It is not a

real restriction, since any assignment to $x$ can always be rewritten in this form by first storing the right-hand side to a local read-only variable $r$, and then assigning the variable $x$ to $r$. For convenience this requirement is usually not respected in the examples.

The language contains void methods only, which is not a restriction as return values can always be passed as method parameters. A special type of methods are constructors, whose implementation must be a safe command, denoted with $sc$.

Commands fork $v$ and join $v$ are used to start and end a thread $v$, respectively. Starting a thread executes the run() method of the called thread. Additionally, the language contains a parallel construct fork $v \parallel sc$, which executes the thread $v$ in parallel with a safe command $sc$. This command is normally used for implementing parallelism within safe commands, where it is required that any thread that is forked must also be joined. Therefore, for convenience, within a safe command, we forbid the standard fork $v$ and join $v$ commands, but we allow fork $v \parallel sc$, when the thread $v$ is also safe.

To commit a lock and make it available for acquiring, the command commit $v$ is used. A thread can acquire or release a commited lock, by using the commands acquire $v$ and release $v$, respectively.

For management of class invariants, we use the two specification commands: pack $v$, to pack an object $v$, and unpack $v.I\{sc\}$, to mark $sc$ as an unpacked segment of the invariant $v.I$. The command within the unpacked segment has to be a safe command.

Finally, the specification-only assert $F$ is used to check the validity of the formula $F$ in a given program state.

Below we list the subset of commands that can be used as safe commands.

$$
\begin{array}{rcl}
sc & ::= & T\ x \mid x = v \mid x = op(\overline{v}) \mid T\ r = x \mid \mathsf{if}\ v\ \mathsf{then}\ sc\ \mathsf{else}\ sc \mid \mathsf{while}\ e\ \{sc\} \\
& & \mid sc; sc \mid v.m(\overline{v}) \text{ where } m \text{ is a safe method} \mid x = \mathsf{new}\ rtype\ C\ \langle \overline{v} \rangle \\
& & \mid x = v.f \mid v.f = v \mid \mathsf{fork}\ v \parallel sc \text{ where } v \text{ is a safe thread} \\
& & \mid \mathsf{unpack}\ v.I\{sc\} \mid \mathsf{pack}\ v \mid \mathsf{assert}\ F
\end{array}
$$

**Types** We define types in our language as:

$$
\begin{array}{rcl}
T, U, V \in \mathsf{Type} & ::= & \mathsf{int} \mid \mathsf{bool} \mid \mathsf{perm} \mid (rtype,\ C\ \langle \overline{v} \rangle) \\
rtype \in RefType & ::= & \mathsf{rep} \mid \mathsf{peer} \mid \mathsf{self} \mid \mathsf{readonly}
\end{array}
$$

| | | | |
|---|---|---|---|
| Predicates | $pd$ | ::= | $\mathsf{pred}\ P(\overline{T}\ \overline{x}) = F\ (P \neq res\_inv)$ |
| | | | $\mid \mathsf{pred}\ res\_inv = F_{res}$ |
| | $inv$ | ::= | $\mathsf{Invariant}\ I : F_{inv}$ |
| Method specification | $spec$ | ::= | $\mathsf{requires}\ F\ \mathsf{ensures}\ F\ \mathsf{pure?}\ \mathsf{safe?}$ |
| Formulas | $F$ | ::= | $e \mid e.f \overset{\pi}{\mapsto} e \mid e.P(\overline{v}) \mid F \oplus F \mid (qt\ T\ x)F$ |
| | | | $\mid \mathsf{holds}(e.I, \pi) \mid \mathsf{break}(e.I, \pi)$ |
| | | | $\mid \mathsf{fresh}(e) \mid \mathsf{initialised}(e) \mid \mathsf{locked}(e) \mid \mathsf{join}(e)$ |
| | $F_{res}$ | ::= | $e \mid e.f \overset{\pi}{\mapsto} e \mid e.P(\overline{v}) \mid F_{res} \oplus F_{res}$ |
| | | | $\mid (qt\ T\ x)F_{res} \mid \mathsf{holds}(e.I, \pi)$ |
| | $F_{inv}$ | ::= | $e_{inv} \mid F_{inv} \oplus F_{inv} \mid (qt\ T\ x)(F_{inv})$ |
| Expressions | $e$ | ::= | $v \mid \_ \mid x \mid op(\overline{e})$ |
| | $e_{inv}$ | ::= | $v \mid e_{inv}.f \mid op(\overline{e_{inv}})$ |
| Quantifiers | $qt$ | $\in$ | $\{\forall, \exists\}$ |
| Operators | $\oplus$ | $\in$ | $\{*, -\!\!*, \wedge, \vee\}$ |

Figure 4.3: Specification language

The tuple $T = (rtype, C\,\langle\overline{v}\rangle)$ represents a type of an object reference $o$. The first component, i.e., $T^1$ (see Appendix A), is the reference type that defines the relation between the object $o$ and the this object. The reference type is a modifier from the set $RefType = \{\mathsf{rep}, \mathsf{peer}, \mathsf{self}, \mathsf{readonly}\}$. The second component, i.e., $T^2$, represents the class of the object $o$. Consequently, two references pointing to the same object might have different reference types if they are in different contexts.

We define a function $df : \mathsf{Type} \mapsto \mathsf{Value}$ that maps every type (except permission types for which no definition is needed) to the default value of a variable of this type. When a variable $x$ of type $T$ is declared, without being initialised, $x$ gets the value $df(T)$. We define this function as:

$$df(\mathsf{int}) \triangleq 0 \quad df(\mathsf{bool}) \triangleq \mathsf{false} \quad df((rtype\ C\,\langle\overline{v}\rangle)) \triangleq \mathsf{null}$$

**Specification formulas** As shown in Figure 4.2, classes in our language are specified with class invariants and predicates, and each method is equipped with a method specification. Figure 4.3 presents the specification language. A predicate is defined by a name $P$, parameters $\overline{x}$ and a definition represented via a specification formula $F$. A special predicate is the predicate with the name $res\_inv$, which represents the resource invariant. Methods contain the standard

contract, **requires** and **ensures** clause; additionally modifiers safe and pure are used to specify that the method is safe and/or pure, respectively.

We distinguish three types of specification formulas:

- *Standard formulas* $F$ are used to specify method contracts. These formulas are expressed in permission-based separation logic, which means that the expressions are framed. The $\mathsf{holds}(e.I, \pi)$ and $\mathsf{break}(e.I, \pi)$ predicates are special predicates used to describe the state of a class invariant. Similarly the $\mathsf{fresh}(e)$ and $\mathsf{initialised}(e)$ predicates describe the state of a given lock. The $\mathsf{locked}(e)$ predicate indicates that the lock is owned by the thread that we reason about. The $\mathsf{join}(e)$ predicate describes that the thread that we reason about has the right to join the thread $e$.

- *Resource invariant formulas* $F_{res}$ are used to express the *res_inv* predicate. They are more restrictive than the standard formulas as they exclude the special predicates $\mathsf{break}(e.I, \pi)$, $\mathsf{fresh}(e)$, $\mathsf{initialised}(e)$ or $\mathsf{locked}(e)$.

- *State formulas* $F_{inv}$ are used to specify class invariants and describe properties over shared memory locations only. These are first-order logic formulas and their syntax excludes permissions or any of the special predicates.

*Class invariant footprint* We use the term footprint of a class invariant $v.I$, written $fp(v.I)$ to refer to the set of shared locations that are referred by $v.I$. We define this formally by induction on the structure of $v.I$:

$$
\begin{array}{lcl}
fp(v) & = & \emptyset \\
fp(v.f_1...f_n) & = & \{v, v.f_1, ..., v.f_1...f_n\} \\
fp(op(\overline{e_{inv}})) & = & \bigcup_{e \in \overline{e_{inv}}} fp(e) \\
fp(F_{inv_1} \oplus F_{inv_2}) & = & fp(F_{inv_1}) \cup fp(F_{inv_2}) \\
fp((qt\ T\ x)(F_{inv})) & = & \bigcup_{v \in T \setminus \{x\}} fp(F_{inv}[v/x])
\end{array}
$$

### 4.3.2 Language Semantics

This section describes the semantics of our language. First we define the semantics of values. Thereafter, we describe the program state, which is necessary to define the semantics of commands and formulas in the language.

*Semantics of expressions* We write $[\![e]\!]^h_s$ to denote the semantics of an expression $e$, given a heap $h$ and a store $s$. When $h$ and $s$ are not relevant we normally omit them. This means that the notation $[\![e]\!]$ is used when $\forall s, h. [\![e]\!]^h_s = [\![e]\!]$.

Permission values have the following semantics:

$$[\![1]\!] = 1 \qquad [\![\mathsf{split}\,(\pi)]\!] = \frac{[\![\pi]\!]}{2},$$

while the semantics of the other values is straightforward:

$$[\![v]\!] = v \text{ for } v \in \{\mathsf{Int}, \mathsf{Bool}, \mathsf{ObjId}, \mathsf{null}\}$$

The semantics of variables $x$, shared locations and operators is defined as:

$$\begin{array}{lcl} [\![x]\!]^h_s & = & s(x) \\ [\![r]\!]^h_s & = & s(r) \\ [\![e.f]\!]^h_s & = & h([\![e]\!]^h_s)^2(f) \\ [\![op(v_1, ..., v_n)]\!] & = & [\![op]\!]([\![v_1]\!], ..., [\![v_n]\!]), \end{array}$$

where the operators $op \in \{==, \wedge, \vee, \Rightarrow, +, -, ...\}$ have the standard semantics.

*Program state* We model the program state as:

$$\sigma \in \mathsf{State} = \mathsf{Heap} \times \mathsf{ThreadPool} \times \mathsf{LockTable} \times \mathsf{InvTable}$$

The first three components are standard and used to express the semantics of the programming commands, while InvTable is a *ghost* component: the specification commands operate on this component only. Concretely:

i) $h \in \mathsf{Heap}$ represents the heap, i.e., the shared memory accessible by all active threads. The heap is a function that maps each object identifier to its *type* and its *store*, i.e., a mapping from object fields to their values. Formally:

$$\begin{array}{lcl} h \in \mathsf{Heap} & = & \mathsf{ObjId} \rightharpoonup (\mathsf{Type} \times \mathsf{Store}) \\ st \in \mathsf{Store} & = & \mathsf{FieldId} \rightharpoonup \mathsf{Value} \end{array}$$

Therefore, $h(o)^1$ represents the type of the object $o$, while $h(o)^2(f)$ represents the value of the field $f$ of object $o$. For convenience, we denote the set of shared locations as $\mathsf{Loc} = \mathsf{ObjId} \times \mathsf{FieldId}$. We denote a location as $o.f \in \mathsf{Loc}$, and refer to its value by $h(o.f)$ instead of $h(o)^2(f)$.

ii) $tp \in \mathsf{ThreadPool}$ defines all threads that operate on the heap. Each thread has a local memory modelled as a stack of frames, and a command to execute. Each frame $fr \in \mathsf{Frame}$ represents the local memory of one method

call; it maps the method local variables to their value:

$$\begin{array}{lcl} tp \in \mathsf{ThreadPool} & = & \mathsf{ThrId} \rightharpoonup \mathsf{Stack(Frame)} \times \mathsf{Cmd} \\ fr \in \mathsf{Frame} & = & \mathsf{Var} \rightharpoonup \mathsf{Value} \end{array}$$

iii) $lt \in \mathsf{LockTable}$ stores the state of all locks. The domain of the function consists of all object identifiers because every object can be used as a lock. If the lock $o$ is fresh, i.e., still not committed, then $lt(o) = \mathsf{Fresh}$; for a free lock we have $lt(o) = \mathsf{Free}$; otherwise $lt(o) = t$ where $t$ is the identifier of the thread that owns $o$. Formally:

$$lt \in \mathsf{LockTable} \quad = \quad \mathsf{ObjId} \rightharpoonup \{\mathsf{Fresh}, \mathsf{Free}\} \uplus \mathsf{ThrId}$$

iv) $it \in \mathsf{InvTable}$ describes the state of the class invariants in the program:

$$it \in \mathsf{InvTable} \quad = \quad (\mathsf{ObjId} \times \mathsf{InvId}) \rightharpoonup \{\mathsf{Packed}, \mathsf{Unpacked}\}$$

For convenience we usually write $o.I$ instead of $(o, I)$. A packed invariant $o.I$ maps to $\mathsf{Packed}$, while an unpacked invariant maps to $\mathsf{Unpacked}$.

### Operational semantics

We define the semantics of the commands in our language from Figure 4.2 in terms of transitions $\sigma \rightsquigarrow \sigma'$. A state is represented via a tuple $(h, tp, lt, it)$. For a thread pool $tp = \{t_1, ...t_n\}$, where each thread $t_i$ is $t_i = (s_i, c_i)$, we write $(t_1, s_1, c_1)$. ... $.(t_n, s_n, c_n)$. Furthermore, a stack in which the top frame is $f$ is represented by $f \cdot s$. The expression $b \triangleright c$ stands for "*if b then c else abort*". The mapping $f[x \mapsto y]$ is equivalent to $f$, except that $x$ maps to $y$.

*Local variables* We start with the semantics of the first four commands listed in Figure 4.2. They all operate on the local memory only, i.e., the top frame of the stack associated to the thread that executes the command. Therefore, the heap, the thread pool and the lock table remain unchanged.

$$\begin{array}{ll} [Dcl] & (h, tp.(t, fr \cdot s, T\ x; c), lt, it) \rightsquigarrow \\ & \qquad x \notin dom(fr) \triangleright (h, tp.(t, fr[x \mapsto df(T)] \cdot s, c), lt, it) \\ [VarSet] & (h, tp.(t, fr \cdot s, x = v; c), lt, it) \rightsquigarrow \\ & \qquad x \in dom(fr) \triangleright (h, tp.(t, fr[x \mapsto v] \cdot s, c), lt, it) \end{array}$$

$$
\begin{array}{ll}
[Op] & (h, tp.(t, fr \cdot s, x = op(\overline{v}); c), lt, it) \rightsquigarrow \\
& \quad x \in dom(fr) \triangleright (h, tp.(t, fr[x \mapsto [\![op(\overline{v})]\!]^h_{fr.s}] \cdot s, c), lt, it) \\
[FinDcl] & (h, tp.(t, fr \cdot s, T\ r = x; c), lt, it) \rightsquigarrow \\
& \quad x \in dom(fr) \triangleright (h, tp.(t, fr \cdot s, c[fr(x)/r]), lt, it)
\end{array}
$$

We give an informal explanation of the rules:

- $[Dcl]$: when a new variable $x$ is declared, the top frame is extended with $x$ mapping to the default value of the type $T$;
- $[VarSet]$: setting a value $v$ to a local variable $x$ updates the top frame, mapping $x$ to the new assigned value;
- $[Op]$: assigning an operation of values to a local variable $x$ also updates the top frame only.
- $[FinDcl]$: when a new *read-only* variable is defined, there is no need to store it on the stack; instead, every occurrence of this variable in the continuation is substituted with the assigned value.

*The if and while constructs* The next two rules describe the *if* and the *while* constructs in our language. We do not distinguish variables that are local to a construct. Thus, local variables in the method should be defined at the beginning of the method before any *if* or *while* construct.

$$
\begin{array}{ll}
[If] & (h, tp.(t, s, \mathsf{if}(v)\{c_1\}\mathsf{else}\{c_2\}; c), lt, it) \rightsquigarrow (h, tp.(t, s, c'; c), lt, it) \\
& \quad \text{where } c' = c_1 \text{ if } [\![v]\!]^h_s \text{ or } c' = c_2 \text{ otherwise} \\
[While] & (h, tp.(t, s, \mathsf{while}\ e\ \{c_1\}; c), lt, it) \rightsquigarrow (h, tp.(t, s, c'), lt, it) \\
& \quad \text{where } c' = c_1; \mathsf{while}\ e\ \{c_1\}; c \text{ if } [\![e]\!]^h_s \text{ or } c' = c \text{ otherwise}
\end{array}
$$

*Method calls* To model method calls we use an additional command return that represents the end of a method. The operational semantics is defined as:

$$
\begin{array}{ll}
[Call] & (h, tp.(t, s, o.m(\overline{v}); c), lt, it) \rightsquigarrow \\
& \quad o \in dom(h) \triangleright (h, tp.(t, \emptyset \cdot s, m\_inline(h(o)^1, m, o, \overline{v}); \mathsf{return}; c), lt, it) \\
[Return] & (h, tp.(t, fr \cdot s, \mathsf{return}; c), lt, it) \rightsquigarrow (h, tp.(t, s, c), lt, it)
\end{array}
$$

The rules show that:

- $[Call]$ : a call to a method $o.m(\overline{v})$ adds a new empty frame on the top of the local stack; the method is then *inlined*: the function $m\_inline(C, m, o, \overline{v})$ (see Appendix B), followed by return, returns the body of the method $m$ defined in the class $C$ ($C$ is the type of the object $o$) such that: the method receiver is replaced with $o$ and method parameters are replaced with $\overline{v}$.
- $[Return]$ : the command return removes the top frame from the stack.

*Shared accesses* Access to the shared heap happens when: i) a new object is created; ii) a heap location is read; or iii) a heap location is assigned. The semantics of these commands is defined as:

$$
\begin{array}{ll}
[New] & (h, tp.(t, fr \cdot s, x = \mathsf{new}\ C\,\langle v\rangle\,; c), lt, it) \rightsquigarrow \\
& \qquad x \in dom(fr) \triangleright (h', tp.(t, fr[x \mapsto o] \cdot s, c), lt[o \mapsto \mathsf{Fresh}], it') \\
& \text{where } h' = h[o \mapsto (C, initStore(C))], o \notin dom(h) \wedge \\
& \qquad it' = it[(o, I) \mapsto \mathsf{Unpacked}]_{\forall I \in inv(C)} \\
[Read] & (h, tp.(t, fr \cdot s, x = o.f; c), lt, it) \rightsquigarrow o \in dom(h) \wedge f \in dom(h(o)^2) \\
& \qquad \wedge x \in dom(fr) \triangleright (h, tp.(t, fr[x \mapsto h(o.f)] \cdot s, c), lt, it) \\
[Write] & (h, tp.(t, s, o.f = v; c), lt, it) \rightsquigarrow \\
& \qquad o \in dom(h) \wedge f \in dom(h(o)^2) \triangleright (h[o.f \mapsto v], tp.(t, s, c), lt, it)
\end{array}
$$

The rules can be understood as follows:

- $[New]$ : when a new object instance is created, a new object identifier $o$ is associated to the variable $x$ and stored in the top frame. The heap is then extended with: $o \mapsto (C, initStore(C))$, where $initStore(C)$ is a mapping that maps all fields of the class $C$ to their default values (see Appendix B). The new object can also be used as a lock, but it currently is *fresh*, and not available for acquiring; thus, the lock table is also extended with $o \mapsto \mathsf{Fresh}$. Furthermore, the invariants table is also extended such that each new class invariant $o.I$ is mapped to $\mathsf{Unpacked}$. For the formal definition of $inv(C)$ see Appendix B.
- $[Read]$ : read access to a shared location updates the top frame; the value of the heap location is assigned to the local variable;
- $[Write]$ : write access to a shared location updates the heap; the new value is assigned to the heap location.

*Thread management* The semantics of the commands management for threads is defined as:

$$
\begin{array}{ll}
[Fork] & (h, tp.(t, s, \mathsf{fork}\ t'; c), lt, it) \rightsquigarrow t' \in dom(h) \wedge t' \notin (dom(tp) \cup \{t\}) \triangleright \\
& \qquad (h, tp.(t, s, c).(t', \emptyset, t\_inline(C, t')\,; \mathsf{return}), lt, it) \\
[Join] & (h, tp.(t, s, \mathsf{join}\ t'; c).(t', s', \mathsf{return}), lt, it) \rightsquigarrow (h, tp.(t, s, c), lt, it) \\
[Parallel] & (h, tp.(t, s, \mathsf{fork}\ t' \parallel c; c'), lt, it) \rightsquigarrow (h, tp.(t, s, \mathsf{fork}\ t'; c; \mathsf{join}\ t'; c'), lt, it)
\end{array}
$$

The rules can be read as:

- $[Fork]$ : The command $\mathsf{fork}\ o$ extends the thread pool with a new thread $(o, s, c)$ where: the stack $s$ contains a single frame that is an empty mapping $(\emptyset)$; $c = t\_inline(C, o)$, i.e., the method body of the $\mathsf{run}()$ method in the

class $C$, see Appendix B.

- [*Join*] : The command join $o$ removes the thread $o$ from the thread pool.
- [*Parallel*]: The parallel construct fork $t \parallel c$ (we explained on page 88) has the same behaviour as the sequence fork $t$; $c$; join $t$.

*Lock management* The next three commands are lock-related commands, which change the state of the lock table $lt$ component.

$$
\begin{array}{ll}
[Commit] & (h, tp.(t, s, \mathsf{commit}\ o; c), lt, it) \rightsquigarrow \\
& \quad lt(o) = \mathsf{Fresh} \triangleright (h, tp.(t, s, c), lt[o \mapsto \mathsf{Free}], it) \\
[Acquire] & (h, tp.(t, s, \mathsf{acquire}\ o; c), lt, it) \rightsquigarrow \\
& \quad o \in dom(h) \triangleright \text{if } lt(o) = \mathsf{Free} \text{ then } (h, tp.(t, s, c), lt[o \mapsto t], it) \\
[Release] & (h, tp.(t, s, \mathsf{release}\ o; c), lt, it) \rightsquigarrow lt(o) = t \triangleright (h, tp.(t, s, c), lt[o \mapsto \mathsf{Free}], it)
\end{array}
$$

Informally:

- [*Commit*] : When an object $o$ is committed, the lock becomes available to acquire; its state is changed to Free.
- [*Acquire*] : After a thread $t$ acquires a lock $o$, the lock $o$ is mapped to $t$.
- [*Release*] : Similarly, releasing a lock $o$ changes the state of $o$ to Free.

*Class invariant management* The commands pack $o$ and unpack $o.I\{c\}$ are used for management of class invariants. They both operate on the InvTable component only, and moreover, there is no restriction for these commands to be executed (they can never be aborted). This is important because whether the program will reach a given state should not depend on its specification.

To model the command unpack $o.I\{c\}$, we add the auxiliary command pack $o.I$ which indicates the end of the unpacked segment. The semantics of the commands is given below:

$$
\begin{array}{ll}
[PackObj] & (h, tp.(t, s, \mathsf{pack}\ o; c), lt, it) \rightsquigarrow \\
& \quad (h, tp.(t, s, c), lt, it[o.I \mapsto \mathsf{Packed}]_{\forall I \in inv(h(o)^1)}) \\
[Unpack] & (h, tp.(t, s, \mathsf{unpack}\ o.I\{c\}; c'), lt, it) \rightsquigarrow \\
& \quad (h, tp.(t, s, c; \mathsf{pack}\ o.I; c'), lt, it[o.I \mapsto \mathsf{Unpacked}]) \\
[PackInv] & (h, tp.(t, s, \mathsf{pack}\ o.I; c), lt, it) \rightsquigarrow \\
& \quad (h, tp.(t, s, c; \mathsf{pack}\ o.I), lt, it[o.I \mapsto \mathsf{Packed}])
\end{array}
$$

- [*PackObj*] : Packing an object $o$ changes the state of all invariants of this object to Packed.
- [*Unpack*] : When a class invariant is unpacked, its state its changed to

Unpacked; thereafter the code in the unpacked segment needs to be executed, followed by a command pack $o.I$.

- [*PackInv*] : At the end of the unpacked segment, the state of the class invariant is changed to Packed.

*Initial program state* We model the initial global program state of a program *pgm* as:

$$\sigma_0 = (\emptyset, (o_{main}, \emptyset, c_{main}), \emptyset, \emptyset)$$

where $c_{main}$ is the main method of the program (the method where the initial execution starts). The thread pool contains a single thread $o_{main}$ with an empty stack, which needs to execute the code $c_{main}$. The thread $o_{main}$ contains no fields, no invariants; thus the heap, the lock and invariant tables are empty.

### Resources and semantics of formulas

We expressed the semantics of the language operators in terms of transitions over the global program state. The semantics of formulas, however, can not be defined directly over the global state. Because our reasoning is thread-local, formulas also carry information that is specific to the thread that we reason about, e.g., permissions and fractions of the holds or break predicates. This information is not part of the global state. Therefore, for the formula $o.f \overset{\pi}{\mapsto} v$ for example, we can express over the global state that "the vaue of $o.f$ stored on the heap is $v$", but we can not express that "the current thread holds permission $\pi$ for the location $o.f$".

Extending the global program state with this thread-local information is inconvenient and complicated. We want to keep the global state clean, such that it describes the behaviour of the program only. How the thread-local data is distributed and transferred among threads in the program is determined via the program specification, which should not affect the state of the program.

**Resources** Therefore, we define the notion of a *resource*, denoted $\mathcal{R}$, and express the semantics of formulas over a given resource. One way to understand the resource-based concept is that the global state is divided into multiple resources, such that each resource owns part of the state only. Resources are distributed among threads in the program. Thus, a resource represents the thread-local state; a thread that owns a resource $\mathcal{R}$ views the global state via $\mathcal{R}$. If the program is sequential, the resource is practically equivalent to the global state.

The concept of resources has been used in Christian Haack et al.'s work [AHHH14]. In a similar fashion, Thomas Dinsdale-Young et al. [DYBG+13] use *views* in their framework for compositional reasoning about concurrent programs, where a view is analogous to a resource; it consists of abstract knowledge about the global state, that describes how the thread views the state.

Resources owned by different threads must be *compatible*, written $\mathcal{R}_1 \# \mathcal{R}_2$. Compatibility means that parts of the global state that is common for both $\mathcal{R}_1$ and $\mathcal{R}_2$ must be consistent. For example, $\mathcal{R}_1 \# \mathcal{R}_2$ ensures that $\mathcal{R}_1$ and $\mathcal{R}_2$ map each location to the same value. Moreover, the compatibility relation describes in a way how the state is divided among threads. For example, $\mathcal{R}_1 \# \mathcal{R}_2$ can describe that the sum of permissions to a concrete shared location in both resources does not exceed 1.

Intuitively, distribution of resources happens normally when threads are forked; when threads join, resources owned by each thread are also joined (merged). To this end, a *join* operation is defined over the set of resources, denoted $\mathcal{R}_1 * \mathcal{R}_2$. The result is a resource that combines all information from both $\mathcal{R}_1$ and $\mathcal{R}_2$. Importantly, joining $\mathcal{R}_1 * \mathcal{R}_2$ is only defined when the resources are compatible, $\mathcal{R}_1 \# \mathcal{R}_2$.

Now we proceed to the formal definition of a resource. We model a resource as a tuple

$$\mathcal{R} = (\mathcal{H}, \mathcal{L}, \mathcal{T}, \mathcal{J})$$

where each component is an abstraction of part of the global state: $\mathcal{H}$ represents the heap, $\mathcal{L}$ abstracts the lock table, $\mathcal{T}$ abstracts the class invariant table, while $\mathcal{J}$ holds information about the forked threads. For a resource $\mathcal{R} = (\mathcal{H}, \mathcal{L}, \mathcal{T}, \mathcal{J})$ that is an abstraction of the global state $\sigma = (h, tp, lt, it)$ we define the relation $\mathit{abstracts}(\sigma, \mathcal{R})$. Its definition is intuitive: $\sigma$ and $\mathcal{R}$ contain consistent information (see Appendix B for the formal definition).

We define the compatibility relation and the joining operator for each component separately: Compatibility and joining of resources are defined componentwise:

$$\begin{aligned} \mathcal{R} \# \mathcal{R}' &\Leftrightarrow \mathcal{H} \# \mathcal{H}' \wedge \mathcal{L} \# \mathcal{L}' \wedge \mathcal{T} \# \mathcal{T}' \wedge \mathcal{J} \# \mathcal{J}' \\ \mathcal{R} * \mathcal{R}' &= \begin{cases} (\mathcal{H} * \mathcal{H}', \mathcal{L} * \mathcal{L}', \mathcal{T} * \mathcal{T}', \mathcal{J} * \mathcal{J}') & \text{if } \mathcal{R} \# \mathcal{R}' \\ \bot & \text{otherwise.} \end{cases} \end{aligned}$$

When modelling the resource $\mathcal{R}$ (and each of its components) we have to think about the following: $\mathcal{R}$ should contain information such that each formula from our language can be expressed over it; and the compatibility relation and

the joining operator should correctly define how the information is distributed among threads.

*Abstract heap* The first component $\mathcal{H}$ is an abstraction of the global heap $h$, where $h$ contains the values of shared locations only. Because our formulas contain also permissions to locations, $\mathcal{H}$ extends $h$ such that it maps every location to its value and to the permission for this location owned by the resource. Thus, the abstract heap $\mathcal{H}$ is a masked heap:

$$\mathcal{H} : \mathsf{ObjId} \rightharpoonup \mathsf{Type} \times (\mathsf{FieldId} \rightharpoonup (\mathsf{Value} \times [0,1]))$$

For simplicity, for a location $o.f$, we write $\mathcal{H}(o.f)^1$ to denote its value, and $\mathcal{H}(o.f)^2$ to denote the permission of $o.f$ stored in $\mathcal{H}$.

Two abstract heaps are compatible, if they agree on the types and values for every location, and the sum of the permissions for every location does not exceed 1. When resources join, the result is a resource that contains the sum of permissions from both resources:

$$\begin{array}{lcl} \mathcal{H}_1 \# \mathcal{H}_2 & \Leftrightarrow & \forall o. \mathcal{H}_1(o)^1 = \mathcal{H}_2(o)^1 \wedge \\ & & \forall (o.f). \mathcal{H}_1(o.f)^1 = \mathcal{H}_2(o.f)^1 \wedge \mathcal{H}_1(o.f)^2 + \mathcal{H}_2(o.f)^2 \leq 1 \\ \mathcal{H}_1 * \mathcal{H}_2 & = & \begin{cases} \lambda o. (\mathcal{H}_1(o)^1, \lambda f. (\mathcal{H}_1(o.f)^1, \mathcal{H}_1(o.f)^2 + \mathcal{H}_2(o.f)^2) & \text{if } \mathcal{H}_1 \# \mathcal{H}_2 \\ \bot & \text{otherwise.} \end{cases} \end{array}$$

The domain of $\mathcal{H}$ contains all locations from the global heap. However, the value of a location $x$ in $\mathcal{H}$ is relevant only when this location is framed, i.e., i) the resource contains a positive permission to $x$; ii) the resource contains a positive fraction of the $\mathsf{holds}$ predicate of an invariant that refers to $x$; or iii) the resource contains a fraction 1 of the $\mathsf{break}$ predicate of an invariant that refers to $x$. Otherwise, the value can just be ignored. Technically, it is not necessary to express this requirement in the compatibility relation, but we will see later that this requirement is important when expressing the semantics of formulas.

*Abstract lock table* The component $\mathcal{L}$ is an abstract lock table that contains information about the state of the locks. We use $\mathcal{L}$ to define the meaning of the predicates: $\mathsf{fresh}(o)$, $\mathsf{initialised}(o)$, or $\mathsf{locked}(o)$. Therefore, we model the abstract lock table as a tuple:

$$\mathcal{L} : \mathsf{ObjId} \rightharpoonup \mathsf{Bool} \times \mathsf{Bool} \times \mathsf{Bool}$$

where: i) $\mathcal{L}(o)^1$ represents whether the lock is fresh; ii) $\mathcal{L}(o)^2$ denotes whether the lock is initialised; iii) and $\mathcal{L}(o)^3$ represents whether the lock is owned by the resource.

To define the compatibility relation and the joining operator, we look at how this information is shared among threads. First the information that a lock is fresh must be local to a thread, and thus, it can not be owned by two resources. This is important, as otherwise it might happen that a thread changes the status of a lock from fresh to initialised, while other threads still keep the old information that the lock is fresh. This means that: if $\mathsf{fresh}(o)$ holds over a resource $\mathcal{R}$, it can not be split over two resources, such that $\mathsf{fresh}(o)$ holds over the two of them: We say that the predicate $\mathsf{fresh}(o)$ is not *copyable*, i.e., it can not be shared among threads:

$$\mathsf{fresh}(o) \mathrel{*\!\!-\!\!*} \mathsf{fresh}(o) * \mathsf{fresh}(o) \;//\; \text{Wrong}$$

In contrast, the information about a lock being initialised can be shared among threads, because once a lock is initialised, it stays initialised forever. The $\mathsf{initialised}(o)$ predicate is copyable and thus:

$$\mathsf{initialised}(o) \mathrel{*\!\!-\!\!*} \mathsf{initialised}(o) * \mathsf{initialised}(o) \;//\; \text{Correct}$$

Obviously, the $\mathsf{locked}(o)$ predicate is also not copyable; owned locks must be local to a given resource. Finally, we define the compatibility relation and the joining operator as follows:

$$\begin{array}{lcll} \mathcal{L}_1 \# \mathcal{L}_2 & \Leftrightarrow & \forall o.(\mathcal{L}_1(o)^1 \wedge \mathcal{L}_2(o)^1 = \mathsf{false}) \wedge (\mathcal{L}_1(o)^2 = \mathcal{L}_2(o)^2) \wedge & \\ & & (\mathcal{L}_1(o)^3 \wedge \mathcal{L}_2(o)^3 = \mathsf{false}) & \\ \mathcal{L}_1 * \mathcal{L}_2 & = & \begin{cases} \lambda o.(\mathcal{L}_1(o)^1 \vee \mathcal{L}_2(o)^1,\ \mathcal{L}_1(o)^2,\ \mathcal{L}_1(o)^3 \vee \mathcal{L}_2(o)^3) & \text{if } \mathcal{L}_1 \# \mathcal{L}_2 \\ \bot & \text{otherwise .} \end{cases} & \end{array}$$

*Abstract invariant table* The abstract invariant table $\mathcal{T}$ contains information about the class invariants in the program and defines the semantics of formulas $\mathsf{holds}(o, \pi)$ and $\mathsf{break}(o, \pi)$. We model $\mathcal{T}$ as:

$$\mathcal{T} : (\mathsf{ObjId} \times \mathsf{InvId}) \rightharpoonup (\{\mathsf{Packed}\} \times [0,1]) \uplus (\{\mathsf{Unpacked}\} \times [0,1])$$

If a class invariant $o.I$ is packed, $\mathcal{T}(o.I) = (\mathsf{Packed}, \pi)$, where $\pi$ is the fraction of the predicate $\mathsf{holds}$ stored in $\mathcal{T}$; for an unpacked invariant, $\mathcal{T}(o.I) = (\mathsf{Unpacked}, \pi)$, where $\pi$ is the fraction of the predicate $\mathsf{break}$.

The total sum of fractions for a predicate holds (or break) owned by all resources must not exceed 1. When resources are joined, the result sums all fractions. Formally we define compatibility and joining as:

$$\begin{array}{lcl} \mathcal{T}_1\#\mathcal{T}_2 & \Leftrightarrow & \forall(o.I).\mathcal{T}_1(o.I)^1 = \mathcal{T}_2(o.I)^1 \wedge \mathcal{T}_1(o.I)^2 + \mathcal{T}_2(o.I)^2 \leq 1 \\ \mathcal{T}_1{*}\mathcal{T}_2 & = & \begin{cases} \lambda o.(\mathcal{T}_1(o.I)^1, \mathcal{T}_1(o.I)^2 + \mathcal{T}_2(o.I)^2) & \text{if } \mathcal{T}_1\#\mathcal{T}_2 \\ \bot & \text{otherwise.} \end{cases} \end{array}$$

*Join table* The last component from a resource is the *join table*, $\mathcal{J}$, which keeps information about which threads are forked and expected to join. $\mathcal{J}$ is used to define the meaning of the join$(o)$ predicate. We model $\mathcal{J}$ as a set of thread identifiers that are expected to join.

$$\mathcal{J} \subseteq \mathsf{Thrld}$$

The join$(o)$ predicate is required when the current thread wants to join the thread $o$. We allow a thread to be joined only once in the program. This is why we force the join$(o)$ predicate to be local to a single thread only, i.e., it is neither copyable, nor can it be split among threads. The formal definition for compatibility and joining is defined as:

$$\begin{array}{lcl} \mathcal{J}_1\#\mathcal{J}_2 & \Leftrightarrow & \mathcal{J}_1 \cap \mathcal{J}_2 = \emptyset \\ \mathcal{J}_1\#\mathcal{J}_2 & = & \begin{cases} \mathcal{J}_1 \cup \mathcal{J}_2 & \text{if } \mathcal{J}_1\#\mathcal{J}_2 \\ \bot & \text{otherwise.} \end{cases} \end{array}$$

**Semantics of formulas** Finally, in Figure 4.4 we present the semantics of the formulas in our language. We use the relation $\mathcal{R}; s \models F$, which expresses validity of the formula $F$ with respect to a stack $s$ and a resource $\mathcal{R}$. The resource is defined as $\mathcal{R} = (\mathcal{H}, \mathcal{L}, \mathcal{T}, \mathcal{J})$.

The semantics of class invariant expressions $[\![e_{inv}]\!]^{\mathcal{H}}_s$ is computed over the abstract heap $\mathcal{H}$, instead of the heap $h$. The definition of $[\![e_{inv}]\!]^{\mathcal{H}}_s$ follows trivially from the definition of $[\![e_{inv}]\!]^{h}_s$ (see page 91), because $\mathcal{H}$ is an extension of $h$. It is important that such an expression may hold over $\mathcal{R}$ only when the expression is framed. We define the formula *framed*$(F_{inv}, \mathcal{R})$ (where $\mathcal{R} = (\mathcal{H}, \mathcal{L}, \mathcal{T}, \mathcal{J})$) as:

$$\begin{array}{lcl} \mathit{framed}(F_{inv}, \mathcal{R}) & \Leftrightarrow & \forall v.f \in \mathit{fp}(F_{inv}).\mathcal{H}(v.f)^2 > 0 \vee \\ & & (\exists o.I \in \mathit{dom}(\mathcal{T}).v.f \in \mathit{fp}(o.I) \\ & & \wedge\ \mathcal{T}(o.I) = (\mathsf{Packed}, \pi) \wedge \pi > 0 \vee \mathcal{T}(o.I) = (\mathsf{Unpacked}, 1)) \end{array}$$

$$\mathcal{R} = (\mathcal{H}, \mathcal{L}, \mathcal{T}, \mathcal{J})$$

| | | | |
|---|---|---|---|
| [*Exp*] | $\mathcal{R}; s \models e_{inv}$ | $\Leftrightarrow$ | $[\![e_{inv}]\!]^{\mathcal{H}}_s = \mathsf{true} \wedge \mathit{framed}(e_{inv}, \mathcal{R})$ |
| [*Perm*] | $\mathcal{R}; s \models e.f \overset{\pi}{\mapsto} e'$ | $\Leftrightarrow$ | $[\![e]\!]^{\mathcal{H}}_s = o \wedge [\![e']\!]^{\mathcal{H}}_s = v \wedge$ $\mathcal{H}(o.f) = (v, \pi') \wedge \pi' \geq \pi$ |
| [*SepConj*] | $\mathcal{R}; s \models F * F'$ | $\Leftrightarrow$ | $\exists \mathcal{R}_1, \mathcal{R}_2. \mathcal{R} = \mathcal{R}_1 * \mathcal{R}_2 \wedge$ $\mathcal{R}_1; s \models F \wedge \mathcal{R}_2; s \models F'$ |
| [*SepImpl*] | $\mathcal{R}; s \models F -\!\!* F'$ | $\Leftrightarrow$ | $\forall \mathcal{R}'. (\mathcal{R} \# \mathcal{R}' \wedge \mathcal{R}'; s \models F) \Rightarrow \mathcal{R} * \mathcal{R}'; s \models F'$ |
| [*Conj*] | $\mathcal{R}; s \models F \wedge F'$ | $\Leftrightarrow$ | $\mathcal{R}; s \models F \wedge \mathcal{R}; s \models F'$ |
| [*Disj*] | $\mathcal{R}; s \models F \vee F'$ | $\Leftrightarrow$ | $\mathcal{R}; s \models F \vee \mathcal{R}; s \models F'$ |
| [*Forall*] | $\mathcal{R}; s \models \forall Tx.F$ | $\Leftrightarrow$ | $\forall v : T. \mathcal{R}; s \models F[v/x]$ |
| [*Exists*] | $\mathcal{R}; s \models \exists Tx.F$ | $\Leftrightarrow$ | $\exists v : T. \mathcal{R}; s \models F[v/x]$ |
| [*Pred*] | $\mathcal{R}; s \models e.P(\overline{v})$ | $\Leftrightarrow$ | $\mathcal{R}; \emptyset \models p\_inline(C, P, o, \overline{v}) \wedge$ $o = [\![e]\!]^{\mathcal{H}}_s \wedge C = \mathcal{H}(o)^1$ |
| [*Fresh*] | $\mathcal{R}; s \models \mathsf{fresh}(e)$ | $\Leftrightarrow$ | $[\![e]\!]^{\mathcal{H}}_s = o \wedge \mathcal{L}(o)^1 = \mathsf{true}$ |
| [*Init*] | $\mathcal{R}; s \models \mathsf{initialised}(e)$ | $\Leftrightarrow$ | $[\![e]\!]^{\mathcal{H}}_s = o \wedge \mathcal{L}(o)^2 = \mathsf{true}$ |
| [*Locked*] | $\mathcal{R}; s \models \mathsf{locked}(e)$ | $\Leftrightarrow$ | $[\![e]\!]^{\mathcal{H}}_s = o \wedge \mathcal{L}(o)^3 = \mathsf{true}$ |
| [*Holds*] | $\mathcal{R}; s \models \mathsf{holds}(e.I, \pi)$ | $\Leftrightarrow$ | $[\![e]\!]^{\mathcal{H}}_s = o \wedge$ $\mathcal{T}(o.I) = (\mathsf{Packed}, \pi') \wedge \pi' \geq \pi$ |
| [*Breaks*] | $\mathcal{R}; s \models \mathsf{break}(e.I, \pi)$ | $\Leftrightarrow$ | $[\![e]\!]^{\mathcal{H}}_s = o \wedge$ $\mathcal{T}(o.I) = (\mathsf{Unpacked}, \pi') \wedge \pi' \geq \pi$ |
| [*Join*] | $\mathcal{R}; s \models \mathsf{join}(e)$ | $\Leftrightarrow$ | $[\![e]\!]^{\mathcal{H}}_s = o \wedge o \in \mathcal{J}$ |

Figure 4.4: Semantics of formulas

The semantics of the other formulas should be clear from the definition of the resource. Below we give an explanation of the most interesting ones:

- [*Perm*]: The formula $e.f \overset{\pi}{\mapsto} e'$ holds when the abstract heap $\mathcal{H}$ stores the value $e'$ for the location $e.f$ and stores sufficient permission for this location, i.e., at least the amount that is declared in the formula.
- [*SepConj*]: The formula $F * F'$ holds over a resource $\mathcal{R}$ if the resource can be divided into two compatible resources $\mathcal{R}_1$ and $\mathcal{R}_2$, such that $F$ can be proved to hold over $\mathcal{R}_1$ and $F'$ over $\mathcal{R}_2$.
- [*SepImpl*]: Separation implication $F -\!\!* F'$ holds over a resource $\mathcal{R}$ when: if $\mathcal{R}$ is extended with a compatible resource $\mathcal{R}'$ on which $F$ can be proved, then $F'$ can be proved on the extension $\mathcal{R} * \mathcal{R}'$.

- [*Pred*]: the expression $e.P(\overline{v})$ holds when the body of the predicate holds over $\mathcal{R}$. The auxiliary function $p\_inline(C, P, o, \overline{v})$ (see Appendix B) returns the body of the predicate $P$ defined in the class $C$, in which $\overline{v}$ replaces the predicate parameters, and $o$ replaces the receiver this.
- [*Fresh*], [*Init*] and [*Locked*]: the predicates $\mathsf{fresh}(e)$, $\mathsf{initialised}(e)$ and $\mathsf{locked}(e)$ are expressed over the $\mathcal{L}$ component, as described above.
- [*Holds*] and [*Breaks*]: the $\mathsf{holds}(e, \pi)$ and the $\mathsf{break}(e, \pi)$ predicates hold when the resource (the $\mathcal{T}$ component) stores sufficient fraction (at least $\pi$) for these predicates.
- [*Join*]: the $\mathsf{join}(e)$ predicate holds when the resource contains the object $e$ in the set of objects expected to be joined. The predicate is required by a thread when it join another thread.

### 4.3.3 Proof System

This section presents the proof rules of our reasoning system. Type restrictions are not included in the rules; we assume well-typed programs. As described in Section 2.1, a Hoare-style logic is an extension of the predicate logic with axioms and inference rules for deriving Hoare triples.

Our proof system consists of: i) *proof theory*, i.e., *logical consequence rules* represented in the form $F \vdash F'$ and *axioms* in the form $\vdash F$; and ii) *Hoare triples*, i.e., *axioms* and *inference rules* that describe the behaviour of the language commands.

*Proof theory* We start with a set of important axioms and logical consequence rules, shown in Figure 4.5. The complete list is more extensive; however we omit the standard rules (most of them extended from predicate logic) and focus on those that are more interesting for our language.

We use in the rules the expression $framed(F_{inv}, F)$ to state that the formula $F$ *frames* the state formula $F_{inv}$, i.e., $F$ expresses permissions for all locations that occur in $F_{inv}$. Formally:

$$framed(F_{inv}, F) \quad \Leftrightarrow \quad \forall \mathcal{R}, s.\mathcal{R}, s \models F \Rightarrow framed(F_{inv}, \mathcal{R})$$

- [*SplitMerge Perm*], [*SplitMerge Holds*] and [*SplitMerge Unpack*]: These rules illustrate splitting and merging of the predicates in our language.
- [*Invariant*]: The rule states that the correctness of a class invariant is a logical consequence of the holds predicate. This is an important rule which allows one to use a class invariant: any positive fraction of the holds

| | |
|---|---|
| [*SplitMerge Perm*] | $\vdash o.f \overset{\pi_1+\pi_2}{\mapsto} v_1 * v_1 == v_2 \mathrel{*\!\!-\!\!*} o.f \overset{\pi_1}{\mapsto} v_1 * o.f \overset{\pi_2}{\mapsto} v_2$ |
| [*SplitMerge Holds*] | $\vdash \mathsf{holds}(o.I, \pi_1 + \pi_2) \mathrel{*\!\!-\!\!*} \mathsf{holds}(o.I, \pi_1) * \mathsf{holds}(o.I, \pi_2)$ |
| [*SplitMerge Unpack*] | $\vdash \mathsf{break}(o.I, \pi_1 + \pi_2) \mathrel{*\!\!-\!\!*} \mathsf{break}(o.I, \pi_1) * \mathsf{break}(o.I, \pi_2)$ |
| [*Invariant*] | $\mathsf{holds}(o.I, \pi) \vdash \mathsf{holds}(o.I, \pi) \ \wedge \ o.I$ |
| [*PermToState*] | $o.f \overset{\pi}{\mapsto} w \vdash o.f \overset{\pi}{\mapsto} w \ \wedge \ o.f == w$ |
| [*Invariant Out*] | $(F_1 \wedge F_{inv}) * F_2 \vdash (F_1 * F_2) \wedge F_{inv}$ |
| [*Invariant In*] | $(F_1 * F_2) \wedge F_{inv} \vdash (F_1 \wedge F_{inv}) * F_2$ if $\mathit{framed}(F_{inv}, F_1)$ |
| [*Invariant Weakening*] | $\dfrac{F^1_{inv} \vdash F^2_{inv}}{F \wedge F^1_{inv} \vdash F \wedge F^2_{inv}}$ |

Figure 4.5: Axioms and logical consequences

predicate guarantees that the class invariant holds.

- [*PermToState*]: The rule is used to construct a state formula using the information from the appropriate resource predicate.
- [*Invariant Out*], [*Invariant In*] and [*Invariant Weakening*]: These rules are used to make transformations to expressions with state formulas. Note that in the rule [*Invariant In*] it is required that $F_{inv}$ is framed by $F_1$, while such a condition is not necessary in [*Invariant Out*]. This is because the left side of the rule [*Invariant Out*] ensures that $F_{inv}$ is framed by $F_1$, which implies that $F_{inv}$ is also framed by $F_1 * F_2$.

*Hoare triples* Figure 4.6 gives the complete list of the Hoare triples, each of them describing the behaviour of a specific command in our language. We give an informal explanation of some of them (for the formal definition of the auxiliary functions used in the rules see Appendix B):

- [*Call*]: The base of this rule is standard: if $F$ holds in the prestate of a method call, and $\{F\}\ c\ \{F'\}$ can be proved where $c$ is the body of the called method, then $F'$ holds in the poststate of the method call. In

$$[Dcl] \quad \frac{}{\vdash \{\mathsf{true}\}\ T\ x\ \{x == df(T)\}}$$

$$[VarSet] \quad \frac{}{\vdash \{x == \_\}\ x = v\ \{x == v\}}$$

$$[Op] \quad \frac{}{\vdash \{x == \_\}\ x = op(\overline{v})\ \{x == op(\overline{v})\}}$$

$$[FinDcl] \quad \frac{}{\vdash \{\mathsf{true}\}\ T\ r = x\ \{r == x\}}$$

$$[If] \quad \frac{\vdash \{F * v\}\ c\ \{F'\} \quad \{F * \neg v\}\ c'\ \{F'\}}{\vdash \{F\}\ \mathsf{if}\ v\ \mathsf{then}\ c\ \mathsf{else}\ c'\ \{F'\}}$$

$$[While] \quad \frac{\vdash \{F * e\}\ c\ \{F\}}{\vdash \{F\}\ \mathsf{while}\ e\ \{c\}\ \{\neg e * F\}}$$

$$[Sequence] \quad \frac{\vdash \{F_1\}\ c_1\ \{F_2\} \quad \vdash \{F_2\}\ c_2\ \{F_3\}}{\vdash \{F_1\}\ c_1; c_2\ \{F_3\}}$$

$$[Call] \quad \frac{\begin{array}{c}\mathsf{this} : T \quad S = \{I | I \in T^2 \wedge writeLocs(F) \cap fp(\mathsf{this}.I) \neq \emptyset\} \\ F_2 = \circledast_{I \in S}(\mathsf{break}(\mathsf{this}.I, \pi) \wedge F_{inv_I}) \\ F_2' = \circledast_{I \in S}(\mathsf{break}(\mathsf{this}.I, \pi) \wedge F'_{inv_I}) \\ \vdash \{F_1 * F_2\}\ m\_inline(m, T^2, o, \overline{v})\ \{F_1' * F_2'\}\end{array}}{\vdash \{F_1 * F_2\}\ o.m(\overline{v})\ \{F_1' * F_2'\}}$$

$$[New] \quad \frac{F = \circledast_{(T\,f) \in fld(C)} x.f \overset{1}{\mapsto} df(T) \quad S = inv(C)}{\begin{array}{l}\{\mathsf{true}\} \\ \vdash \quad x = \mathsf{new}\ rtype\ C\ \langle \overline{v} \rangle \\ \{F * \mathsf{fresh}(x) * \circledast_{I \in S} \mathsf{break}(x.I, 1)\}\end{array}}$$

$$[Read] \quad \frac{}{\vdash \{o.f \overset{\pi}{\mapsto} w\}\ x = o.f\ \{o.f \overset{\pi}{\mapsto} w * x == w\}}$$

$$[Write] \quad \frac{o : T \quad S = \{I | I \in inv(T^2), o.f \in fp(o.I)\}}{\begin{array}{l}\{o.f \overset{1}{\mapsto} \_ * \circledast_{I \in S} (\mathsf{break}(o.I, \pi) \wedge F_{inv_I}[w/o.f])\} \\ \vdash \quad o.f = w \\ \{o.f \overset{1}{\mapsto} w * \circledast_{I \in S} (\mathsf{break}(o.I, \pi) \wedge F_{inv_I})\}\end{array}}$$

$$[Fork] \quad \frac{t : T \quad F = precond(T^2, run)}{\vdash \{F[t/\mathsf{this}]\}\ \mathsf{fork}\ t\ \{\mathsf{join}(t)\}}$$

$$[Join] \quad \frac{t : T \quad F' = postcond(T^2, run)}{\vdash \{\mathsf{join}(t)\}\ \mathsf{join}\ t\ \{F'[t/\mathsf{this}]\}}$$

$$[Parallel] \quad \frac{\vdash \{F\}\ \mathsf{fork}\ t; c; \mathsf{join}\ t\ \{F'\}}{\vdash \{F\}\ \mathsf{fork}\ t \parallel c\ \{F'\}}$$

$$[Commit] \quad \frac{}{\vdash \{\mathsf{fresh}(o)\ *\ o.res\_inv\}\ \mathsf{commit}\ o\ \{\mathsf{initialised}(o)\}}$$

$$[Acquire] \quad \frac{}{\vdash \{\mathsf{initialised}(o)\}\ \mathsf{acquire}\ o\ \{o.res\_inv\ *\ \mathsf{locked}(o)\}}$$

$$[Release] \quad \frac{}{\vdash \{o.res\_inv\ *\ \mathsf{locked}(o)\}\ \mathsf{release}\ o\ \{\mathsf{true}\}}$$

$$[PackObj] \quad \frac{o : T \quad S = \{I | I \in inv(T^2)\}}{\vdash \{\circledast_{I \in S}(\mathsf{break}(o.I, 1)\ \wedge\ o.I)\}\ \mathsf{pack}\ o\ \{\circledast_{I \in S}\mathsf{holds}(o.I, 1)\}}$$

$$[Unpack] \quad \frac{\vdash \{\mathsf{break}(o.I, 1)\ \wedge\ o.I\}\ c\ \{\mathsf{break}(o.I, 1)\ \wedge\ o.I\}}{\vdash \{\mathsf{holds}(o.I, 1)\}\ \mathsf{unpack}\ o.I\{c\}\ \{\mathsf{holds}(o.I, 1)\}}$$

$$[Consequence] \quad \frac{F_1 \vdash F_1' \quad \vdash \{F_1'\}\ c\ \{F_2'\} \quad F_2' \vdash F_2}{\vdash \{F_1\}\ c\ \{F_2\}}$$

$$[Frame] \quad \frac{\vdash \{F_1\}\ c\ \{F_2\} \quad freevars(F_3) \cap writes(c) = \emptyset}{\vdash \{F_1\ *\ F_3\}\ c\ \{F_2\ *\ F_3\}}$$

Figure 4.6: Hoare triples

addition to this, the rule also includes the condition **R2"**: a predicate $\mathsf{break}(\mathsf{this}.I, \pi) \wedge F_{inv}$ is required for every invariant $\mathsf{this}.I$ that refers to a field that is possibly assigned in the method ($writeLocs(F)$).

- [*New*]: When a new object instance is created, the creator thread obtains: i) a write permission to each location $x.f$ of the new object, initialised to a default value; ii) the $\mathsf{fresh}(x)$ predicate, which can later be used to commit the object (convert it to a lock); and iii) a complete fraction of the $\mathsf{break}(x.I, 1)$ predicate for each class invariant of the new object.
- [*Read*]: Reading a shared location is standard: a read permission for the location is required.
- [*Write*]: Writing a shared location requires a write permission for this location. Additionally (as described in **R2'** above), for each invariant that refers to the location, the predicate $\mathsf{break}(\mathsf{this}.I, \pi) \wedge F_{inv}$ is required where $F_{inv}$ may be any state formula framed by the break predicate.
- [*Fork*]: When a thread $t$ is started, its precondition must be satisfied; the thread that forks $t$ obtains the $\mathsf{join}(t)$ predicate.
- [*Join*]: To join a thread $t$, the $\mathsf{join}(t)$ predicate is required, and the postcondition of $t$ holds in the poststate of the join call.
- [*Commit*]: Commiting an object $o$ exchanges the predicate $\mathsf{fresh}(o)$ for $\mathsf{initialised}(o)$, and additionally the resource invariant is transferred from the thread to the lock.
- [*Acquire*]: To acquire a lock, the thread must show the $\mathsf{initialised}(o)$ predicate; it then obtains the resource invariant and the predicate $\mathsf{locked}(o)$. Note the $\mathsf{initialised}(o)$ predicate is not explicitly mentioned in the postcondition because it is a copyable predicate.
- [*Release*]: Releasing a lock $o$ consumes both the resource invariant of the lock, and the $\mathsf{locked}(o)$ predicate.
- [*PackObj*]: packing an object $o$ requires the $\mathsf{break}(o.I, 1)$ predicate for class invariant $o.I$ and it is required that each invariant holds; each predicate $\mathsf{break}(o.I, 1)$ is then exchanged for the $\mathsf{holds}(o.I, 1)$ predicate.
- [*Unpack*]: This rule describes that the complete $\mathsf{holds}(o.I, 1)$ predicate is required in the prestate of the unpacked segment of $o.I$. Within the segment, the $\mathsf{holds}(o.I, 1)$ predicate is exchanged for $\mathsf{break}(o.I, 1)$, which has to be returned back at the end of the segment together with the guarantee that the invariant $o.I$ holds.
- [*Consequence*]: This rule allows strengthening of the precondition and weakening of the postcondition (as discussed in Section 2.1).
- [*Frame*]: The *Frame* rule is the basic separation-logic rule (discussed in Section 3.2). The side condition $freevars(F_3) \cap writes(c) = \emptyset$ is standard, it

states that: any free variable that occurs in $F_3$ is not assigned by the program $c$. This condition is needed because local variables are not involved in the concept of separation.

**Example 4.7.** *To illustrate the use of the proof rules in our system, in Listing 4.11 we present the step-by-step construction of a proof outline of the simple program discussed above in Listing 4.3 (page 70).*

## 4.4 Soundness

In this section we formulate and discuss soundness of our system. Concretely we show that our proof system guarantees that:

- verified programs are *partially correct* (Theorem 4.3);
- verified programs are *free of data races* (Theorem 4.4); and
- verified programs are *free of high-level data races* (Theorem 4.5);

We discuss correctness of the system as follows: i) we first introduce the notion of a *valid program state*, written $\sigma : \diamond$; ii) we define an invariant property over the global program state (Invariant 4.1), which ensures that every reachable program state is valid; iii) we formulate Theorem 4.3, Theorem 4.4 and Theorem 4.5, whose correctness is easy to prove using the validity of Invariant 4.1.

### 4.4.1 Valid Program States

Informally, to say that a program state $\sigma$ is valid, we should be able to find a resource $\mathcal{R}$ that is an abstraction of $\sigma$, and we should split this resource into shares $\mathcal{R} = (\mathcal{R}_{t_1} * ... * \mathcal{R}_{t_n}) * (\mathcal{R}_{l_1} * ... * \mathcal{R}_{l_m})$, such that every thread $t_i$ from the thread pool is valid over $\mathcal{R}_{t_i}$, and the resource invariant of each *free* lock $l_i$ holds over the resource $\mathcal{R}_{l_i}$. Basically, we should be able to share the state among all active parties in this state. Free locks must also be considered in this sharing, because the part of the state that is protected by a lock that is free can not be given to any active thread. Formally, a program state $\sigma$ is valid when:

$$[State] \quad \frac{\mathcal{R} = \mathcal{R}_{tp} * \mathcal{R}_{lt} \quad abstracts(\sigma, \mathcal{R}) \quad \mathcal{R}_{tp} \vdash tp : \diamond \quad \mathcal{R}_{lt} \vdash lt : \diamond}{\sigma = (h, tp, lt, it) : \diamond}$$

A lock table $lt$ is valid over a resource $\mathcal{R}$, written $\mathcal{R} \vdash lt : \diamond$ when:

$$[LockTable] \quad \frac{\mathcal{R}; \emptyset \models \circledast_{o \in dom(lt) \wedge lt(o) = \mathsf{Free}} o.res\_inv}{\mathcal{R} \vdash lt : \diamond}$$

```
{holds(I, 1) * this.y ↦¹ vy}
    //@ unpack I {
// [Unpack] rule
{(break(I, 1) ∧ this.x+this.y ≥ 0) * this.y ↦¹ vy}
  int I1 = this.y;
// [Read] rule
{(break(I, 1) ∧ this.x+this.y ≥ 0) * this.y ↦¹ vy * I1==vy}
  int I2 = I1 +1;
// [Op] rule
{(break(I, 1) ∧ this.x+this.y ≥ 0) * this.y ↦¹ vy * I1==vy * I2==I1+1}
// [PermToState] axiom
{(break(I, 1) ∧ this.x+this.y ≥ 0) * (this.y ↦¹ vy ∧ this.y ==vy) * I1==vy *
  I2==I1+1}
// [Invariant Out]
{((break(I, 1) ∧ this.x+this.y ≥ 0 * this.y ↦¹ vy) ∧ this.y ==vy) * I1==vy *
  I2==I1+1}
// [Invariant In]
{(break(I, 1) ∧ this.x+this.y ≥ 0 ∧ this.y ==vy) * this.y ↦¹ vy * I1==vy *
I2==I1+1}
// substitutions and transformations
{(break(I, 1) ∧ this.x+I2 ≥ 1) * this.y ↦¹ vy * I1==vy * I2==I1+1}
  this.y = I2;
// [Write] rule
{(break(I, 1) ∧ this.x+this.y ≥ 1) * this.y ↦¹ vy+1}
    //@ }
// [Unpack] rule
{holds(I, 1) * this.y ↦¹ vy+1}
```

Listing 4.11: Example of using the proof system

From the definition of $\circledast$ (see Appendix A), the premise of the [*LockTable*] rule can be read as: "split the resource $\mathcal{R}$ in shares, and distribute the shares among all locks that are free in that state; the resource invariant for each of these locks should hold over the associated share".

The rule for a valid thread pool over a resource $\mathcal{R}$ describes that we should find a way to split $\mathcal{R}$ in multiple shares, and distribute each share to one thread from the pool, such that the thread is valid over that share. Formally:

$$[\mathit{ThreadPool}]\quad \frac{}{\mathcal{R} \vdash \emptyset : \diamond} \qquad \frac{\mathcal{R} = \mathcal{R}_t * \mathcal{R}_{tp} \quad \mathcal{R}_t \vdash (t, s, c) : \diamond \quad \mathcal{R}_{tp} \vdash tp : \diamond}{\mathcal{R} \vdash ((t, s, c).tp : \diamond)}$$

Finally, the rule [*Thread*] shows the validity of a single thread $t$ over a resource $\mathcal{R}$:

$$[\mathit{Thread}]\quad \frac{\mathcal{R}; s \models F \quad \vdash \{F\}\ c\ \{\mathit{postcond}(\mathit{classOf}(t), \mathit{run})\}}{\mathcal{R} = (\mathcal{H}, \mathcal{L}, \mathcal{T}, \mathcal{J}) \vdash (t, s, c) : \diamond}$$

The premise of the rules states that: we should be able to prove a formula $F$ on the resource $\mathcal{R}$ and derive a proof $\{F\}\ c\ \{F'\}$ where $F'$ is the postcondition of the thread $t$ (see Appendix B).

For example if $\sigma$ is a state in which: there exist two locks $l_1$ and $l_2$ that are in a *free* state; and two threads in the thread pool, $t_1$ and $t_2$. Then $\sigma$ is a valid state if there exists a resource $\mathcal{R}$ that is an abstraction of $\sigma$ and that can be split in 4 compatible resources $\mathcal{R} = \mathcal{R}_1 * \mathcal{R}_2 * \mathcal{R}_3 * \mathcal{R}_4$, such that: $\mathcal{R}_1 \models l_1.\mathit{res\_inv}$; $\mathcal{R}_2 \models l_2.\mathit{res\_inv}$; $\mathcal{R}_3 \vdash t_1$; and $\mathcal{R}_4 \vdash t_2$;

The example below shows that a state that contains a data race is not a valid state.

**Example 4.8.** *Let $\sigma = (h, tp, lt, it)$ be a state such that the thread pool consists of two threads, $tp = (t_1, s_1, o.f = 4).(t_2, s_2, o.f = 5)$, where $o.f \in \mathit{dom}(h)$. Thus, both threads are trying to write simultaneously on the same shared location.*

*We prove that the state is not valid by contradiction. Let $\sigma : \diamond$. Then from the rule [State], there exists a resource $\mathcal{R}$ that abstracts $\sigma$, which can be divided into compatible resources, $\mathcal{R} = \mathcal{R}_1 * \mathcal{R}_2 * \mathcal{R}_{lt}$, where $\mathcal{R}_1 \vdash (t_1, s_1, o.f = 4)$ and $\mathcal{R}_2 \vdash (t_2, s_2, o.f = 5)$. Then, from $\mathcal{R}_1 \vdash (t_1, s_1, o.f = 4)$ and the rule [Thread], we have that $\vdash \{F\}\ x = 4\ \{F'\}$ and $\mathcal{R}_1, s_1 \models F$ for some $F$ and $F'$. Now, to prove $\vdash \{F\}\ x = 4\ \{F'\}$, from the Hoare triple [Write], it is possible only if $\mathcal{R}_1, s_1 \models o.f \overset{1}{\mapsto} \_$. From the semantics of the formula $o.f \overset{1}{\mapsto} \_$, we know that the abstract heap $\mathcal{H}_1$ in the resource $\mathcal{R}_1$ contains a permission 1 for $o.f$, i.e., $\mathcal{H}_1(o.f) = (\_, 1)$. Analogously, from $\mathcal{R}_1 \vdash (t_2, s_2, o.f = 5)$, we also have that $\mathcal{H}_2(o.f) = (\_, 1)$. But then, $\mathcal{H}_1$ and $\mathcal{H}_2$ are not compatible, which leads to a*

*contradiction.*

### 4.4.2 Global Program State Invariant

Next we define the invariant over the global state which states validity of all program states.

**Lemma 4.1** (Initial states)**.** *The initial global program state of any verified program with a main method $c_{main}$ is a valid state.*

*Proof.* The initial program state for the program is: $\sigma_0 = (\emptyset, (o_{main}, \emptyset, c_{main}), \emptyset, \emptyset)$. Following the definition of [State] we choose the trivial resource $\mathcal{R} = (\emptyset, \emptyset, \emptyset, \emptyset)$, for which we have $\mathit{abstracts}(\sigma_0, \mathcal{R})$. There are no free locks in this state or other threads except the main thread, thus we should just prove that $\mathcal{R} \vdash (o_{\mathsf{main}}, \emptyset, c_{main})$. This follows from $\mathcal{R} \models \mathsf{true}$ and $\vdash \{\mathsf{true}\}\ c_{main}\ \{\mathsf{true}\}$. Therefore, the initial state is a valid state, $\sigma_0 : \diamond$. □

**Invariant 4.1** (Valid states)**.** *Any state $\sigma$ of a verified program reachable at runtime, is a valid state, $\sigma : \diamond$.*

*Proof.* From Lemma 4.1, we know that the initial program state $\sigma_0$ is valid. Then, by induction over the structure of $c$ we show that: for any state $\sigma$, if $\sigma : \diamond$ and $\sigma \rightsquigarrow \sigma'$ then $\sigma' : \diamond$. The proof boils down to the following:

i) prove soundness of logical consequence rules (see Theorem 4.1); and

ii) prove soundness of the Hoare triples (see Theorem 4.2).

□

**Theorem 4.1** (Logical consequence)**.** *For every logical consequence rule $F_1, ...F_n \vdash F$ in our proof system we have: given an arbitrary resource $\mathcal{R}$ and a stack $s$, if $\mathcal{R}, s \models F_1 * ... * F_n$ then $\mathcal{R}, s \models F$.*

*Proof.* The correctness of the four axioms [*SplitMerge Perm*], [*SplitMerge Holds*], [*SplitMerge Unpack*] and [*PemToStateForm*] follows trivially from the semantics of the formulas. We prove the rule [*Invariant*].

Let $\mathcal{R}$ be a resource and $s$ a stack such that $\mathcal{R}, s \models \mathsf{holds}(o.I, \pi)$. From the semantics of the $\mathsf{holds}(o.I, \pi)$ we have that $\mathcal{T}(o.I) = (\mathsf{Packed}, \pi'), \pi' \geq \pi$. Let $\mathcal{R}$ be an abstraction of a program state $\sigma$ with an invariant table $it$, then $it(o.I) = \mathsf{Packed}$. From Invariant 4.2 defined below, it directly follows that $[\![o.I]\!]^h_s = \mathsf{true}$ and $\mathcal{R}, s \models \mathsf{holds}(o.I, \pi) \wedge o.I$. □

**Invariant 4.2** (Valid class invariants)**.** *For every program state $\sigma = (h, tp, lt, it)$ of a verified program, if $it(o.I) = \mathsf{Packed}$, then $[\![o.I]\!]^h$.*

*Proof.* We prove this by contradiction. From the operational semantics of the commands in our language, we have that there exists at least one state $\sigma_0 = (h_0, tp_0.(t_0, s_0, c_0; c_0'), lt_0, it_0)$, such that $c_0 = \mathsf{pack}\ o.I$ (the end command of an unpacked segment of $o.I$) or $c_0 = \mathsf{pack}\ o$, such that $\sigma_0$ has happened before $\sigma$, written $\sigma_0 < \sigma$. Let $\sigma_0$ be the latest program state that satisfies this condition.

From $\sigma_0 : \diamond$ (see Invariant 4.1) and the Hoare triple [*PackObj*] and [*Unpack*], we have that there exists a resource $\mathcal{R}$ such that $\mathcal{R}, s \models o.I$ and $[\![o.I]\!]^h = \mathsf{true}$. From the operational semantics of the $\mathsf{pack}\ o$ and $\mathsf{pack}\ o.I$ commands, we know that in the poststate of the command $c_0$, the invariant table maps $o.I$ to $\mathsf{Packed}$. Assume that the class invariant $o.I$ has been invalidated between the program states $\sigma_0$ and $\sigma$. Therefore, there exists a state $\sigma_1 = (h_1, tp_1.(t_1, p.f = w; c_1), lt_1, it_1)$, where $\sigma_0 < \sigma_1 < \sigma$ and $p.f \in fp(o.I)$. From Lemma 4.2 we have that in the state $\sigma_1$ $it(o.I) = \mathsf{Unpacked}$. This means that between $\sigma_1$ and $\sigma$ there is another state $\sigma_1 < \sigma_2 < \sigma$ that is a prestate of the command $\mathsf{pack}\ o.I$. This contradicts the assumption that $\sigma_0$ is the latest state satisfying the condition above. □

**Lemma 4.2.** *Let $\sigma = (h, tp.(t, s, p.f = w; c'), lt, it)$ be a state in a verified program reachable at runtime; then, for any class invariant $o.I \in dom(it)$ that refers to $p.f$, i.e., $p.f \in fp(o.I)$, $it(o.I) = \mathsf{Unpacked}$.*

*Proof.* From the definition of admissible invariants (Definition 4.3), we have that $p.f$ can be represented as $o.p_1.p_2...p_n.f$, where $n \geq 0$, $o$ is owner of $p_1$ and $p_i$ is owner of $p_{i+1}, i = 1..n-1$. We consider two cases:

(i) $n = 0$, then $p = o$. Because $\sigma : \diamond$ and from the Hoare triple [*Write*], there exists a resource $\mathcal{R}$ that abstracts $\sigma$, such that $\mathcal{R} = \mathcal{R}_t * \mathcal{R}'$ and $\mathcal{R}_t, s \models \mathsf{break}(o.I, \pi)$. From the semantics of $\mathsf{break}(o.I, \pi)$, we have that $it(o.I) = \mathsf{Unpacked}$.

(ii) $n > 0$, then $o$ is a transitive owner of $p$. Let $m$ be the method of execution of the assignment $p.f = w$. From the guarantee **RO** of the Universe type system (see Section 4.2.2), for each $p_i$, the path to the invocation of $m$ is preceded by a method call for which $p_i$ is a receiver. For convenience, we denote with $m_x$ the method call to $m$ with a receiver $x$ (i.e., $x.m(\overline{v})$). Thus, the path to $m$'s invocation is:

$$... \ m_o \ ... \ m_{p_1} \ ... \ m_{p_n} ... \ m, \text{ where } n \geqslant 1,$$

where each method $m_{p_i}$ is called by a method with a receiver $p_{i-1}$, $i = 2..n$ and $m_{p_1}$ is called by a method with a receiver $o$. We denote with $\sigma_1$ the prestate of the invocation of the method $m_{p_1}$. Because the state $\sigma$ is a prestate of assignment to $p.f$, from the precondition of the [*Write*] Hoare triple, $\mathcal{R}_t, s \models p.f \overset{1}{\mapsto} \_$. From the proof rules in our system, one can prove the validity of $p.f \overset{1}{\mapsto} \_$ in $\sigma$ in one of the following three ways:

(a) the permission $p.f \overset{1}{\mapsto} \_$ is required in the precondition of the method $m_{p_1}$. Then, from the Hoare triple [*Call*], we have that the predicate $\mathsf{break}(o.I, \pi)$ holds over the resource associated to the thread that executes the call, which implies that the state of $o.I$ in the global invariant table is $it(o.I) = \mathsf{Unpacked}$. The language syntax ensures that the end of the unpacked segment happens after the end of method $m_{p_1}$ and thus, in the state $\sigma$ it also holds that $it(o.I) = \mathsf{Unpacked}$.

(b) Permission to $p.f$ is protected by a lock $l$, and between the states $\sigma_1$ and $\sigma$ either the thread $t$ has acquired the lock $l$, or another thread $t_1$ has acquired $l$ and later has forked $f$. Then, the field $f \in \mathit{fldResInv}(L)$, where $L$ is the class of $l$ and from the rule **TR2** (see Section 4.2.2), the object $l$ must be transitively owned by $o$. Therefore, the field $l \in \mathit{relFld}(C)$, where $C$ is the class of $o$. Since $f \in \mathit{fld}(I)$ and thus, $f \in \mathit{fld}(I) \cap \mathit{fldResInv}(L)$, but from the rule **RL** (Section 4.2.2) we have $\mathit{fld}(I) \cap \mathit{fldResInv}(L) = \emptyset$, this leads to a contradiction.

(c) The location $p_1.p_2...p_n.f$ does not exist in the state $\sigma_1$, but the object $p_n$ is created in a state $\sigma_2$ between states $\sigma_1$ and $\sigma$, $\sigma_1 < \sigma_2 < \sigma$. In this case, at least one reference $p_i, i = 1..n-1$ exists in the state $\sigma_1$, which is assigned in a state $\sigma_3$, where $\sigma_1 < \sigma_3 < \sigma_2$. Since the location $p_i \in \mathit{fp}(o.I)$, permission to $p_i$ is required in the precondition of the method $m_{p_1}$. Again, the Hoare triple [*Call*] ensures that in the prestate $\sigma_1$ of the invocation of the method $m_{p_1}$, the state of the invariant $o.I$ is $it(o.I) = \mathsf{Unpacked}$. This concludes the proof.

□

**Theorem 4.2** (Hoare triples)**.** *For every Hoare triple* $\{F\}\ c\ \{F'\}$ *from the proof system (Figure 4.6) it holds: if* $\sigma = (h, tp.(t, s, c; c'), lt, it) \rightsquigarrow^* (h', tp'.(t, s', c'), lt', it') = \sigma'$ *and* $\mathcal{R}_\sigma = \mathcal{R}_t * \mathcal{R}_{lt} * \mathcal{R}_{tp}$ *is a resource for which:* $\mathit{abstracts}(\sigma, \mathcal{R}_\sigma)$, $\mathcal{R}_t, s \models F$, $\mathcal{R}_{lt} \vdash lt : \diamond$ *and* $\mathcal{R}_{tp} \vdash tp : \diamond$, *then there exists* $\mathcal{R}'_\sigma = \mathcal{R}'_t * \mathcal{R}'_{lt} * \mathcal{R}'_{tp}$ *for which:* $\mathit{abstracts}(\sigma', \mathcal{R}'_\sigma)$, $\mathcal{R}'_t, s' \models F'$, $\mathcal{R}'_{lt} \vdash lt' : \diamond$ *and* $\mathcal{R}'_{tp} \vdash tp' : \diamond$.

*Proof.* The proof is extensive but simple, when following the semantics of the language. To illustrate the proof mechanism we select a few interesting cases: the [*Acquire*] and [*Join*] axioms, which include redistribution of resources, and the [*Unpack*] rule, as a representative inference rule.

i) *Proof of the* [*Acquire*] *rule*

$$[Acquire] \quad \frac{}{\vdash \{\mathsf{initialised}(o)\}\ \mathsf{acquire}\ o\ \{o.res\_inv\ *\ \mathsf{locked}(o)\}}$$

Let $\sigma$ and $\sigma'$ be respectively the pre- and poststate of the $\mathsf{acquire}\ o$ command:

$$\sigma = (h, tp.(t, s, \mathsf{acquire}\ o; c), lt, it) \rightsquigarrow (h, tp.(t, s, c), lt', it) = \sigma'$$

Let $\mathcal{R}_\sigma$ be a resource that abstracts $\sigma$. From $\sigma : \diamond$ we have:

$$\mathcal{R}_\sigma = \mathcal{R}_t * \mathcal{R}_{lt} * \mathcal{R}_{tp} \tag{4.1}$$
$$\mathcal{R}_t, s \models \mathsf{initialised}(o) \tag{4.2}$$
$$\mathcal{R}_{lt} \vdash lt : \diamond \tag{4.3}$$
$$\mathcal{R}_{tp} \vdash tp : \diamond \tag{4.4}$$

We need to find a resource $\mathcal{R}'_\sigma$ that abstracts $\sigma'$ such that:

$$\mathcal{R}'_\sigma = \mathcal{R}'_t * \mathcal{R}'_{lt} * \mathcal{R}'_{tp} \tag{4.5}$$
$$\mathcal{R}'_t, s \models o.res\_inv\ *\ \mathsf{locked}(o) \tag{4.6}$$
$$\mathcal{R}'_{lt} \vdash lt' : \diamond \tag{4.7}$$
$$\mathcal{R}'_{tp} \vdash tp : \diamond \tag{4.8}$$

Let $S$ be a set of lock objects in the state $\sigma$ that are in a free state:

$$S = \{o_i | o_i \in dom(lt) \wedge lt(o_i) = \mathsf{Free}\}$$

Then, from (4.7) and rule [*LockTable*] (page 107), we have that:

$$\mathcal{R}_{lt} = \circledast_{o_i \in S} \mathcal{R}^{o_i} \text{ such that } \forall o_i \in S. \mathcal{R}^{o_i}; \emptyset \models o_i.res\_inv$$

Furthermore, from the operational semantics of the command $\mathsf{locked}(o)$, we

have that $lt(o) = \mathsf{Free}$ and $lt'(o) = t$. Thus, $o \in S$ and

$$\mathcal{R}_{lt} = \circledast_{o_i \in S \setminus \{o\}} \mathcal{R}^{o_i} * \mathcal{R}^o \text{ such that}$$
$$\forall o_i \in S \setminus \{o\}. \mathcal{R}^{o_i}; \emptyset \models o_i.res\_inv \wedge \mathcal{R}^o; \emptyset \models o.res\_inv$$

Now, we can choose a resource $\mathcal{R}'_{lt} = \circledast_{o_i \in S \setminus \{o\}} \mathcal{R}^{o_i}$. Because $lt'(o) \neq \mathsf{Free}$, we have $\mathcal{R}'_{lt} \vdash lt : \diamond$; thus (4.7) is satisfied. Next, we chose $\mathcal{R}'_t = \mathcal{R}'' * \mathcal{R}^o$, such that $\mathcal{R}'' = (\mathcal{H}, \mathcal{L}, \mathcal{T}, \mathcal{J})$ is equivalent to $\mathcal{R}_t$ except that $\mathcal{L}(o)^3 = \mathsf{true}$. We get $\mathcal{R}^o, s \models o.res\_inv$ and $\mathcal{R}'', s \models \mathsf{locked}(o)$, from where it follows that (4.6) holds. Finally, we can satisfy (4.8) by choosing $\mathcal{R}'_{tp} = \mathcal{R}_{tp}$. Resources $\mathcal{R}'_t$, $\mathcal{R}'_{lt}$ and $\mathcal{R}'_{tp}$ are compatible and by joining them we obtain a resource $\mathcal{R}'_\sigma$ that abstracts the state $\sigma'$, which is required by the statement (4.5). This concludes the proof.

ii) *Proof of the* [*Join*] *rule*

$$[Join] \quad \frac{t : T \quad F' = postcond(T^2, run)}{\vdash \{\mathsf{join}(t)\}\ \mathsf{join}\ t\ \{F'[t/\mathsf{this}]\}}$$

Let $\sigma$ and $\sigma'$ be respectively the pre- and poststate of the $\mathsf{join}\ t$ command:

$$\sigma = (h, tp.(t_0, s_0, \mathsf{join}\ t; c_0).(t, s, \mathsf{return}), lt, it) \rightsquigarrow (h, tp.(t_0, s_0, c_0), lt, it) = \sigma'$$

Let $\mathcal{R}_\sigma$ be a resource that abstracts $\sigma$. From $\sigma : \diamond$ we have:

$$\mathcal{R}_\sigma = \mathcal{R}_{t_0} * \mathcal{R}_{lt} * \mathcal{R}_{tp} \tag{4.9}$$
$$\mathcal{R}_{t_0}, s_0 \models \mathsf{join}(t) \tag{4.10}$$
$$\mathcal{R}_{lt} \vdash lt : \diamond \tag{4.11}$$
$$\mathcal{R}_{tp} \vdash tp \cup \{t\} : \diamond \tag{4.12}$$

We need to find a resource $\mathcal{R}'_\sigma$ that abstracts $\sigma'$ such that:

$$\mathcal{R}'_\sigma = \mathcal{R}'_{t_0} * \mathcal{R}'_{lt} * \mathcal{R}'_{tp} \tag{4.13}$$
$$\mathcal{R}'_{t_0}, s_0 \models F'[t/\mathsf{this}] \tag{4.14}$$
$$\mathcal{R}'_{lt} \vdash lt' : \diamond \tag{4.15}$$
$$\mathcal{R}'_{tp} \vdash tp : \diamond \tag{4.16}$$

From (4.12) and rule [*ThreadPool*] (page 109), we have that:

$$\mathcal{R}_{tp} = \circledast_{t_i \in dom(tp)} \mathcal{R}^{t_i} * \mathcal{R}^{t} \text{ such that}$$
$$\forall t_i \in dom(tp).\mathcal{R}^{t_i} \vdash (t_i, s^{t_i}, c^{t_i}) \land \mathcal{R}_t \vdash (t, s, \mathsf{return}).$$

Next, from $\mathcal{R}_t \vdash (t, s, \mathsf{return})$ and rule [*Thread*] (page 109) we have that $\mathcal{R}_t, s \models F$ and $\vdash \{F\}\ \mathsf{return}\ \{F'\}$ and therefore, $\mathcal{R}_t, s \models F'$.

Now, we can choose a resource $\mathcal{R}'_{lt} = \mathcal{R}_{lt}$, from where (4.15) is satisfied. Next, we choose $\mathcal{R}'_{tp} = \circledast_{t_i \in dom(tp)} \mathcal{R}_{t_i}$, from where $\mathcal{R}'_{tp} \vdash tp'$ and thus, (4.16) holds. Finally, we chose $\mathcal{R}'_{t_0} = \mathcal{R}_{t_0} * \mathcal{R}_t$. Because $\mathcal{R}_t, s \models F'$, we have $\mathcal{R}_t, s_0 \models F'[t/\mathsf{this}]$, from where (4.14) holds. Resources $\mathcal{R}'_{t_0}$, $\mathcal{R}'_{lt}$ and $\mathcal{R}'_{tp}$ are compatible and by joining them we obtain a resource $\mathcal{R}'_\sigma$ that abstracts the state $\sigma'$, which is required by (4.13). This concludes the proof.

iii) *Proof of the* [*Unpack*] *rule*

$$[Unpack] \quad \frac{\vdash \{\mathsf{break}(o.I, 1)\ \land\ o.I\}\ c\ \{\mathsf{break}(o.I, 1)\ \land\ o.I\}}{\vdash \{\mathsf{holds}(o.I, 1)\}\ \mathsf{unpack}\ o.I\{c\}\ \{\mathsf{holds}(o.I, 1)\}}$$

Let $\sigma$ and $\sigma'$ be respectively the pre- and poststate of the $\mathsf{unpack}\ o.I\{c\}$ command:

$$\sigma = (h, tp.(t, s, \mathsf{unpack}\ o.I\{c\}; c'), lt, it) \rightsquigarrow^* (h', tp'.(t, s', c'), lt', it') = \sigma'$$

Let $\mathcal{R}_\sigma$ be a resource that abstracts $\sigma$. From $\sigma : \diamond$ we have:

$$\mathcal{R}_\sigma = \mathcal{R}_t * \mathcal{R}_{lt} * \mathcal{R}_{tp} \tag{4.17}$$
$$\mathcal{R}_t, s \models \mathsf{break}(o.I, 1)\ \land\ o.I \tag{4.18}$$
$$\mathcal{R}_{lt} \vdash lt : \diamond \tag{4.19}$$
$$\mathcal{R}_{tp} \vdash tp : \diamond \tag{4.20}$$

We need to find a resource $\mathcal{R}'_\sigma$ that abstracts $\sigma'$ such that:

$$\mathcal{R}'_\sigma = \mathcal{R}'_t * \mathcal{R}'_{lt} * \mathcal{R}'_{tp} \tag{4.21}$$
$$\mathcal{R}'_t, s' \models \mathsf{holds}(o.I, 1) \tag{4.22}$$
$$\mathcal{R}'_{lt} \vdash lt' : \diamond \tag{4.23}$$
$$\mathcal{R}'_{tp} \vdash tp' : \diamond \tag{4.24}$$

Let $\sigma_1 = (h, tp.(t, s, c; \mathsf{pack}\ o.I; c'), lt, it)$. Then we can choose a resource $\mathcal{R}_1 = (\mathcal{H}_1, \mathcal{L}_1, \mathcal{T}_1, \mathcal{J}_1)$ such that it is equivalent to $\mathcal{R}_\sigma$, except that $\mathcal{T}_1(o.I) = (\mathsf{Unpacked}, 1)$. Then $\mathcal{R}_1$ abstracts the state $\sigma_1$ such that: $\mathcal{R}_1 = \mathcal{R}^1_{lt} * \mathcal{R}^1_{tp} * \mathcal{R}^1_t$, $\mathcal{R}^1_{lt} \vdash lt$, $\mathcal{R}^1_{tp} \vdash tp$, and $\mathcal{R}^1_t \models \mathsf{break}(o.I, 1)$. (We choose $\mathcal{R}^1_{lt}$, $\mathcal{R}^1_{tp}$ and $\mathcal{R}^1_t$ to be equivalent to $\mathcal{R}_{lt}$, $\mathcal{R}_{tp}$ and $\mathcal{R}_t$ respectively, except that the abstract invariant map in each $\mathcal{R}^1_{lt}$, $\mathcal{R}^1_{tp}$ and $\mathcal{R}^1_t$ maps $o.I$ to $(\mathsf{Unpacked}, 1)$). From the logical consequence [*Invariant*] (page 103) we have $\mathcal{R}^1_t \models \mathsf{break}(o.I, 1) \wedge o.I$.

Furthermore, let $\sigma_2 = (h', tp'.(t, s', \mathsf{pack}\ o.I; c'), lt', it'[o.I \mapsto \mathsf{Unpacked}])$. Then from the premise of the [*Unpack*] Hoare triple, we know that there exists a resource $\mathcal{R}_2$ that abstracts the state $\sigma_2$ such that: $\mathcal{R}_2 = \mathcal{R}^2_{lt} * \mathcal{R}^2_{tp} * \mathcal{R}^2_t$, $\mathcal{R}^2_{lt} \vdash lt'$, $\mathcal{R}^2_{tp} \vdash tp'$, and $\mathcal{R}^2_t \models \mathsf{break}(o.I, 1) \wedge o.I$.

Then, the state $\sigma'$ can be abstracted by a resource $\mathcal{R}'_\sigma = (\mathcal{H}_2, \mathcal{L}_2, \mathcal{T}_2, \mathcal{J}_2)$, which is equivalent to the resource $\mathcal{R}_2$ except that $\mathcal{T}_2(o.I) = (\mathsf{Packed}, 1)$. Thus, $\mathcal{R}'_\sigma$ can be split into $\mathcal{R}'_t$, $\mathcal{R}'_{lt}$ and $\mathcal{R}'_{ts}$ (each of them equivalent respectively to $\mathcal{R}^2_{lt} * \mathcal{R}^2_{tp} * \mathcal{R}^2_t$, except that their abstract invariant map maps $o.I$ to $(\mathsf{Packed}, 1)$), for which the statements (4.22), (4.23) and (4.24) are satisfied, which concludes the proof.

□

### 4.4.3 Soundness Theorems

Next, we formulate and discuss the properties that our verification system satisfies: partial correctness, data race-freedom and high-level data race-freedom.

**Lemma 4.3** (Monotonicity of resources)**.** *If $\mathcal{R}, s \models F$, then for any extended resource $\mathcal{R} * \mathcal{R}'$ we have $\mathcal{R} * \mathcal{R}', s \models F$.*

*Proof.* The intuition of this lemma is simple: if a formula $F$ holds over a resource $\mathcal{R}$, it holds over any extension of $\mathcal{R}$. The proof is by structural induction on $F$. We give here only an example.

Let the formula $F$ be $o.f \overset{\pi}{\mapsto} v$, and $\mathcal{H}$ and $\mathcal{H}'$ are abstract heaps from both resources $\mathcal{R}$ and $\mathcal{R}'$, respectively. From $\mathcal{R}, s \models o.f \overset{\pi}{\mapsto} v$ we have $\mathcal{H}(o.f) = (v, \pi'), \pi' \geq \pi$. For the abstract heap $\mathcal{H}'$ which is compatible with $\mathcal{H}$ ($\mathcal{H} \# \mathcal{H}'$) we have $\mathcal{H}'(o.f) = (v, \pi''), \pi'' \geq 0$. Thus, for the joined abstract table we have $(\mathcal{H} * \mathcal{H}')(o.f) = (v, \pi' + \pi''), \pi' + \pi'' \geq \pi$. Therefore, $\mathcal{R} * \mathcal{R}', s \models o.f \overset{\pi}{\mapsto} v$. □

**Theorem 4.3** (Partial correctness)**.** *If $c$ is a verified program with an initial state $\sigma_0$ and $\sigma_0 \rightsquigarrow^* \sigma$ where $\sigma = (h, tp.(t, s, \mathsf{assert}\ F), lt, it)$, then there exists a resource $\mathcal{R}$ such that $abstracts(\sigma, \mathcal{R})$ and $\mathcal{R}, s \models F$.*

*Proof.* The proof is trivial when using Invariant 4.1. Because the state $\sigma$ is reachable at runtime, it is a valid state, $\sigma : \diamond$. From the definition of [*State*] we have that $\exists \mathcal{R}_\sigma.\mathit{abstracts}(\sigma, \mathcal{R}_\sigma)$ and $\mathcal{R}_\sigma = \mathcal{R}_t * \mathcal{R}'$, such that $\mathcal{R}_t \vdash (t, s, \mathsf{assert}\ F)$. From the definition of [*Thread*] we have that $\mathcal{R}_t, s \models F$. If $F$ holds over a resource $\mathcal{R}_t$, from Lemma 4.3, it also holds over $\mathcal{R}_t * \mathcal{R}'$; thus for $\mathcal{R}_\sigma$ we have $\mathcal{R}_\sigma, s \models F$. This concludes the proof. □

**Theorem 4.4** (Data race-freedom)**.** *A verified program never reaches a state $\sigma$ that contains a data race, i.e., $\sigma = (h, tp.(t_1, s_1, c_1; c_1').(t_2, s_2, c_2; c_2'), lt, it)$, where $c_1$ is $o.f = w$ and $c_2$ is either $o.f = w'$ or $x = o.f$.*

*Proof.* The proof is by contradiction. Let $\sigma$ be a state reachable at runtime. Then from Invariant 4.1 we have that $\sigma : \diamond$. This means that there exists a resource $\mathcal{R}_\sigma$ such that: $\mathit{abstracts}(\sigma, \mathcal{R}_\sigma)$ and $\mathcal{R}_\sigma = (\mathcal{R}_{t_1} * \mathcal{R}_{t_2}) * \mathcal{R}'$, where $\mathcal{R}_{t_1} \vdash (t_1, s_1, c_1; c_1')$, $\mathcal{R}_{t_2} \vdash (t_1, s_1, c_2; c_2')$ and $\mathcal{R}_{t_1} \# \mathcal{R}_{t_2}$. Because the command $c_1$ is an assignment to the location $o.f$ and from the precondition of the Hoare triple [*Write*], we have that $\mathcal{R}_{t_1}, s_1 \models o.f \overset{1}{\mapsto} \_$. Similarly, because $c_2$ is reading or writing at location $o.f$, from the Hoare triples [*Read*] and [*Write*], we have $\mathcal{R}_{t_2}, s_2 \models o.f \overset{\pi}{\mapsto} \_, \pi > 0$. Furtermore, from the semantics of the formula $o.f \overset{\pi}{\mapsto} \_$ we have that $\mathcal{H}_{t_1}(o.f) = (\_, 1)$ and $\mathcal{H}_{t_2}(o.f) = (\_, \pi), \pi > 0$ where $\mathcal{H}_{t_1}$ and $\mathcal{H}_{t_2}$ are the abstract heaps of $\mathcal{R}_{t_1}$ and $\mathcal{R}_{t_2}$, respectively. Therefore, for the sum of the permissions in both resources we get $1 + \pi > 1$, and thus $\mathcal{H}_{t_1} \# \mathcal{H}_{t_2}$ does not hold, which leads to a contradiction. □

**Theorem 4.5** (High-level data race-freedom)**.** *If a location $p.f$ is in a critical state $\sigma$ of an unpacked segment of a class invariant $o.I$, then any non-local thread of this segment can not access $p.f$.*

*Proof.* Let $\sigma_1 = (h_1, tp_1.(t_1, s_1, \mathsf{unpack}\ o.I\{c_1\}; c_1'), lt_1, it_1)$ be a prestate of an unpacked segment, $\sigma_2 = (h_2, tp_2.(t_1, s_2, c_1'), lt_2, it_2)$ be the poststate of this segment, and $\sigma = (h, tp.(t, s, p.f = w; c'), lt, it)$ be a prestate of an assignment that happens within the unpacked segment, i.e., $\sigma_1 < \sigma < \sigma_2$. We should prove that the thread $t$ is local to the unpacked segment.

From the Hoare triple [*Write*], we should be able to prove the formula $p.f \overset{1}{\mapsto} \_$ for a resource $\mathcal{R}_t$ that abstracts the state $\sigma$. From the syntax of the language, lock-related commands are not allowed within an unpacked segment. Therefore, one can prove $p.f \overset{1}{\mapsto} \_$ over $\mathcal{R}_t$ only if $p.f \overset{1}{\mapsto} \_$ holds over a resource $\mathcal{R}_{t_1}$ that abstracts the state $\sigma_1$ and $t$ is either equal to $t_1$ or is forked by $t_1$ or any other local thread to the segment. In other words, the syntax of the unpacked segment

ensures that within the segment, no permission could leak to a thread that is non-local to the segment. Therefore $t$ is a local thread to the segment. □

## 4.5 Related Work and Conclusions

Class invariants are essential in the verification of object-oriented software, they build the core principles in major specification languages as JML [LPC+07] or Spec# [BLS05]. They allow one to specify properties expected to be maintained continuously, during the entire program execution. A verification technique should allow a class invariant to be temporarily broken, but ensure that it is preserved in all *visible* program states. Sadly, the benefit of using class invariants comes together with serious verification challenges. First, modular verification is hardly achievable without imposing certain restrictions to the program and to the definition of the class invariants. Second, in the presence of multiple threads, it becomes unclear which are the visible states of the program and where exactly a class invariant must hold.

**Related work** The early work on verification of class invariants in sequential programs [Mey97, LG86] is unsound for more complex data structures, for example if a class invariant captures properties over multiple objects. Later, Poetzsch-Heffter [PH97] and Huizing *et al.* [HK00] developed sound techniques that do not restrict the invariant definition or the program itself; however, these approaches are not modular.

Müller *et al.* [MPHL06] propose two sound techniques for modular reasoning: the *ownership technique* and the less restrictive *visibility technique*. Both concepts, as well as Lu *et al.*'s modular technique [LPX07], are designed for ownership-based type systems. These techniques are captured in an abstract unified framework, developed and formalised by Drossopoulou *et al.*'s [DFMS08]. This framework however, has not been applied on a concrete verification technique for concurrent programs.

Weiß models class invariants with a boolean model field *inv* [Wei11]. Their validity is checked only on demand. Specifications use *inv* explicitly where needed, while *this.inv* is implicitly generated in the pre- and postcondition of every method.

The techniques above are applicable in sequential programs only. We are not aware of much work done on verification of class invariants for multithreaded programs. Comparable to our approach is Jacobs *et al.*'s technique [JPLS05] for verifying multithreaded programs with class invariants, based on the *Boogie*

*methodology* [BDF+04] for sequential programs. This technique is based on the following protocol. A thread can lock (acquire) an object $o$ and become its owner. To modify any field of the object $o$, the thread owner must first unpack the object $o$. Unpacking brings the object in a *mutable* state, in which $o$'s invariant may be invalidated. This invalidating is safe, because the owner thread has a full ownership of the object, and no other thread may observe the invalidated object $o$. To pack the object $o$, the owner thread must show that the object is *consistent* (its invariant holds); a packed object can then be released. Modularity is provided by using an ownership-based technique. However, in contrast to our technique, this technique allows a thread to break an invariant of an object only if it completely owns the object (and therewith all transitively owned objects).

A different approach for modular verification of object invariants in concurrent programs is proposed by Cohen [CMST10], implemented in VCC [CDH+09b]. Each object is assigned a two-state invariant expressing the relation that needs to be preserved between any two consecutive states. A verification mechanism provides a semantic check whether all invariants are *admissible*, i.e., preserved by the execution of all updates. For example if in our class LinePoint (see Listing 4.10) we define a class invariant I1: y $\geq$ \old(y), and in the class Line (see Listing 4.9) a class invariant I2: p1.y $\geq$0, both invariants are admissible: I1 is established by all local updates in LinePoint, while I2 is guaranteed because of the validity of I1. However, the invariant that we currently have defined in LinePoint in Listing 4.9 (I: p1.x-p2.x>5 $\vee$ p2.x-p1.x>5) is not admissible, because it cannot be proved to be preserved after any update. The authors state how invariants can be defined to be made admissible. The disadvantage of the approach is that it brings a high specification overhead.

**Our technique** The technique we introduced in this chapter is a novel approach for verifying class invariants in multithreaded programs. It builds on a variant of permission-based separation logic, but deviates from the standard rules of this logic. In particular, we allow class invariants to express properties only over state and thus, their definition is free of permission expressions.

To control validity of class invariants we use special predicates associated with each invariant: a fraction holds($o.I, \pi$) ensures that the invariant holds; while a fraction break($o.I, \pi$) gives right to a thread to break the invariant. Breaking happens in a controlled manner, the segment is explicitly marked as *unpacked*, and the "breaking task" may be delegated only to threads that are *local* to the unpacked segment. No external thread may observe that the

invariant is invalidated. These additional predicates give more flexibility in how permissions are distributed. In particular, breaking and re-establishing an invariant is allowed without holding all permissions associated to the invariant. To achieve modularity, we use an ownership-based type system.

*Future directions* We have not discussed how class invariants could be expressed using abstract predicates or model methods. Class inheritance is also not treated by our technique. We do not expect that extending the approach to support these concepts would lead to complications; however, a more detailed research in this direction will be particularly useful.

Another interesting challenge is to make the technique more permissive by allowing arbitrary parallel threads to break simultaneously the same class invariant. For example, for the class invariant $\mathsf{x}+\mathsf{y} \geq 0$ in the class Point (Listing 4.5), it should be possible that two threads increase x and y respectively; none of these updates will "harm" the invariant. The presented technique currently allows this but only when the invariant is unpacked and both threads are local to the unpacked segment. This challenge is already addressed in this thesis. Our new proposed technique uses the intuition of history-based reasoning, and therefore is presented later in Section 5.7.

# Chapter 5

# History-based Verification of Functional Properties

Class invariants are a useful, but insufficient mechanism to describe functional properties of programs. They describe stability, but are not suitable to describe how the program state changes over time. To describe change, assertions can be added at specific control points in the program; whenever a thread execution reaches that point, the assertion property must hold. Furthermore, to achieve modular verification, assertions must be placed in the pre- and poststate of each method in the form of a method contract.

In a concurrent program, however, we can state that an assertion holds when a thread reaches its control point, only when this assertion is stable and cannot be invalidated by any other parallel thread. This means that the behaviour of a synchronised block can easily be described by assertions placed in the synchronised segment, because this code is protected from any thread interleaving. However, specifying method contracts can often be a problem, because synchronisation of methods is usually *internal*, and therefore, their pre- and poststates are not "safe" points to place the desired assertions.

In Chapter 1 we introduced the following question:

> *Challenge 2: How to specify and verify expressive specifications that describe the functional behaviour of multithreaded programs?*

This chapter responds to this question with a new technique for reasoning about functional behaviour of concurrent programs. The technique extends

permission-based separation logic with a new *history-based* mechanism. Histories, represented as *process algebra terms* [GR01], allow modular specifications that are both expressive and intuitive. We formalise our technique and prove it sound. For this we use the language and formalisation already presented in Chapter 4.3, here extended with a new mechanism for management of histories. The technique has been implemented into our verification tool VerCors.

**Outline** The chapter is structured as follows: First in Section 5.1 we identify the problem of verifying functional properties in concurrent programs and study its meaning in a separation logic-based environment. In Section 5.2 we give a short background on the process algebra language we use. In Section 5.3 we give an informal, but comprehensive explanation of our technique. In Section 5.4 we illustrate our approach on a few examples. We formalise our approach in Section 5.5: Section 5.5.1 presents the language, Section 5.5.2 its semantics, Section 5.5.3 the proof system, and Section 5.5.4 proves soundness. In Section 5.6 we discuss the implementation of the approach in our verification tool VerCors. In Section 5.7 we revisit the technique for concurrent class invariants and show how this technique can be improved when applying the concept of histories. Finally in Section 5.8 we conclude and discuss related work.

## 5.1 The Problem of Functional Verification in Concurrent Programs

**The Owicki-Gries approach** In Chapter 1.3.2 we sketched the problem of specifying method behaviour on a simple, well-known example originating from Owicki and Gries's seminal paper [OG76b]. Here we present this example in its original form (see Listing 5.1), and discuss the solution of Owicki and Gries.

Listing 5.1 presents two threads running in parallel, each of them incrementing the value of a shared location x by 1. The command cobegin c1 // c2 coend denotes parallel execution of two program commands. Line 3 shows that the location x is protected by the resource r (a resource is equivalent to a lock); this means that x can be accessed only within the critical section of the resource r (when the lock is held). The command **with** r **when** b **do** c indicates that c is a critical section of r and can be entered when b is true. If the value of x initially was 0 (line 2), we want to prove that after both threads have finished their updates (line 20), the value of x equals 2.

Owicki and Gries's solution to verify this program uses *auxiliary (specification-only) variables*. Each thread uses its own auxiliary variable (a and b, respect-

ively) to keep track of its local state. Thus, all local changes have to be tracked explicitly by updates to the auxiliary variables that are specified by the programmer. The resource r also protects a and b (line 3). A *resource invariant* I(r) in line 1 specifies the *invariant property* that relates the value of x to the auxiliary variables a and b. After both threads finish their execution, the values of a and b as well as the invariant I(r) (line 20) are used to prove that the value of the shared variable x equals 2.

```
   /* I(r) = {x=a+b} */
2  {a=0 ∧ b=0 ∧ x=0}
   resource r(x,a,b): cobegin
4  {a = 0}
     with r when true do
6          begin
             x:=x+1;
8            a:=1;
           end
10 {a = 1}
   //
12 {b = 0}
     with r when true do
14         begin
             x:=x+1;
16           b:=1;
           end
18  {b = 1}
   coend
20 {a=1 ∧ b=1 ∧ I(r)}
   {x=2}
```

Listing 5.1: Counter example, Owicki and Gries's solution

**A modular extension** As observed by Jacobs and Piessens [JP11], this approach does not generalise to method calls: if the code for acquiring a resource r and updating the value of x (lines 5 - 9) is inside a method incr(), the same method cannot be called by both threads, because each thread requires an update on a different auxiliary variable.

To resolve this, Jacobs and Piessens propose a logic that allows one to augment the client program with auxiliary update code (as a higher-order parameter) that is passed as an argument to the method. For example, for the incr() method, the user has to add ghost code a = 1 or b = 1, respectively to both method calls. This logic is expressive and supports various examples; however, the drawback of the approach is that it requires a concrete invariant property to be specified that should remain stable under the updates of all threads. Moreover, the user needs to provide the concrete updates to the local state in an explicit way at each method call, which imposes a large overhead on verification.

*Our solution* We propose an alternative approach to reason about concurrent programs, while using the power of *process algebra*. Concretely, we use *histories*, represented as process algebra terms, to capture abstractly the behaviour of part of the program. This allows one to specify the local behaviour of

a method intuitively in terms of actions added to a local history; local histories can be combined into global histories, and by resolving the global histories, the reachable state properties can be determined. The approach is based on a variant of *permission-based separation logic*. As a novelty, we extend the definition of the *separating conjunction (∗)* to allow splitting and merging histories. The main advantage of the method is that it allows intuitive and expressive specifications.

## 5.2 Background: the $\mu$CRL Language

We use the µCRL [GR01] process algebra to model histories. Process algebras in general are languages that provide an algebraic approach to study the behaviour of concurrent systems. Basic entities of these languages are *actions*, and process algebra terms are built of actions using several composition operators. By applying transformations on the process algebra term, one can derive properties about the behaviour of the system.

Variants of process algebras have been developed in the beginning of the eighties, most common of them are ACP (Algebra of Communicating Processes) [BK84], CSP (Communicating Sequential Processes) [Hoa78] and CCS (Calculus of Communicaing Systems) [Mil82]. These are basic process algebras suitable for studying some elementary properties of interactions between systems. The process algebra that we use, µCRL, extends ACP with *data*, thus allowing parameterised process algebra terms. This makes µCRL more powerful, and sufficiently expressive for our needs.

*Basic entities* Actions as basic primitives in a process algebra language are names from a set $\mathcal{A} = \{a, b, c...\}$, each of them representing an atomic (indivisible) behaviour. An action has just a name, and no interpretation. In this way, process algebras are modelling languages that give an abstract view of a concurrent system, without taking into account the nature of the system.

Apart from the atomic actions $a, b, c, ...$, µCRL also admits:

- a *deadlock action* $\delta$, which represents a deadlock (stagnation); and
- a *silent action* $\tau$, which represents an action without behaviour.

*Algebraic operators* Actions are used to structurally build more complex processes $p$. With $\epsilon$ we denote the *empty process*. Composition of actions and processes is done using the following operators:

- *sequencing composition*: $p_1 \cdot p_2$ describes sequential execution of the two processes $p_1$ and $p_2$. The following basic axioms describe sequencing composition with the silent and the deadlock action: $\tau \cdot p = p \cdot \tau = p$ and $\delta \cdot p = p \cdot \delta = \delta$.

- *alternative composition*: $p_1 + p_2$ describes choice: either $p_1$ or $p_2$ will execute. Alternative composition with a deadlock action is: $\delta + p = p$.

- *parallel composition*: $p_1 \parallel p_2$ describes parallel execution of $p_1$ or $p_2$. Parallel composition is defined as all possible interleavings between both processes, using the *left merge* ($\lfloor\!\lfloor$) and *communication merge* ($|$) operators:

$$p_1 \parallel p_2 = (p_1 \lfloor\!\lfloor p_2) + (p_2 \lfloor\!\lfloor p_1) + (p_1 \mid p_2)$$

  The operator $\lfloor\!\lfloor$ defines a parallel composition of two processes where the initial step is always the first action of the left-hand operator:

$$(a \cdot p_1) \lfloor\!\lfloor p_2 = a \cdot (p_1 \parallel p_2)$$

  The operator $|$ defines a parallel composition of two processes where the first step is a communication between the first actions of each process:

$$a \cdot p_1 \mid b \cdot p_2 = a \mid b \cdot (p_1 \parallel p_2)$$

  The result of a communication between two actions is defined by a function

$$\gamma : \mathcal{A} \times \mathcal{A} \mapsto \mathcal{A}, \quad \text{i.e., } a \mid b = \gamma(a, b)$$

  **Example 5.1.** *Let $p$ be a process algebra term defined as $(a \cdot b) \parallel c$, where $a$, $b$, $c$ are elements from the set of actions $\mathcal{A}$ and the communication function $\gamma$ is defined as $\forall a_1, a_2 \in \mathcal{A}.\gamma(a_1, a_2) = \delta$. The process $p$ can be transformed into a sum of three possible traces (interleavings):*

$$\begin{aligned} p = (a \cdot b) \parallel c = & (a \cdot b) \lfloor\!\lfloor c + c \lfloor\!\lfloor (a \cdot b) + (a \cdot b) \mid c = \\ & a \cdot (b \parallel c) + c \cdot (a \cdot b) + (a \mid c) \cdot b = \\ & a \cdot (b \lfloor\!\lfloor c + c \lfloor\!\lfloor b + b \mid c) + c \cdot a \cdot b + \delta \cdot b = \\ & a \cdot (b \cdot c + c \cdot b + \delta) + c \cdot a \cdot b + \delta = \\ & a \cdot b \cdot c + a \cdot c \cdot b + c \cdot a \cdot b \end{aligned}$$

- *abstraction operator*: $\tau_{\mathcal{A}'}(p)$ hides certain actions: it renames all occur-

rences of actions from the set $\mathcal{A}'$ by $\tau$. For example, $\tau_{\{b\}}(a \cdot b + c) = a \cdot \tau + c = a + c$.

- *encapsulation operator*: $\partial_{\mathcal{A}'}(p)$ disables unwanted actions by replacing all occurrences of actions in $\mathcal{A}'$ by $\delta$. For example, $\partial_{\{b\}}(a \cdot b + c) = a \cdot \delta + c = c$.
- *conditional operator*: $p \lhd b \rhd q$ describes the behaviour of $p$ if $b$ is $\mathsf{true}$ and the behaviour of $q$ otherwise.
- *sum operator*: $\sum_{d:D} p(d)$ represents a possibly infinite choice over data of type $D$.

## 5.3 The Concepts of History-Based Reasoning

This section gives an informal but detailed description of our methodology. We illustrate our approach on a version of the classical Owicki-Gries example, presented in Listing 5.2 in our object-oriented language. The class Counter represents a shared counter containing a single integer property x. A location c.x can be accessed only by a thread holding the lock c.

To specify the Counter class, assume that we use the classical approach: we associate a resource invariant to the lock, defined as $\mathsf{res\_inv} = \mathsf{x} \overset{1}{\mapsto} \mathsf{v}$ [O'H07, AHHH14]. However, this predicate stores not only the access permission to x, but also information about the value of x. As the method incr() uses internal synchronisation, after the lock is released in line 30, the predicate $\mathsf{res\_inv} = \mathsf{x} \overset{1}{\mapsto} \mathsf{v}$ will be transferred to the lock, and therewith, the current thread will lose all information about the value of x. This makes describing the method's functional behaviour in the postcondition problematic.

Therefore, our approach aims to separate permissions to locations from their values (the functional properties). A resource invariant can be used to store permissions to access a location, while information about the value stored at this location is treated separately, by using a *history*.

**Histories** A history refers to a set of locations $L$ and is called *a history over* $L$. It records all updates made to any of the locations in $L$. The same location cannot appear in more than one existing history simultaneously.

We use a predicate $\mathsf{Hist}(L, 1, R, H)$, called a *history predicate*, to capture a history over locations $L$. This contains complete knowledge about the *initial state* $\sigma$ of the history (i.e., the state when no action has been recorded in the history), and about all updates to the locations in $L$ after the state $\sigma$. We call the

```
class Counter {
  int x;
//@ pred res_inv = x ↦¹ₕ _;

/*@ accessible {x};
  @ assignable {x};
  @ requires k>0;
  @ ensures x=\old(x)+k;
  @ action inc(int k);
@*/

//@ requires x ↦¹ _;
//@ ensures x ↦¹ vx;
 Counter(int vx){
   this.x = vx;
 }
//@ requires Hist(L, π, R, H) * x ∈ L *
//@     initialised(this);
//@ ensures Hist(L, π, R, H·inc(1));
 void incr(){
  acquire this;
{Hist(L, π, R, H) * x ↦¹ₕ v}
//@ action inc(1){
{Hist(L, π, R, H) * x ↦¹ₐ v}
      x = x+1;
{Hist(L, π, R, H) * x ↦¹ₐ v+1}
//@ }
{Hist(L, π, R, H·inc(1)) * x ↦¹ₕ v+1}
  release this;
{Hist(L, π, R, H·inc(1))}
 }
}
```

Listing 5.2: The Counter example

values of the locations in $L$ in the initial state *initial values*, while the predicate $R$ is called *the initial property*: it captures the knowledge about the initial values. More precisely, $R$ is a predicate over $L$, such that $R[\sigma(l)/l]_{\forall l \in L}$ holds, where $\sigma(l)$ denotes the value of $l$ in state $\sigma$. Further, $H$ is a µCRL process, which records the *behaviour of* $L$, i.e., the *history of updates* over locations in $L$. The second parameter $\pi$ in the history predicate is used to make it a splittable predicate: a predicate $\mathsf{Hist}(L, \pi, R, H)$, where $\pi < 1$ contains only *partial* knowledge about the behaviour of $L$.

**Creating a history** A history over $L$ is created by the specification command $\mathsf{crhist}(L, R)$, where $R$ is a boolean formula over locations in $L$ that holds in the current state. This command requires a full permission predicate $l \overset{1}{\mapsto} v$ for each location $l \in L$, converts it to a new *history permission predicate*, i.e., $l \overset{1}{\mapsto}_h v$, and produces a history predicate $\mathsf{Hist}(L, 1, R, \epsilon)$. The predicate $l \overset{1}{\mapsto}_h v$ has essentially the same meaning as $l \overset{1}{\mapsto} v$: a splittable predicate that keeps the access permission for the location $l$ and its current *local* value $v$. However, the mark $h$ indicates that the location $l$ is bound to an existing history, and every change of $l$ must be recorded in this history. Consuming the permission

predicate when creating the history ensures that the same location can be traced by at most one history at the time.

In the example in Listing 5.2, the lock's resource invariant is defined via a predicate $\mathsf{x} \overset{1}{\mapsto}_h \_$, instead of $\mathsf{x} \overset{1}{\mapsto} \_$ (line 3). This means that while the permission to update x is stored in the lock, independently there exists a history that refers to x and records all updates to x.

Listing 5.3 presents the Client class that uses a Counter. The client creates the Counter object c, obtaining the full $\mathsf{c.x} \overset{1}{\mapsto}_h 0$ predicate (line 5). It then creates a history over a single location c.x (line 6) and exchanges the $\mathsf{c.x} \overset{1}{\mapsto} 0$ predicate for the predicates $\mathsf{c.x} \overset{1}{\mapsto}_h 0$ and $\mathsf{Hist}(\{\mathsf{c.x}\}, 1, \mathsf{c.x}{==}0, \epsilon)$. After the lock is committed, (line 8) and the permissions are transferred to the lock, the client still keeps the full history predicate. This guarantees that no other thread may update the location c.x until the history predicate is split; the value is stable even without holding any access permission to c.x.

**Splitting and merging of histories** The history may be redistributed among parallel threads by splitting the predicate $\mathsf{Hist}(L, \pi, R, H)$ into two separate predicates, with histories $H_1$ and $H_2$, such that $H = H_1 \parallel H_2$. Each predicate is used by one parallel thread, and each thread records its own updates in its own partial history. The basic idea is to split the history $H$ such that $H_1 = H$ and $H_2 = \epsilon$. However, this should be done in such a way that if we later merge the two histories, we know at which point $H$ was split. More specifically, if we split $H$, and then one thread does an action $a$, and the other thread an action $b$, and then the histories are merged, this should result in a history $H \cdot (a \parallel b)$.

To ensure proper *synchronisation* of histories, we add *synchronisation barriers*. That is, given two history predicates with histories $H_1$ and $H_2$, and actions $s_1$ and $s_2$ such that $\gamma(s_1, s_2) = \tau$, we allow to extend the histories to $H_1 \cdot s_1$ and $H_2 \cdot s_2$. We call $s_1$ and $s_2$ *synchronisation actions* (for convenience, we usually denote two synchronisation actions with $s$ and $\overline{s}$). It is safe to add such a synchronisation barrier, because we know that all actions in the history so far must happen before this synchronisation. When the threads are joined, all partial histories over the same set of locations $L$ are *merged* together. To allow merging histories, we require that each thread is joined at most once in the program.

In Listing 5.3 the Hist predicate is split when the client forks each thread (lines 14 and 16). Thus both threads can record their changes in parallel in their own partial history. Note that in this example there is no need of adding a synchronisation barrier, because we split the history when it is still empty. The

```
   class Client{
2
   void main(){
4    Counter c = new Counter(0);
   {c.x ↦¹ 0 * fresh(c)}
6  //@ crHist({c.x}, c.x==0);
   {c.x ↦¹_h 0 * Hist({c.x}, 1, c.x==0, ϵ) * fresh(c)}
8  //@ commit c;
   {Hist({c.x}, 1, c.x==0, ϵ) * initialised(c)}
10   Task t1 = new Task(c);
     Task t2 = new Task(c);
12 {Hist({c.x}, 1, c.x==0, ϵ) * initialised(c) * initialised(c) *
              t1.c ↦¹ c * t2.c ↦¹ c}
14   fork t1;
   {Hist({c.x}, 1/2, c.x==0, ϵ) * initialised(c)}
16   fork t2;
   {Hist({c.x}, 1/4, c.x==0, ϵ)}
18   join t1;
   {Hist({c.x}, 3/4, c.x==0, c.inc(1))}
20   join t2;
   {Hist(c.x, 1, c.x==0, c.inc(1) || c.inc(1))}
22 //@ reinit({c.x}, c.x==2);
   {Hist({c.x}, 1, c.x==2, ϵ)}
24 {c.x ↦¹ 2}
    }
26 }
   class Task{
28   Counter c;
   //@ requires c ↦¹ _;
30 //@ ensures c ↦¹ vc;
     Task(Counter vc){
32     this.c = vc;
     }
34 //@ requires Hist(L, π, R, H) * c.x ∈ L * initialised(c);
   //@ ensures Hist(L, π, R, H·c.inc(1));
36   void run(){
       c.incr();
38   }
   }
```

Listing 5.3: The Counter example, the Client class

use of a synchronisation barrier is illustrated later in Example 5.2.

**Recording updates in a history** We extend the specification language with *actions*. Each action is defined by an *action name* and a list of parameters. An action is equipped with an *action specification*: pre- and postcondition; an *accessible* clause which defines the *footprint of the action*, i.e., a set of locations that are allowed to be accessed within the action; and an *assignable* clause, which specifies the locations allowed to be updated:

```
/*@ accessible footprint
  @ assignable modified_locations
  @ requires precondition
  @ ensures postcondition
  @ action actName (parameters);
@*/
```

Listing 5.2 shows a definition of an action inc (lines 5 - 9), which represents an increment of the location x by $k$. Note that the action contract is written in a pure JML language, without the need to explicitly specify permissions, as they are treated separately. In particular, action contracts are used to reason about a trace of a history, which (as discussed above) is a sequential program.

An action may be associated with a program segment that implements the action specification. For this purpose, we introduce a specification command action $a(\overline{v})\{sc\}$, which marks the program block $sc$ as an implementation of the action $a$ with arguments $\overline{v}$. We call $sc$ an *action segment*. In Listing 5.2, we specify an action segment of the action inc in lines 24 - 28.

*Recording Actions* In the prestate of the action segment, a history predicate $\mathsf{Hist}(L, \pi, R, H)$ is required, which captures the behaviour of the footprint locations of the action $a$. i.e., $\forall l \in fp(a).l \in L$. At the end of the action segment, the action is recorded in the history. For this, it is necessary that the action segment implements the specification of the action $a$. For example, in Lst. 5.2 the history $H$ is extended with an action inc(1), line 31.

*Restrictions within an action* As discussed above, an action must be observed by the environmental threads as if it is *atomic*. Thus, it is essential that within the action segment the footprint locations of the action are *stable*, i.e., they cannot be modified by any other thread. Moreover, a modified location should not be visible to other threads until the action is finished. Furthermore, the same thread must not record the same update more than once in the history.

Thus, a thread cannot have started more than one action over the same location simultaneously.

To ensure this, we impose several restrictions on what is allowed in the action segment. In the prestate of the action $a$, we require that the current thread has a positive permission to every footprint location of $a$. Within the action segment we forbid the running thread to release permissions and to make them accessible to other threads. Concretely, within an action segment, we allow only a specific subcategory of commands. This excludes lock-related operations (acquiring, releasing or committing a lock), forking or joining threads, or starting another action.

In this way, we allow two actions to interleave only if they refer to disjoint sets of locations, or if their common locations are only readable by both threads. We also allow a single thread to have at most one started action at a time. It might be possible to lift some of these restrictions later; however, this would probably add extra complexity to the verification approach, while we have not yet encountered an example where these restrictions become problematic.

*Updates within an action* If a history $H$ over $l$ exists, the access permission to $l$ is provided by the $l \overset{\pi}{\mapsto}_h v$ predicate (instead of $l \overset{\pi}{\mapsto} v$). Every update to $l$ must then be part of an action that will be recorded in $H$. Thus, the predicate $l \overset{\pi}{\mapsto}_h v$ provides a "valid" permission only within an action segment with a footprint that refers to $l$. To this end, within the action segment, the history permission predicates $l \overset{\pi}{\mapsto}_h v$ are exchanged for *action permission predicates*, i.e., $l \overset{\pi}{\mapsto}_a v$. Thus, our logic allows a thread to access a shared location when it holds an appropriate fraction of either the permission predicate, $l \overset{\pi}{\mapsto} v$, or the action permission predicate, $l \overset{\pi}{\mapsto}_a v$.

In the example in Listing 5.2, within the action segment, the predicate $\mathsf{x} \overset{1}{\mapsto}_h \mathsf{v}$ is exchanged for $\mathsf{x} \overset{1}{\mapsto}_a \mathsf{v}$ (see lines 24 - 28). The predicate $\mathsf{x} \overset{1}{\mapsto}_a \mathsf{v}$ provides a valid permission to access the location $\mathsf{x}$.

**History reinitialisation** When a thread has the *full* $\mathsf{Hist}(L, 1, R, H)$ predicate, it has complete knowledge of the values of the locations in $L$. The state of these locations is then *stable* and no other thread can update them. The $\mathsf{Hist}$ predicate remembers a predicate $R$ that was true in the previous initial state $\sigma$ of the history, while the history $H$ stores the abstract behaviour of the locations in $L$ after the state $\sigma$. Thus, it is possible to *reinitialise* the $\mathsf{Hist}$ predicate, i.e., reset the history to $H = \epsilon$ and update $R$ to a new predicate $R'$ that holds on the current state. Thus, reasoning about the continuation of the program will

be done with an initial empty history.

The specification command reinit$(L, R')$ converts the Hist$(L, 1, R, H)$ predicate to a new Hist$(L, 1, R', \epsilon)$. Reinitialisation is successful when the new property $R'$ can be proven to hold after the execution of any trace $w$ from the set of traces in $H$, i.e., $\forall w \in \mathsf{Traces}(H).\{R\}w\{R'\}$. As stated above, each trace $w$ from the term $H$ is a sequence of actions specified with a pre- and postcondition and thus, $w$ can be seen as a sequential program.

In Listing 5.3, the history is reinitialised at line 22. The new specified predicate over the location x is: x==2. Notice that at this point, the client does not hold any permission to access x. However, holding the full Hist predicate is enough to reason about the current value of x.

**Destroying a history** It is possible to obtain the $l \overset{1}{\mapsto} v$ predicates back for the locations that are traced in a history. This is done by *destroying the history*, by using the dsthist$(L)$ specification command. The Hist$(L, 1, R, \epsilon)$ predicate and the $l \overset{1}{\mapsto}_h v$ predicates for all $l \in L$ are exchanged for the corresponding $l \overset{1}{\mapsto} v$ predicates. Thereafter, the $l \overset{1}{\mapsto} v$ predicate may be used to access the location $l$, or the client can use this predicate to create a new history over a different set of locations.

## 5.4 Examples

In this section we illustrate the history-based reasoning on two more involved examples: Example 5.2 includes recursive method calls and a shared location that is protected by two different locks, and Example 5.3 shows how one can use histories to reason about functional properties of more complex coarse-grained concurrent data structures.

**Example 5.2.** *(Recursion and multiple locks) Consider a class ComplexCounter (Listing 5.4) with three fields:* data, x *and* y. *It has two locks:* lockx *protects write access to* x *and read access to* data, *while* locky *protects write access to* y *and read access to* data. *If a thread holds both* lockx *and* locky, *it has write access to* data.

*Methods* addX() *and* addY() *(see Listing 5.5) increase respectively* x *and* y *by* data, *while* incr(n) *is a recursive method that increments* data *by* $n$. *The synchronised code in the methods* addX(), addY() *and* incr(n) *is associated with an appropriate action (lines 32, 44, 58). To specify the* incr(n) *method, we additionally specify a recursive process* p, *line 23. The contract of the* incr(n) *method shows that what the current thread does is not an atomic action, but a*

```
    class ComplexCounter {
2
     int data; int x; int y;
4   /*@ pred invx=x ↦¹ₕ _ * data ↦^{1/2}ₕ _;
      @ pred invy=y ↦¹ₕ _ * data ↦^{1/2}ₕ _;
6     @ pred state(int vd, int vx, int vy) =
      @   data ↦¹vd * x ↦¹vx * y ↦¹vy * lockx ↦¹_ * locky ↦¹_;
8   @*/

10  Lock/*@<invx>@*/ lockx;
    Lock/*@<invy>@*/ locky;
12
    //@ requires state;
14  //@ ensures state * fresh(lockx) * fresh(locky);
    ComplexCounter(){
16    lockx = new Lock/*@<invx>@*/();
      locky = new Lock/*@<invy>@*/();
18  }
    . . .
```

Listing 5.4: the ComplexCounter class

*process that can be interleaved with other actions. The contract of the process must correspond to the contracts of the actions it is composed of.*

*Listing 5.6 presents the* ComplexCounterClient *class. The client creates a* ComplexCounter *object* c *and shares it with two other parallel threads,* tx *and* ty*. The client thread updates* c.data *(lines 20, 26), while the threads* tx *and* ty *update the locations* c.x *and* c.y *(lines 18, 24). We want to prove that at the end, after both threads have terminated, the statement* $10 \leq$ c.x+c.y $\leq 40$ *holds.*

*The values of* c.x *and* c.y *at the end depend on the moment when* c.data *has been updated. Thus, the history should trace the updates of all three locations,* c.x*,* c.y *and* c.data*. Each thread then instantiates actions that refer to different sets of locations, but all actions are recorded in the same history. When the threads terminate, the client has the complete knowledge of the program behaviour, in the form of a process algebra term* H=p(10) · s · p(10) ∥ addx() ∥ $\overline{s}$ · addy() *(line 30). By reasoning about* H*, we can prove that the property* R1: $10 \leq$ c.x+c.y $\leq 40$ *holds in the current state. The history predicate is then reinitialised to* Hist(L, 1, R1, $\epsilon$)*.*

Example 5.2 shows that our technique also allows reasoning about more

```
class ComplexCounter{
  . . .
/*@ accessible {x, data};
  @ assignable {x};
  @ ensures x = \old(x) +data;
  @ action addx();

  @ accessible {y, data};
  @ assignable {y};
  @ ensures y = \old(y) +data;
  @ action addy();

  @ accessible {data};
  @ assignable {data};
  @ requires k>0;
  @ ensures data = \old(data) +k;
  @ action inc(int k);

  @ accessible {data};
  @ assignable {data};
  @ ensures data = \old(data)+n;
  @ proc p(int n) =
  @ inc(1) · p(n-1) ◁ n>0 ▷ ϵ;
@*/

/*@ requires Hist(L, π, R, H) *
  @  data, x ∈ L;
  @ ensures Hist(L, π, R, H · addx());
@*/
 void addX(){
    acquire lx;
//@ action addx(){
      x=x+data;
//@ }
    release lx;
 }

/*@ requires Hist(L, π, R, H) *
  @  data, y ∈ L;
  @ ensures Hist(L, π, R, H · addy());
@*/
void addY(){
  acquire ly;
//@ action addy(){
    y=y+data;
//@ }
  release ly;
 }

/*@ requires Hist(L, π, R, H) *
  @  data ∈ L;
  @ ensures Hist(L, π, R, H · p(n));
@*/
void incr(int n){
  if (n>0){
    acquire lx;
    acquire ly;
//@ action inc(1){
      data = data +1;
//@ }
    release lx;
    release ly;
     incr(n−1);
  }
 }
}
```

Listing 5.5: the ComplexCounter class

complicated scenarios in which a location is protected by different locks. By using a technique based on the Owicki-Gries method, providing a concrete resource invariant for every lock that describes certain behaviour would be rather difficult. With our approach, we make a clear separation between permissions and behaviour of locations. Thus, while the lock stores the permissions, the

```
 // L = {c.data, c.x, c.y}
 // R = c.data==0 ∧ c.x==0 ∧ c.y=0
class ComplexCounterClient{
  ThreadX tx;
  ThreadY ty;
void main(){
  ComplexCounter c = new ComplexCounter();
{c.state * fresh(c.lockx) * fresh(c.locky)}
  tx = new ThreadX(c);
  ty = new ThreadY(c);
//@ crHist(L, R);                    //create history
{c.data ↦^1_h 0 * c.x ↦^1_h 0 * c.y ↦^1_h 0 * Hist(L, 1, R, ϵ)}
//@ commit c.lockx;
{c.data ↦^{1/2}_h 0 * c.y ↦^1_h 0 * Hist(L, 1, R, ϵ)}
//@ commit c.locky;
{Hist(L, 1, R, ϵ)}    //split history
{Hist(L, 1/2, R, ϵ) * Hist(L, 1/2, R, ϵ)}
  fork tx;     // tx calls c.addx();
{Hist(L, 1/2, R, ϵ)}
  c.incr(10);
{Hist(L, 1/2, R, p(10))}          //split history
{Hist(L, 1/4, R, p(10)) * Hist(L, 1/4, R, ϵ)}    //sync. barrier
{Hist(L, 1/4, R, p(10) · s) * Hist(L, 1/4, R, ϵ · s̄)}
  fork ty;     // ty calls c.addy();
{Hist(L, 1/4, R, p(10) · s))}
  c.incr(10);
{Hist(L, 1/4, R, p(10) · s · p(10))}
  join tx;
  join ty;                 //merge
{Hist(L, 1, R, p(10) · s · p(10) ‖ addx() ‖ s̄ · addy())}
//@ reinit(L, 10≤c.x+c.y≤40);
{Hist(L, 1, 10≤c.x+c.y≤40, ϵ)}
 }
}
```

Listing 5.6: the ComplexCounterClient class

```
class Set{

  Node first;
  Lock lock;
//@ ghost sset ss;

/*@ pred state(sset ss) =
  @ first ↦¹ u *
  @ u == null ⇒ ss == ∅ *
  @ u ≠ null ⇒ first.state(ss);
  @ pred pinv = ss ↦¹ₕ v * state(v);
@*/

/*@ accessible {ss};
  @ assignable {ss};
  @ ensures ss=\old(ss) ∪ {k};
  @ action a(int k)

  @ accessible {ss};
  @ assignable {ss};
  @ ensures ss=\old(ss) \ {k};
  @ action r(int k)
@*/

//@ requires ss ↦¹ _ * state();
//@ ensures Hist({ss}, 1, ss==∅, ϵ);
  Set(){
    lock= new Lock/*@<pinv>@*/();
//@ crHist({ss}, ss==∅);
{Hist({ss}, 1, ss==∅, ϵ)
 * ss ↦¹ₕ ∅ * state(ss)}
//@ commit lock;
  }

//@ requires Hist({ss}, π, R, H));
//@ ensures Hist({ss},π,R,H · a(data));
  void add(int data){
    acquire lock;
//... add data if not already in the set
  //@ action a(data){
  //@ ss = ss ∪ {data};
  //@ }
   release lock;
  }

//@ requires Hist({ss}, π, R, H));
//@ ensures Hist({ss},π,R,H · r(data));
  void remove(int data){
    acquire lock;
// ... remove data if in the set
  //@ action r(data){
  //@ ss = ss \ {data};
  //@ }
    release lock;
   }
}

class Node {
  int data; Node next;
/*@ pred state(sset ss) =
   @ data ↦¹ v * next↦¹ u *
   @ u == null ⇒ ss=={data} *
   @ u ≠ null ⇒ data ∈ ss *
   @ next.state(ss\{data})
//...
}
```

Listing 5.7: A Set data structure

behaviour is captured independently by the history.

**Example 5.3.** *(Concurrent Set Data Structure) Listing 5.7 and Listing 5.8 present respectively a Set data structure and a client class that uses the structure.*

```
   class Client{
2   Thread1 t1; Thread2 t2; Thread3 t3;
    void main(){
4    Set s = new Set();
   { Hist({s.ss}, 1, s.ss==∅, ϵ) }
6    set.add(2);
   { Hist({s.ss}, 1, s.ss==∅, ss.a(2)) }
8    t1 = new Thread1(s);
     t2 = new Thread2(s);
10   t3 = new Thread3(s);
     fork t1;  //t1 calls s.add(4)
12   fork t2;  //t2 calls s.remove(6)
     fork t3;  //t3 calls s.add(6)
14   join t1;
     join t2;
16   join t3;
   { Hist({s.ss}, 1, s.ss==∅, ss.a(2) · (ss.a(4) ||  ss.r(6) || ss.a(6))) }
18 //@ reinit({s.ss}, {2,4} ⊆ s.ss)
   {Hist({s.ss}, 1, {2,4} ⊆ s.ss, ϵ) }
20  }
   }
```

Listing 5.8: A Set data structure example, the client

*To verify this example, we use additionally* ghost *(specification-only) fields and* specification data types*. However, because these features are orthogonal to the main ideas of our history-based reasoning, for simplicity they are not included in our formal verification system in Section 5.5.*

*The* Set *class (Listing 5.7) represents a set of integers and is implemented as a linked list with unique elements. We associate the* Set *data structure with a representative* ghost *field* ss *(line 5), which has a data type* sset*. Additionally, the resource invariant ensures that the sequential ghost field is always* compatible *with the actual data structure (line 11).*

*Furthermore, we define a history over the ghost field. Therefore, method contracts are expressed in terms of local changes to this history. After threads are joined, we use the history to reason about the structure of the sequential set* ss*, while the resource invariant is used to guarantee that it has the same content as the actual data structure.*

*The* Client *class is represented in Listing 5.8. The client thread creates an empty set and adds the element* 2 *to the set (lines 4 and 6). The set is then*

| | | | |
|---|---|---|---|
| Class definition | $cl$ | ::= | $\mathsf{class}\ C\ \{\overline{fd}\ \overline{md}\ \overline{inv}\ \overline{pd}\ \overline{act}\ \overline{proc}\}$ |
| Commands | $c$ | ::= | $\ldots \mid \mathsf{crhist}(L, R) \mid \mathsf{action}\ v.a(\overline{w})\{ac\}$ |
| | | | $\mid \mathsf{reinit}(L, R) \mid \mathsf{dsthist}(L)$ |
| Location sets | $L$ | $\ni$ | $v.f$ |
| Boolean expressions | $R$ | ::= | $b \mid e == e \mid e > e \mid e < e \ldots$ |
| Action commands | $ac$ | ::= | $T\ x \mid x = v \mid x = op(\overline{v}) \mid T\ r = x \mid$ |
| | | | $\mid \mathsf{while}\ b\ \{ac\} \mid \mathsf{if}\ b\ \mathsf{then}\ ac\ \mathsf{else}\ ac \mid ac; ac \mid$ |
| | | | $x = \mathsf{new}\ C\ \langle\overline{v}\rangle \mid x = v.f \mid v.f = v$ |
| | | | $\mid v.m(\overline{w})$ if $m$ is composed of $ac$ |

Figure 5.1: Language syntax

*shared between three parallel threads: thread* t1 *adds the element* 4 *to the set (if it is not there), thread* t2 *removes the element* 6 *(if it is in the set) and thread* t3 *adds the element* 6 *(if it is not there). At the end when threads are joined, we prove that the elements* 2 *and* 4 *exist in the set.*

## 5.5 Formalisation

We proceed with a formal explanation of the approach. We use the same language and formalisation as presented in Section 4.3, but extended with an appropriate history-based mechanism. This section gives a detailed presentation of the history extension only.

### 5.5.1 Language Syntax

Figure 5.1 shows that the definition of a class may also include a set of specified actions $\overline{act}$ and a set of processes $\overline{proc}$. The set of commands defined in Section 4.3.1 is extended with four new *specification commands* for the management of histories: i) $\mathsf{crhist}(L, R)$, creates a history over a set of locations $L$ with an initial property $R$; ii) $\mathsf{dsthist}(L)$, destroys a history over $L$; iii) $\mathsf{reinit}(L, R)$, reinitialises a history over $L$ with a new initial property $R$; and iv) $\mathsf{action}\ v.a(\overline{w})\{ac\}$, marks the program segment $ac$ as an implementation of the action $v.a(\overline{w})$; Within an action segment, only a restricted set of commands $ac$ are allowed.

| | | | |
|---|---|---|---|
| Actions | $act$ | ::= | $pspec$ action $a(\overline{T}\ \overline{r})$; |
| Processes | $proc$ | ::= | $pspec$ process $p(\overline{T}\ \overline{r}) = H$; |
| Process specifications | $pspec$ | ::= | accessible $L$ assignable $L$ requires $F$ ensures $F$ |
| Histories | $H$ | ::= | $\epsilon \mid s \mid a(\overline{v}) \mid H \triangleleft b \triangleright H \mid$ $\mid H \cdot H \mid H + H \mid H \parallel H \mid p(\overline{r})$ |
| Update actions | $a(\overline{v})$ | $\in$ | UAct |
| Synch. actions | $s, \overline{s}$ | $\in$ | SAct |
| Formulas | $F$ | ::= | $\ldots \mid e.f \overset{\pi}{\mapsto}_h e \mid e.f \overset{\pi}{\mapsto}_a e \mid \mathsf{Hist}(L, \pi, R, H)$ |

Figure 5.2: Specification language

*Specification language* Actions and processes are part of the program specification. As shown in Figure 5.2, actions only have a specification, and no body. Processes have a specification and a body, which must be defined as a proper process expression, a history $H$. Histories are composed of actions: we distinguish between *update actions* $a(\overline{v}) \in \mathsf{UAct}$, for which an action definition exists, and *synchronisation actions* $s \in \mathsf{SAct}$, used to add synchronisation barriers.

The set of formulas is extended with three new predicates: i) $e.f \overset{\pi}{\mapsto}_h e$, i.e., the history permission predicate, denoting that the location $e.f$ is bound to a history; ii) $e.f \overset{\pi}{\mapsto}_a e$, i.e., the action permission predicate, denoting that an action over a location $e.f$ is active; and iii) $\mathsf{Hist}(L, \pi, R, H)$, i.e., the history predicate that represents the history $H$ over set of locations $L$.

### 5.5.2 Language Semantics

*Semantics of histories* The semantics of a history term is defined in terms of its traces. The set of actions that build histories is defined as:

$$\mathcal{A} = \mathsf{UAct} \cup \mathsf{SAct} \cup \{\tau, \delta\},$$

where the communication function $\gamma$ is defined as:

$$\gamma(a, b) = \begin{cases} \tau & \text{if } a, b \in \mathsf{SAct} \text{ define a synchronisation barrier} \\ \delta & \text{otherwise} \end{cases}$$

We use the standard single step semantics $H \xrightarrow{a} H'$ for $H$ moving in one step to $H'$. We extend this to:

$$H \stackrel{a}{\Rightarrow} H' \Leftrightarrow H \xrightarrow{\tau}{}^{*} \xrightarrow{a} \xrightarrow{\tau}{}^{*} H', \text{ for } a \neq \tau$$
$$H \stackrel{\epsilon}{\Rightarrow} H$$
$$H \stackrel{aw}{\Rightarrow} H' \Leftrightarrow H \stackrel{a}{\Rightarrow} \stackrel{w}{\Rightarrow} H'$$

Now the *global completed trace semantics* of a process $H$ is defined as:

$$\mathsf{TS}(H) = \{w \mid \partial_{\mathsf{SAct}}(H) \stackrel{w}{\Rightarrow} \epsilon\}$$

**Example 5.4.** *Assume we have a complete history $H = a \cdot s \cdot b \parallel \overline{s} \cdot c$. To calculate the global trace semantics of $H$, we first need to eliminate the parallel operators ($\parallel$, $\lfloor\!\lfloor$ and $\mid$). After several transformation steps, we get:*

$$\begin{aligned} H = a \cdot s \cdot b \parallel \overline{s} \cdot c = & a \cdot \tau \cdot b \cdot c + a \cdot \tau \cdot c \cdot b + a \cdot \tau \cdot b \mid c + \\ & a \cdot s \cdot ... \ + \ ... + \quad \textit{//all starting with } a \cdot s \\ & a \cdot \overline{s} \cdot ... \ + \ ... \ + \quad \textit{// all starting with } a \cdot \overline{s} \\ & \overline{s} \cdot ... \ + \ ... \ + \quad \textit{// all starting with } \overline{s} \\ & a \mid \overline{s} \cdot ... \ + \ ... \ + \quad \textit{// all starting with } a \mid \overline{s} \end{aligned}$$

*where $\tau$ is obtained as a result of $s \mid \overline{s}$. Next, all synchronisation actions that still occur in $H$ are replaced with $\delta$:*

$$\begin{aligned} \partial_{\mathsf{SAct}}(H) = & a \cdot \tau \cdot b \cdot c + a \cdot \tau \cdot c \cdot b + a \cdot \tau \cdot \delta + \\ & a \cdot \delta \cdot ... \ + \ ... + a \cdot \delta \cdot ... \ + \ ... \ + \delta \cdot ... \ + \ ... + \delta \cdot ... \end{aligned}$$

*Therefore, we have that the set $\mathsf{TS}(H)$ contains two possible traces of actions:*

$$\mathsf{TS}(H) = \{a \cdot b \cdot c + a \cdot c \cdot b\}$$

*Program state* To model histories, we extend the program state with two history-related components, a *history map* HistMap and a *sequence of active actions* ActiveActions:

$$\begin{aligned} \sigma \in \mathsf{State} \quad = \quad & \mathsf{Heap} \times \mathsf{ThreadPool} \times \mathsf{LockTable} \times \mathsf{InvTable} \times \\ & \mathsf{HistMap} \times \mathsf{ActiveActions} \end{aligned}$$

Both HistMap and ActiveActions are ghost components, used to describe the behaviour of the history-related specification commands only. Concretely:

i) $hm \in \mathsf{HistMap}$ stores information about the histories. For every history over $L$ that exists in the program state, $hm$ maps a sequence of the locations in $L$ to a sequence of their corresponding initial values and to a sequence of actions recorded in the history. Each action is represented by an identifier of the object receiver, an identifier of the action ($a \in \mathsf{ActId}$) and a list of action parameters:

$$\begin{aligned} hm &\in \mathsf{HistMap} = \overline{\mathsf{Loc}} \rightharpoonup (\overline{\mathsf{Value}} \times \overline{\mathsf{Action}}) \\ act &\in \mathsf{Action} = \mathsf{ObjId} \times \mathsf{ActId} \times \overline{\mathsf{Value}} \\ l &\in \mathsf{Loc} = \mathsf{ObjId} \times \mathsf{FieldId} \end{aligned}$$

For example, in a state in which a single action $o.inc(1)$ has been added to a history over locations $\{o.data, o.x, o.y\}$, where the initial values of all three locations were 0 (see Example 5.2 from Section 5.3), the state of the history map is:

$$hm = [o.data, o.x, o.y] \mapsto ([0,0,0], [(o.inc, [1])])$$

Importantly, two histories always refer to *disjoint* sets of locations: $\forall L_1, L_2 \in dom(hm).\ L_1 \cap L_2 = \emptyset$. This is ensured by the logic: when a history over $l$ is created, the *full* permission predicate for $l$ is consumed and returned back only once the history is destroyed.

ii) $sa \in \mathsf{ActiveActions}$ represents a sequence of actions that are currently active and are not yet recorded in the history. For every active action we store the locations that the action refers to.

$$\begin{aligned} sa &\in \mathsf{ActiveActions} = \overline{\mathsf{ActiveAction}} \\ aa &\in \mathsf{ActiveAction} = \mathsf{Set}(\mathsf{Loc}) \end{aligned}$$

**Operational semantics** We present the operational semantics of the commands in our language in the form of a transition $\sigma \rightsquigarrow \sigma'$, where the state $\sigma$ now contains two additional components, $hm$ and $sa$, $\sigma = (h, tp, lt, it, hm, sa)$. The semantics of all commands from Section 4.3.1 remains the same: none of these commands change the state of $hm$ or $sa$. Here we illustrate only the semantics of the new history-related specification commands.

*Histories management* The commands $\mathsf{crhist}(L, R)$, $\mathsf{dsthist}(L)$, $\mathsf{reinit}(L, R)$ and $\mathsf{action}\ o.a(\overline{v})\{ac\}$ are used for management of histories. As the other specification-only commands, they cannot be aborted. Further, they all operate on the last two components only, $hm$ and $sa$, which ensures that the specification does not change the behaviour of the program. To model the command $\mathsf{action}\ o.a(\overline{v})\{ac\}$, we add an auxiliary command $\mathsf{endaction}\ o.a(\overline{v})$, which is executed at the end of the action segment. The formal definition is given below:

$$
\begin{array}{ll}
[\mathit{Create}] & (h, tp.(t, s, \mathsf{crhist}(\{l_1, ..., l_n\}, R); c), lt, it, hm, sa) \rightsquigarrow \\
& \quad (h, tp.(t, s, c), lt, it, hm[[l_1, ..., l_n] \mapsto ([h(l_1), ..., h(l_n)], \mathsf{nil})], sa) \\
[\mathit{Destroy}] & (h, tp.(t, s, \mathsf{dsthist}(\{l_1, ..., l_n\}); c), lt, it, hm, sa) \rightsquigarrow \\
& \quad (h, tp.(t, s, c), lt, it, hm[[l_1, ..., l_n] \mapsto \bot], sa) \\
[\mathit{Reinit}] & (h, tp.(t, s, \mathsf{crhist}(\{l_1, ..., l_n\}, R); c), lt, it, hm, sa) \rightsquigarrow \\
& \quad (h, tp.(t, s, c), lt, it, hm[[l_1, ..., l_n] \mapsto ([h(l_1), ..., h(l_n)], \mathsf{nil})], sa) \\
[\mathit{Action}] & (h, tp.(t, s, \mathsf{action}\ o.a(\overline{v})\{ac\}; c), lt, it, hm, sa) \rightsquigarrow \\
& \quad (h, tp.(t, s, ac; \mathsf{endaction}\ o.a(\overline{v}); c), lt, it, hm, sa{+\!\!+}fp(h(o)^1, a, o)) \\
[\mathit{End\ action}] & (h, tp.(t, s, \mathsf{endaction}\ o.a(\overline{v}); c), lt, it, hm, sa) \rightsquigarrow \\
& \quad (h, tp.(t, s, c), lt, it, hm', sa — fp(h(o)^1, a, o)) \\
& \quad hm' = hm[[l_1, ...l_n] \mapsto (hm([l_1, ...l_n])^1, hm([l_1, ...l_n])^2 {+\!\!+} (o.a, \overline{v}))] \\
& \quad \text{where } [l_1, ...l_n] \in dom(hm), fp(h(o)^1, a, o) \subseteq \{l_1, ..., l_n\}
\end{array}
$$

- [*Create*] : When a history over a set of locations $L = \{l_1, ..., l_n\}$ is created, the history map $hm$ is extended with a new item representing the new history: an array of locations $[l_1, ..., l_n]$ mapping to their corresponding current values (which are copied from the heap) and an empty sequence of actions recorded to this history.
- [*Destroy*] : When a history over a set of locations $L$ is destroyed, the representative mapping is removed from $hm$.
- [*Reinit*] : When reinitialising a history over $L$, the appropriate history sequence in $hm$ is emptied, and the values of the locations in $L$ stored in $hm$ are updated to the current value of these locations.
- [*Action*] : Starting a new action adds the set of locations referred by this action to the sequence $sa$ of active actions. The function $fp(C, a, o)$ returns the set of assignable locations of the action $o.a$, where $C$ is the class of the object $o$ (see Appendix C).
- [*End action*] : At the end of an action, the set of its footprint locations is removed from $sa$, and the action is recorded in the related history in $hm$.

*Initial program state* Having the two additional components of the program state, the initial program state is now extended, such that both the history map and the sequence of active actions are empty:

$$\sigma_0 = (\emptyset, (o_{main}, \emptyset, c_{main}), \emptyset, \emptyset, \emptyset, \mathsf{nil})$$

**Resources** As already explained in Section 4.3.2, we describe the meaning of the formulas in our language over a resource $\mathcal{R}$, i.e., an abstraction of the global program state. Therefore, we adapt the definition of $\mathcal{R}$, to give semantics to the new formulas: $e.f \overset{\pi}{\mapsto}_h v$, $e.f \overset{\pi}{\mapsto}_a v$ and $\mathsf{Hist}(L, \pi, R, H)$, as well as to the existing predicate $e.f \overset{\pi}{\mapsto} v$, whose semantics needs to be changed. To this end, we extend the definition of a resource with two new components: *abstract history map* $\mathcal{M}$ and *abstract sequence of active actions* $\mathcal{A}$:

$$\mathcal{R} = (\mathcal{H}, \mathcal{L}, \mathcal{T}, \mathcal{J}, \mathcal{M}, \mathcal{A})$$

For a resource $\mathcal{R}$ that abstracts the global state $\sigma$, the relation $\mathit{abstracts}(\sigma, \mathcal{R})$ holds (see Appendix C for its formal definition).

The definition of the *compatibility* and the *joining* operator is also accordingly extended:

$$\begin{array}{lcl} \mathcal{R}\#\mathcal{R}' & \Leftrightarrow & \mathcal{H}\#\mathcal{H}' \wedge \mathcal{L}\#\mathcal{L}' \wedge \mathcal{T}\#\mathcal{T}' \wedge \mathcal{J}\#\mathcal{J}' \wedge \mathcal{M}\#\mathcal{M}' \wedge \mathcal{A}\#\mathcal{A}' \\ \mathcal{R}*\mathcal{R}' & = & \begin{cases} (\mathcal{H}*\mathcal{H}', \mathcal{L}*\mathcal{L}', \mathcal{T}*\mathcal{T}', \mathcal{J}*\mathcal{J}', \mathcal{M}*\mathcal{M}', \mathcal{A}*\mathcal{A}') & \text{if } \mathcal{R}\#\mathcal{R}' \\ \bot & \text{otherwise} \end{cases} \end{array}$$

Below we give an explanation of the two new components.

*Abstract history map* The component $\mathcal{M}$ abstracts the global history map $hm$, mapping the footprint locations of every existing history to the initial values of these locations, and the sequence of actions recorded to the history. To describe the meaning of the $\mathsf{Hist}(L, \pi, R, H)$ predicate, this is not sufficient, because the predicate represents only a fraction $\pi$ of the history, and moreover, the process algebra $H$ of the predicate is a local history; thus, it is incomplete and it contains only some of the actions recorded in the global history. To this end, we extend the definition of $\mathcal{M}$, mapping every history to a fraction $[0, 1]$, and associating every recorded action with a boolean flag to indicate whether this action is owned by the resource. Formally it is represented as:

$$\mathcal{M} : \overline{\mathsf{Loc}} \rightharpoonup \overline{\mathsf{Value}} \times [0, 1] \times \overline{\mathsf{Action} \times \mathsf{bool}}$$

The definition of the compatibility and joining operator is verbose but intuitive: two history maps are compatible when for every history they store the same initial values, the sum of the fractions in both resources does not exceed 1 and the same action does not appear in both resources. When two resources join, the resulting resource contains the sum of the fractions and all actions from both resources. Furthermore, for an abstract history map $\mathcal{M}$ it is important that: if $\mathcal{M}$ stores no fraction for the history over $L$, then the history map contains no action over $L$ (all associated actions, if any, are marked with the flag $\mathsf{false}$). We refer to this condition as *soundness of an abstract history map* and denote it with $\mathcal{M} : \diamond$. This condition ensures that the resource that contains a full fraction of the history predicate contains the complete history stored in the global history map. Fomally:

$$
\begin{array}{lcl}
\mathcal{M} : \diamond & \Leftrightarrow & \forall \bar{l} \in \mathit{dom}(\mathcal{M}).\mathcal{M}(\bar{l})^2 == 0 \Rightarrow \nexists i.\mathcal{M}(\bar{l})^3\,[i] = (\_, \mathsf{true}) \\
\mathcal{M}_1 \# \mathcal{M}_2 & \Leftrightarrow & \mathcal{M}_1 : \diamond \wedge \mathcal{M}_2 : \diamond \wedge \\
 & & \mathit{dom}(\mathcal{M}_1) = \mathit{dom}(\mathcal{M}_2) \wedge \forall \bar{l} \in \mathit{dom}(\mathcal{M}_1).\mathcal{M}_1(\bar{l})^1 = \mathcal{M}_2(\bar{l})^1 \wedge \\
 & & \wedge \mathcal{M}_1(\bar{l})^2 + \mathcal{M}_2(\bar{l})^2 \leq 1 \wedge \mathcal{M}_1(\bar{l})^3 \# \mathcal{M}_2(\bar{l})^3 \\
S_1 \# S_2 & = & |S_1| = |S_2| \wedge \forall i.S_1\,[i]^1 = S_2\,[i]^1 \wedge \neg(S_1\,[i]^2 \wedge S_2\,[i]^2) \\
\mathcal{M}_1 * \mathcal{M}_2 & = & \begin{cases} \lambda \bar{l}.(\mathcal{M}_1(\bar{l})^1, \mathcal{M}_1(\bar{l})^2 + \mathcal{M}_2(\bar{l})^2, \mathcal{M}_1(\bar{l})^3 * \mathcal{M}_2(\bar{l})^3) & \text{if } \mathcal{M}_1 \# \mathcal{M}_2 \\ \bot & \text{otherwise} \end{cases} \\
S_1 * S_2 & = & \lambda i.(S_1[i]^1, S_1[i]^2 \vee S_2[i]^2) \\
 & & \text{where } S : \overline{\mathsf{Action} \times \mathsf{bool}}
\end{array}
$$

**Example 5.5.** *To give a better understanding, we give an example of three abstract history maps $\mathcal{M}_1$, $\mathcal{M}_2$ and $\mathcal{M}_2$:*

$$
\begin{array}{lcl}
\mathcal{M}_1 & = & [o.f, o.f'] \mapsto ([0,2], 1/4, [(act_1, \mathsf{false}), (act_2, \mathsf{true}), (act_1, \mathsf{false})]) \\
\mathcal{M}_2 & = & [o.f, o.f'] \mapsto ([0,2], 1/4, [(act_1, \mathsf{true}), (act_2, \mathsf{false}), (act_1, \mathsf{false})]) \\
\mathcal{M}_3 & = & [o.f, o.f'] \mapsto ([0,2], 1/2, [(act_1, \mathsf{true}), (act_2, \mathsf{true}), (act_1, \mathsf{true})])
\end{array}
$$

*The first two abstract maps are compatible $\mathcal{M}_1 \# \mathcal{M}_2$, but $\mathcal{M}_3$ is not compatible with any of them, because it contains actions that already exist in $\mathcal{M}_1$ and $\mathcal{M}_2$. We can join $\mathcal{M}_1$ and $\mathcal{M}_2$ (because they are compatible) and obtain the result:*

$$
\mathcal{M}_1 * \mathcal{M}_2 = [o.f, o.f'] \mapsto ([0,2],\ 1/2, [(act_1, \mathsf{true}), (act_2, \mathsf{true}), (act_1, \mathsf{false})])
$$

$$\mathcal{R} = (\mathcal{H}, \mathcal{L}, \mathcal{T}, \mathcal{J}, \mathcal{M}, \mathcal{A})$$

$$\begin{array}{rl}
[Perm] & \mathcal{R}; s \models e.f \overset{\pi}{\mapsto} e' \Leftrightarrow [\![e]\!]^{\mathcal{H}}_s = o \wedge [\![e']\!]^{\mathcal{H}}_s = o' \wedge \\
& \mathcal{H}(o.f) = (o', \pi') \wedge \pi' \geq \pi \wedge \forall \bar{l} \in dom(\mathcal{M}).o.f \notin \bar{l} \\
[HistPerm] & \mathcal{R}; s \models e.f \overset{\pi}{\mapsto}_h e' \Leftrightarrow [\![e]\!]^{\mathcal{H}}_s = o \wedge [\![e']\!]^{\mathcal{H}}_s = o' \wedge \\
& \mathcal{H}(o.f) = (o', \pi') \wedge \pi' \geq \pi \wedge \exists \bar{l} \in dom(\mathcal{M}).o.f \in \bar{l} \\
[ActPerm] & \mathcal{R}; s \models e.f \overset{\pi}{\mapsto}_a e' \Leftrightarrow [\![e]\!]^{\mathcal{H}}_s = o \wedge [\![e']\!]^{\mathcal{H}}_s = o' \wedge \\
& \mathcal{H}(o.f) = (o', \pi') \wedge \pi' \geq \pi \wedge \exists aa \in \mathcal{A}.o.f \in aa^1 \wedge aa^2 = \mathsf{true} \\
[History] & \mathcal{R}; s \models \mathsf{Hist}(\{e_1.f_1, ..., e_n.f_n\}, \pi, R, H) \Leftrightarrow \\
& i) \quad \forall i = 1..n. [\![e_i]\!]^{\mathcal{H}}_s = o_i \wedge \bar{l} = [o_1.f_1, ..., o_n.f_n] \in dom(\mathcal{M}) \wedge \\
& ii) \quad R[\mathcal{M}(\bar{l})^1\,[i]\,/o_i.f_i]_{\forall i=1..n} = \mathsf{true} \wedge \\
& iii) \quad \mathcal{M}(\bar{l})^2 \geq \pi \wedge \\
& iv) \quad \mathsf{filter}(\mathcal{M}(\bar{l})^3) \in CT_G(H)
\end{array}$$

Figure 5.3: Semantics of formulas

*Abstract sequence of active actions* The second component $\mathcal{A}$ is an abstraction of the global sequence of active actions. We need this component to express the semantics of the formula $e.f \overset{\pi}{\mapsto}_a v$, which informally means that the local thread has started an action over the location $e.f$. Therefore, we should be able to assign each active action to a particular thread. To this end, the $\mathcal{A}$ component assigns also a boolean flag to every active action to indicate whether this active action is owned by the resource.

$$\mathcal{A} : \overline{\mathsf{Set}(\mathsf{Loc}) \times \mathsf{bool}}$$

The compatibility relation holds when the same active action does not appear in both resources, while the joining operator combines the actions from both resources:

$$\begin{array}{rcl}
\mathcal{A}_1 \# \mathcal{A}_2 & \Leftrightarrow & |\mathcal{A}_1| = |\mathcal{A}_2| \wedge \forall i. \mathcal{A}_1\,[i]^1 = \mathcal{A}_2\,[i]^1 \wedge \neg(\mathcal{A}_1\,[i]^2 \wedge \mathcal{A}_2\,[i]^2) \\
\mathcal{A}_1 * \mathcal{A}_2 & = & \begin{cases} \lambda i.(\mathcal{A}_1[i]^1, \mathcal{A}_1[i]^2 \vee \mathcal{A}_2[i]^2) & \text{if } \mathcal{A}_1 \# \mathcal{A}_2 \\ \bot & \text{otherwise .} \end{cases}
\end{array}$$

**Semantics of formulas** Figure 5.3 presents the semantics of the history-related predicates.

- [*Perm*] : The predicate $e.f \overset{\pi}{\mapsto} e'$ describes that the value of $e.f$ is $e'$, the abstract heap $\mathcal{H}$ contains at least $\pi$ permission for $e.f$, and the location $e.f$ does not occur in any history in the abstract history map $\mathcal{M}$.
- [*HistPerm*] : The predicate $e.f \overset{\pi}{\mapsto}_h e'$ has the same meaning as $e.f \overset{\pi}{\mapsto} e'$, except that the location $e.f$ occurs in a history in $\mathcal{M}$.
- [*ActPerm*] : The predicate $e.f \overset{\pi}{\mapsto}_a e'$ describes that $\mathcal{H}$ stores the value $e'$ and at least permission $\pi$ for $e.f$, while the abstract sequence of active actions contains an action that refers to $e.f$.
- [*History*] : The predicate $\mathsf{Hist}(L, \pi, R, H)$ is valid when i) $\mathcal{M}$ contains a mapping for the locations in $L$; ii) the formula $R$ holds when these locations are substituted with the related initial values stored in this mapping; iii) the fraction in the mapping is at least $\pi$; and iv) if $S$ is the sequence of actions in the mapping, then $\mathsf{filter}(S)$ belongs to $\mathsf{TS}(H)$. The function $\mathsf{filter}(S)$ (see Appendix C) returns the subsequence of the sequence $S$, ordered in the same order, containing only those actions from $S$ that are marked with the flag $\mathsf{true}$. For example, if $S = [(act_1, \mathsf{true}), (act_2, \mathsf{false}), (act_3, \mathsf{true})]$, then $\mathsf{filter}(S) = [act_1, act_3]$.

### 5.5.3 Proof System

We extend the proof system from Section 4.3.3 with a set of axioms added to the proof theory, and a set of Hoare triples.

*Proof theory* The first two axioms shown in Figure 5.4 describe that the history permission predicate $o.f \overset{\pi}{\mapsto}_h v$ and the history predicate $\mathsf{Hist}(L, \pi, R, H)$ can be split into fractions and fractions can be merged. The last axiom is used to add a synchronisation barrier. Note that a rule for splitting and merging an action permission is not needed, because this predicate exists only within an action where distribution of resources among threads is not allowed.

*Hoare triples* The rest of the rules in our proof system are inference rules describing the behaviour of a specific command. These are listed in Figure 5.5:

- [*ReadA*] and [*WriteA*]: These rules are used to access a location when an action over this location is in progress. Note that the standard rules [*Read*] and [*Write*] (Figure 4.6, Section 4.3.3) are also valid to allow accessing a location when there is no history maintained for this location.

$$[\mathit{SplitMerge\ HPerm}] \quad \vdash o.f \overset{\pi_1+\pi_2}{\mapsto}_h v \mathrel{*\!\!-\!\!*} o.f \overset{\pi_1}{\mapsto}_h v * o.f \overset{\pi_2}{\mapsto}_h v$$

$$[\mathit{SplitMerge\ Hist}] \quad \vdash \mathsf{Hist}(L, \pi_1 + \pi_2, R, H_1 \parallel H_2) \mathrel{*\!\!-\!\!*} \mathsf{Hist}(L, \pi_1, R, H_1) * \mathsf{Hist}(L, \pi_2, R, H_2)$$

$$[\mathit{Sync\ Barrier}] \quad \vdash \frac{\gamma(s, \overline{s}) = \tau}{\mathsf{Hist}(L, \pi_1, R, H_1) * \mathsf{Hist}(L, \pi_2, R, H_2) \mathrel{-\!\!*} \mathsf{Hist}(L, \pi_1, R, H_1 \cdot s) * \mathsf{Hist}(L, \pi_2, R, H_2 \cdot \overline{s})}$$

Figure 5.4: History-related axioms

$$[\mathit{ReadA}] \quad \frac{}{\vdash \{o.f \overset{\pi}{\mapsto}_a w\}\ x = o.f\ \{o.f \overset{\pi}{\mapsto}_a w * x == w\}}$$

$$[\mathit{WriteA}] \quad \frac{o : T \quad S = \{I | I \in \mathit{inv}(T^2), o.f \in \mathit{fp}(o.I)\} \quad F = \circledast_{I \in S} \mathsf{break}(o.I, \pi)}{\vdash \{o.f \overset{1}{\mapsto}_a \_ * F\}\ o.f = w\ \{o.f \overset{1}{\mapsto}_a w * F\}}$$

$$[\mathit{Create}] \quad \frac{}{\vdash \{\circledast_{o.f \in L} o.f \overset{1}{\mapsto} w * R\}\ \mathsf{crhist}(L, R)\ \{\circledast_{o.f \in L} o.f \overset{1}{\mapsto}_h w * \mathsf{Hist}(L, 1, R, \epsilon)\}}$$

$$[\mathit{Destroy}] \quad \frac{}{\vdash \{\circledast_{o.f \in L} o.f \overset{1}{\mapsto}_h w * \mathsf{Hist}(L, 1, R, \epsilon)\}\mathsf{dsthist}(L)\{\circledast_{o.f \in L} o.f \overset{1}{\mapsto} w\}}$$

$$[\mathit{Action}] \quad \frac{\begin{array}{c} o : T_1 \quad \overline{w} : \overline{T} \quad \mathsf{abody}(T_1^2, a, \overline{T}) = \mathsf{requires}\ F\ \mathsf{ensures}\ F' \mathsf{accessible}\ L_a\ a(\overline{r}); \\ L_a \in L;\ \sigma = [\overline{w}/\overline{r}, o/\mathsf{this}] \\ \vdash \{\circledast_{l \in L_a} l \overset{\pi_l}{\mapsto}_a u * F[\sigma]\}\ ac\ \{\circledast_{l \in L_a} l \overset{\pi_l}{\mapsto}_a v * F'[\sigma]\} \end{array}}{\begin{array}{c} \{\circledast_{l \in L_a} l \overset{\pi_l}{\mapsto}_h u * \mathsf{Hist}(L, \pi, R, H) * F[\sigma]\} \\ \vdash \quad \mathsf{action}\ o.a(\overline{w})\{ac\}; \\ \{\circledast_{l \in L_a} l \overset{\pi_l}{\mapsto}_h v * \mathsf{Hist}(L, \pi, R, H \cdot o.a(\overline{w})) * F'[\sigma]\} \end{array}}$$

$$[\mathit{Reinit}] \quad \frac{\forall w \in \mathsf{TS}(H).\ \vdash \{R\} w \{R'\}}{\vdash \{\mathsf{Hist}(L, 1, R, H)\}\ \mathsf{reinit}(L, R')\ \{\mathsf{Hist}(L, 1, R', \epsilon)\}}$$

Figure 5.5: History-related Hoare triples

- [*Create*] and [*Destroy*]: Creating a history over $L$ converts all write permissions for the locations in $L$ to history permission predicates and produces a history predicate with an empty history. Destroying a history is analogous.
- [*Action*]: The history permissions are converted to action permissions within the action; if the action implementation satisfies the action's contract, the action will be recorded in the history. The function $\mathsf{abody}(C, a, \overline{V})$ returns the body of the action $a(\overline{V}\ \overline{x})$ in class $C$ (see Appendix C).
- [*Reinit*]: The premise in this rule requires that the Hoare triple $\{R\}w\{R'\}$ holds for every trace $w \in \mathsf{TS}(H)$. Importantly, $w$ is a trace of actions, where every action can also be considered as a call to an abstract method (an action contains a specification and no implementation); thus, the trace $w$ is also a sequential program statement.

### 5.5.4 Soundness

The proof system in Section 4.3 guarantees *partial correctness*, *data race-freedom* and *high-level data race-freedom*. To ensure that our new logic equipped with the history-based mechanism still guarantees these properties, we need to extend the proof of Invariant 4.1, which states that every reachable program state is valid. Concretely, we need to prove soundness of the newly added axioms and inference rules. Note that the rules [*State*], [*LockTable*], [*ThreadPool*] and [*Thread*] from Section 4.4.1 remain the same, except that the global and the abstract states, $\sigma$ and $\mathcal{R}$ repectively, are extended appropriately with the new history related components. Below we show the validity of the [*Reinit*] rule, while soundness for the other rules follows trivially from the semantics of the language.

*Proof of the* [*Reinit*] *rule.*

$$[\mathit{Reinit}]\quad \frac{\forall w \in \mathsf{TS}(H).\ \vdash \{R\}w\{R'\}}{\vdash \{\mathsf{Hist}(L, 1, R, H)\}\ \mathsf{reinit}(L, R')\ \{\mathsf{Hist}(L, 1, R', \epsilon)\}}$$

*Proof.* Let $\sigma$ and $\sigma'$ be the pre- and poststate of the $\mathsf{reinit}(L, R')$ command, respectively:

$$\sigma = (h, tp.(t, s, \mathsf{reinit}(L, R'); c), lt, it, hm, sa) \rightsquigarrow (h, tp.(t, s, c), lt, it, hm', sa) = \sigma'$$

Let $\mathcal{R}_\sigma$ be a resource that abstracts $\sigma$. From $\sigma : \diamond$ we have:

$$\mathcal{R}_\sigma = \mathcal{R}_t * \mathcal{R}_{lt} * \mathcal{R}_{tp} \tag{5.1}$$
$$\mathcal{R}_t, s \models \mathsf{Hist}(L, 1, R, H) \tag{5.2}$$
$$\mathcal{R}_{lt} \vdash lt : \diamond \tag{5.3}$$
$$\mathcal{R}_{tp} \vdash tp : \diamond \tag{5.4}$$

We need to prove that there exists a resource $\mathcal{R}_{\sigma'}$ that abstracts $\sigma'$ such that:

$$\mathcal{R}_{\sigma'} = \mathcal{R}'_t * \mathcal{R}'_{lt} * \mathcal{R}'_{tp} \tag{5.5}$$
$$\mathcal{R}'_t, s \models \mathsf{Hist}(L, 1, R', \epsilon) \tag{5.6}$$
$$\mathcal{R}'_{lt} \vdash lt : \diamond \tag{5.7}$$
$$\mathcal{R}'_{tp} \vdash tp : \diamond \tag{5.8}$$

From (5.2) we have that the global history map $hm$ in $\sigma$ contains a history over $L = \{l_1, ..., l_n\}$:

$$\exists [l_1, ..., l_n] \in dom(hm) \qquad hm([l_1, ..., l_n]) = ([v_1, ...v_n], [act_1, ..., act_m])$$

Let $\sigma_{init}$ be the last initial state of the history, i.e., the prestate of the last $\mathsf{reinit}(L, R)$ command, if any, or the prestate of the $\mathsf{crhist}(L, R)$ command, otherwise. The operational semantics of our language ensures that such a command exists. Further, let $\sigma_1$ be the state before the first action $act_1$ has started, and $\sigma_m$ be the state after the last action $act_m$ has been recorded to the history. Thus, $\sigma_{init} \leq \sigma_1 \leq \sigma_m \leq \sigma < \sigma'$, where "$\leq$" denotes "precedes or is equal to". If no action has been recorded to the history ($hm([l_1, ...l_n]) = ([v_1, .., v_n], \mathsf{nil})$) we have $\sigma_{init} = \sigma_1 = \sigma_m = \sigma < \sigma'$.

From (5.2) we know that $R$ holds over the initial values of locations stored in $hm$, i.e., $R[v_i/l_i]_{\forall i=1..n} = \mathsf{true}$. This implies that $R$ holds on the heap $h_{init}$ in the state $\sigma_{init}$, when the values from $h_{init}$ have been copied to the history map, $R[h_{init}(l_i)/l_i]_{\forall i=1..n} = \mathsf{true}$. Furthermore, no update of $l_i$ might have happened between $\sigma_{init}$ and $\sigma_1$. This is ensured by the logic: any update must be preceded by starting an action. Therefore, the values of all locations in $l_i \in L$ in $\sigma_{init}$ and $\sigma_1$ are equal. We denote this $\sigma_{init} =_L \sigma_1$. Thus, $R$ also holds on the heap $h_1$ in the state $\sigma_1$:

$$R[h_1(l_i)/l_i]_{\forall i=1..n} = \mathsf{true} \tag{5.9}$$

Furthermore, because from (5.2) the complete predicate $\mathsf{Hist}(L, 1, R, H)$ holds over $\mathcal{R}_t$, from the definition of compatibility of abstract history maps, we have that the abstract history map $\mathcal{M}_t$ in $\mathcal{R}_t$ is equivalent to the abstract history map $\mathcal{M}_\sigma$ in the complete resource $\mathcal{R}_\sigma = \mathcal{R}_t * \mathcal{R}_{lt} * \mathcal{R}_{tp}$. Because $\mathcal{R}_\sigma$ is an abstraction of the global history map $hm$ in $\sigma$, we have that $\mathcal{M}_\sigma$ and $\mathcal{M}_t$ contain the complete sequence of actions from the global history, $\mathcal{M}_t(L) = ([v_1, ..., v_n], 1, [(act_1, \mathsf{true}), ..., (act_m, \mathsf{true}))$. Therefore, $(act_1, ..., act_m) \in \mathsf{TS}(H)$.

The premise of the [*Reinit*] rule states that $\{R\}w\{R'\}$ holds for every $w \in \mathsf{TS}(H)$. From $(act_1, ..., act_m) \in \mathsf{TS}(H)$, we have $\{R\}(act_1, ..., act_m)\{R'\}$. This means that if $R$ holds on the heap in the state $\sigma_1$ (which is already shown by 5.9), we can conclude that $R'$ holds in the state $\sigma_m$ (the syntax restrictions within an action segment guarantee that the program execution results in a state equivalent to the result state of an execution in which the actions happen serially, without overlapping). Moreover, $\sigma_1 =_L \sigma$ because no update of $l \in L$ might have happened between $\sigma_1$ and $\sigma$. Therefore, $R'$ holds on the heap in $\sigma$:

$$R[h(l_i)/l_i]_{\forall i=1..n} = \mathsf{true} \tag{5.10}$$

From the operational semantics of the command $\mathsf{reinit}(L, R')$, for the global history map in state $\sigma'$ we have $hm'([l_1, .., l_n]) = ([h(l_1), .., h(l_n)], \mathsf{nil})$. The new global state $\sigma'$ may be abstracted by a resource $\mathcal{R}' = \mathcal{R}'_t * \mathcal{R}'_{lt} * \mathcal{R}'_{tp}$ such that: $\mathcal{R}'_{lt} = \mathcal{R}_{lt}$ and $\mathcal{R}'_{tp} = \mathcal{R}_{tp}$, from where (5.7 and 5.8) are satisfied; and $\mathcal{R}'_t = (\mathcal{H}, \mathcal{L}, \mathcal{T}, \mathcal{J}, \mathcal{M}', \mathcal{A})$, where $\mathcal{R}_t = (\mathcal{H}, \mathcal{L}, \mathcal{T}, \mathcal{J}, \mathcal{M}, \mathcal{A})$, and $\mathcal{M}' = \mathcal{M}[[l_1, .., l_n] \mapsto ([h(l_1), ..., h(l_n)], 1, \mathsf{nil})]$. Because from (5.10) we have that $R'$ holds over the values $h(l_1), ..., h(l_n)$, it is trivial to prove that the predicate $\mathsf{Hist}(L, 1, R', \epsilon)$ holds over the resource $\mathcal{R}'_t$, which proves (5.6). This concludes the proof. □

## 5.6 Tool Support

Our history-based technique has been integrated in our program verifier, the VerCors tool set [BH14]. The tool performs verification of multithreaded programs written in languages such as Java and C (it was also extended to support verification of OpenCL kernel programs [BHM14, BDH15]). Programs are annotated with separation logic-based specifications; the tool then transforms the program into a simplified language understandable for one of the back-end verifiers, Chalice [LMS09] and Silver [JKM+14]; the back-end tool then performs the verification.

To verify programs specified with histories, there are two verification tasks to be performed. In top down order, it needs to be checked:

i) if the proof rule [*Reinit*] (see Section 5.5.3) is applied correctly, i.e., for every $w \in \mathsf{TS}(H)$, the Hoare triple $\{R\}w\{R'\}$ logically follows from the contracts on the actions; and

ii) if the local histories are properly maintained in the program.

Below we give a short explanation of both verification tasks.

**Verification of the** [*Reinit*] **rule** To verify the functional behaviour of histories, the tool requires that every action and every process specified in the program is equipped with a contract. Each action is then translated to an abstract method (without implementation) with a corresponding specification. For processes there are two steps to be done: *process transformation* and *method generation.*

*Process transformation* Every process is first transformed to a *guarded sequential* process (which contains no merge ($\|$) operator). This rewriting is done by applying techniques known from *linearisation* of processes (see e.g. [GPU01]). First, the definition is expanded by applying the axioms of process algebra and unfolding defined processes until the result is a guarded process. Then, all parallel compositions are replaced by defined processes. To perform the latter step, the user has to specify all parallel compositions that might occur.

**Example 5.6.** *As an example, we consider a process* par(n,m)=p(n) $\|$ p(m)*, where* p(n) *is the process defined in Listing 5.5, page 134. Thus, the expression describes a program where two threads are running in parallel, each of them repeatedly increasing a shared location* data*, respectively* n *and* m *times. For the tool to reason about the behaviour of this process, it will automatically perform partial linearisation of the process, i.e., derive a new process* par'(n,m) *from* par(n,m) *that is sequential:*

$$\begin{aligned}
\mathsf{par}'(\mathsf{n},\mathsf{m}) &= \mathsf{p}(\mathsf{n}) \parallel \mathsf{p}(\mathsf{m}) = \cdots \\
&= (\mathsf{inc}(1).(\mathsf{p}(\mathsf{n}-1) \parallel \mathsf{p}(\mathsf{m}))) \lhd \mathsf{n} > 0 \rhd \mathsf{p}(\mathsf{m}) + \\
&\quad (\mathsf{inc}(1).(\mathsf{par}(\mathsf{m}-1) \parallel \mathsf{p}(\mathsf{n}))) \lhd \mathsf{m} > 0 \rhd \mathsf{p}(\mathsf{n}) \\
&= (\mathsf{inc}(1).\mathsf{par}(\mathsf{n}-1,\mathsf{m})) \lhd \mathsf{n} > 0 \rhd \mathsf{p}(\mathsf{m}) + \\
&\quad (\mathsf{inc}(1).\mathsf{par}(\mathsf{m}-1,\mathsf{n})) \lhd \mathsf{m} > 0 \rhd \mathsf{p}(\mathsf{n})
\end{aligned}$$

*Processes* par' *and* par *are equivalent and thus, verifying that the derived process* par' *satisfies its contract proves that* par *satisfies its contract too.*

For history processes that are very complex, it might be possible to define a second process, prove that the processes are equivalent and show that the simple process satisfies its contract. This simplifies verification because the simple process is easier to specify and verify and the equivalence proof can be carried out by external tools without considering functional specification of processes. For example, we can use the `lpsbisim2pbes` from the mCRL2 toolset [GMR+09]. However, this is not yet integrated in the tool.

*Method generation* As a second step, the transformed process is translated to a method to verify that the ensured data modifications follow logically from those specified for the actions. This translation is straightforward: all process algebra operators of sequential processes are also control flow operators in Java, except the *alternative composition* (the "+" operator). Thus, we encode this operator with an *if statement* with a randomly assigned boolean value as a condition.

**Example 5.7.** *To verify that the process par' (which is guarded and sequential) satisfies its contract, we check the following generated code (where* **if**( * ) *stands for non-deterministic choice and empty*() *is a predefined* empty process *(*$\epsilon$*)):*

```
//@ requires n ≥ 0 ∧ m ≥ 0;
//@ ensures x == \old(x) + n + m;
void par'(int n, int m){
 if ( * ) {
   if (n > 0) {
     inc(1);
     par'(n - 1, m);
   }
   else {
     else { p(m); }
   }
 }
 else {
   if (m > 0) {
     inc(1);
     par'(n, m - 1);
   }
     else { p(n); }
 }
}
```

**Verification of local history maintenance** To verify compliance with histories, the proof obligations are encoded as program specifications in plain separation logic. To achieve this, for each action segment, it is verified that the statements in the segment satisfy the requirements of the associated action. Furthermore, the encoding uses two dedicated data types: First, a class *History* is used with a constructor that encodes the rule for creating a history, and

methods that encode the other history-related rules (splitting, merging, reinitialisation or destroying a history). Second, a data type is used to replace the process algebra terms that are not a native data type of the back end.

To verify that an action is recorded properly, at the beginning of the action segment, the values of the footprint locations of the action are stored in local variables. At the end of the action segment, an assertion is set to check the validity of the postcondition of the action such that: for any location $x$, every occurrence of $\backslash\textbf{old}(x)$ in the postcondition is replaced with the value of the stored local variables corresponding to $x$. In addition, another assertion checks the precondition of the action.

## 5.7 Concurrent Class Invariants - Revisited

History-based verification has the potential for broader applicability. In this section we revisit the technique for concurrent class invariants presented in Chapter 4 and show that by thinking in terms of histories, this technique can be modified and made more permissive. Concretely, the new technique for verification of class invariants allows multiple threads to break the same class invariant simultaneously, which increases the applicability of the approach.

Below we give an informal explanation of our idea and illustrate how it works on two examples. A complete formalisation of the system is not presented; however, we argue that the formalisation and the soundness of the history-based technique from Section 5.5 supports the technique presented here. In particular, the new technique for verification of class invariants is a simplification of the more general history-based framework.

### 5.7.1 The Problem of Simultaneous Breaking of an Invariant

The protocol for verifying class invariants presented in Chapter 4 uses the two predicates $\mathsf{holds}(o.I, 1)$ and $\mathsf{break}(o.I, 1)$. These predicates give flexibility, because they control breaking and re-establishing of the invariant independently of the permissions to the footprint locations of the invariant. However, this approach prevents multiple threads from breaking the same invariant simultaneously (unless these threads are local to an unpacked segment of the invariant).

For example, Listing 5.9 illustrates a class Point (already discussed in Section 4.2), such that locations x and y are protected by two different locks lx and ly, respectively. If two threads execute in parallel methods moveX() and moveY(), after both threads join, the invariant will still be preserved. Our verification technique however, is not able to verify this scenario. In particular, the

```
class Point{
  int x, int y;
  Lock lx;  //protects x
  Lock ly;  //protects y
  //@ Invariant I: x+y≥0;
  ...
  void moveX(){
    acquire lx;
    //@ unpack I {
        x = x+1;
    //@ }
    release lx;
  }
  void moveY(){
    acquire ly;
    //@ unpack I {
        y = y+1;
    //@ }
    release ly;
  }
}
```

Listing 5.9: Simultaneous breaking of a class invariant

technique requires that both threads hold the holds$(I, 1)$ predicate to start the unpacked segment, while this is clearly not possible.

This restriction might seem natural considering the basic rules of concurrent verification; however, the approach we present here shows that we can lift this restriction. The result is a technique that is applicable to a much broader class of examples.

### 5.7.2 A New Protocol for Verifying Class Invariants

The example in Listing 5.9 shows that in the poststate of the unpacked segment in method moveX() (or respectively in method moveY()), we cannot ensure that the class invariant holds, because we have no knowledge about the value of y. However, if we allow the method contract to express the contribution of the local thread only, after both threads join, by merging the local contributions we can prove that the invariant holds.

This way of thinking suggests to use the scheme of history-based reasoning. We propose the following protocol. When multiple threads want to break the same class invariant in parallel, we can initially create an empty *global* history (represented as a process algebra term) associated to this class invariant. We can then split the history into *local* histories and distribute each split to a separate thread. Threads can record their local contributions in the form of actions to the local history. When threads join, by merging their local contributions, we can deduce the information that the class invariant holds. Each action recorded in the history represents exactly a specific unpacked segment in the program. Therefore, because actions have restricted syntax (losing permissions within the

action is not allowed), the outcome of the original program is equivalent to the outcome of a concrete trace from the global history.

Importantly, because here we are interested in verifying class invariants, the history-based concept can be sufficiently simplified. In particular, the initial property of a history over a class invariant $o.I$ is the invariant itself (note that class invariants are permissions-free expressions). The pre- and postcondition of each action is also the invariant expression. This means that there is no need to explicitly record actions to the history. If each unpacked segment guarantees that it preserves the class invariant, then without calculating all traces from the global history, we can directly conclude that every trace ends in a state in which the class invariant holds. Therefore, the original program, which has the same behaviour as one of these traces, also ends in a state in which the invariant holds. Below, we give a more detailed presentation of the new technique.

*Assuming a class invariant* A splittable predicate $\mathsf{holds}(o.I, 1)$ is associated to the class invariant $o.I$ to control whether $o.I$ is in a stable state. Any fraction of the $\mathsf{holds}(o.I, \pi)$ predicate indicates that the class invariant $o.I$ holds.

*Breaking a class invariant* Before breaking a class invariant $o.I$, the full $\mathsf{holds}(o.I, 1)$ predicate must be first consumed. This ensures that no thread may rely on the validity of $o.I$ any more. This can be done using the rule

$$\mathsf{holds}(o.I, 1) \mathrel{*\!\!-\!\!*} \mathsf{break}(o.I, 1, \mathsf{false}),$$

which states that the $\mathsf{holds}(o.I, 1)$ predicate is semantically equivalent with a predicate $\mathsf{break}(o.I, 1, \mathsf{false})$, i.e., *break predicate*. The break predicate suggests that the invariant might be broken. It is a group, and can be split and distributed among multiple threads:

$$\mathsf{break}(o.I, \pi, b) \mathrel{*\!\!-\!\!*} \mathsf{break}(o.I, \pi/2, b) * \mathsf{break}(o.I, \pi/2, b)$$

The $\mathsf{break}(o.I, 1, b)$ predicate plays the role of a history predicate over $o.I$. Exchanging the $\mathsf{holds}(o.I, 1)$ predicate for $\mathsf{break}(o.I, 1, \mathsf{false})$ can be understood as creating a history over $o.I$, while exchanging $\mathsf{break}(o.I, 1, \mathsf{false})$ for $\mathsf{holds}(o.I, 1)$ means destroying the history over $o.I$. Translated in terms of the histories, the break predicate $\mathsf{break}(o.I, \pi, b)$ represents a history over the set of locations that belong to the footprint of $o.I$, while the initial property of this history is the expression $o.I$. The last parameter $b$ is a boolean value explained below. As explained above, for a history over a class invariant, there is no need to keep

the recorded actions explicitly; thus, the break predicate contains no process algebra term parameter.

Furthermore, every update of a footprint location of a class invariant must be captured by a suitable action, i.e., an unpacked segment of $o.I$. This is why we need the last parameter of the $\mathsf{break}(o.I, \pi, b)$ predicate: $b$ is a boolean value that indicates whether the thread is within an unpacked segment. Within the unpacked segment of $o.I$, the predicate $\mathsf{break}(o.I, \pi, \mathsf{false})$ is transformed into $\mathsf{break}(o.I, \pi, \mathsf{true})$. To write on a location $p.f$ that belongs to the footprint of $o.I$, a thread must hold both write permission to $p.f$, i.e., $p.f \overset{1}{\mapsto} \_$, and any positive fraction of the predicate $\mathsf{break}(o.I, \pi, \mathsf{true})$. The rule for writing a shared location is defined as:

$$[\mathit{Write}] \quad \frac{o : T \quad S = \{I | I \in \mathit{inv}(T^2), o.f \in \mathit{fp}(o.I)\}}{\begin{array}{c}\{o.f \overset{1}{\mapsto} \_ \ * \ \circledast_{I \in S} \mathsf{break}(o.I, \pi, \mathsf{true})\} \\ \vdash \quad o.f = w \\ \{o.f \overset{1}{\mapsto} w \ * \ \circledast_{I \in S} \mathsf{break}(o.I, \pi, \mathsf{true})\}\end{array}}$$

*Re-establishing a class invariant* We discussed above that our technique requires one to prove that: if the invariant is broken within the unpacked segment, it will be re-established at the end of the segment. While in general we can accept this requirement as valid, it does require some more precise formulation.

In particular, in the pre- and poststate of the unpacked segment, we do not have read permissions for all footprint locations of the invariant and thus, we cannot state properties about the invariant - the invariant is not stable. However, because the execution of the original program is equivalent to execution of a program in which actions do not overlap (because of the syntactic restrictions within the unpacked segment), this gives us the right to reason about the code within an unpacked segment as if it is run sequentially.

The requirement above can be more precisely described as: if the unpacked segment begins in a state in which the invariant holds and the thread runs sequentially, the invariant will still hold at the end of the segment. The Hoare triple $[\mathit{Unpack}]$ is defined as:

$$[\mathit{Unpack}] \quad \frac{\begin{array}{c}\vdash \{\mathsf{break}(o.I, \pi, \mathsf{true})\} \ c \ \{\mathsf{break}(o.I, c, \mathsf{true})\} \\ \vdash_{\mathit{seq}} \{o.I\} \ c \ \{o.I\}\end{array}}{\vdash \{\mathsf{break}(o.I, \pi, \mathsf{false})\} \ \mathsf{unpack} \ o.I\{c\} \ \{\mathsf{break}(o.I, c, \mathsf{false})\}}$$

The first line in the premise of the rule requires that the code $c$ is verified using the standard rules from our reasoning system. This means that for every access

to a location, an appropriate permission is required. The second line requires that the preservation of the invariant expression should be proved in a sequential reasoning system, where no permissions for locations are required.

Therefore, what is important to understand is that we do not guarantee that the invariant holds at the beginning or at the end of the unpacked segment. The [*Unpack*] rule says that: the thread will preserve the invariant if it is run in isolation. If every thread promises this, the invariant will indeed hold at the moment when we have the full $\mathsf{break}(o.I, 1, \mathsf{false})$ predicate, i.e., when no thread writes on any footprint location of the invariant.

*Object initialisation* Initialisation of a new object is similar as with the approach explained in Section 4.2: every class invariant of the newly created object is in an unpacked state, and can directly be broken. Thus, construction of a new object $o$ produces a $\mathsf{break}(o.I, 1, \mathsf{true})$ predicate for every invariant $o.I$.

$$[New] \quad \frac{F = \circledast_{(T\,f)\in fld(C)} x.f \overset{1}{\mapsto} df(T) \quad S = inv(C)}{\begin{array}{l}\{\mathsf{true}\} \\ \vdash \quad x = \mathsf{new}\ rtype\ C\,\langle \overline{v}\rangle \\ \{F\ *\ \mathsf{fresh}(x)\ *\ \circledast_{I\in S}\ \mathsf{break}(o.I, 1, \mathsf{true})\}\end{array}}$$

After all object fields have been initialised, all object invariants $o.I$ should hold. The $\mathsf{pack}\ o$ command then consumes the $\mathsf{break}(o.I, 1, \mathsf{true})$ predicate and converts it into a new $\mathsf{holds}(o.I, 1)$ predicate.

$$[PackObj] \quad \frac{o : T \quad S = \{I\} | I \in inv(T^2)}{\vdash \{\circledast_{I\in S}(\mathsf{break}(o.I, 1, \mathsf{true})\ *\ o.I)\}\ \mathsf{pack}\ o\ \{\circledast_{I\in S}\mathsf{holds}(o.I, 1)\}}$$

### 5.7.3 Examples

We illustrate the applicability of our new approach on two examples: Example 5.8 presents the proof outline of the class Point (discussed above) and a client that uses this class, while Example 5.9 shows that the approach is also suitable for verifying more complex data structures.

**Example 5.8.** *Listing 5.10 illustrates the class Point. Fields x and y are protected by locks lx and ly, respectively, while the invariant I requires that the relation x+y≥0 is constantly preserved.*

*A class PointClient in Listing 5.11 presents the client that uses the Point class. Creating a new Point object (line 3) produces a predicate break(p.I, 1,* **true***), which*

```
class Point{
  int x;  int y;
  Lock lx;   Lock ly;
//@ Invariant I: x+y≥0;

//@ requires x ↦¹ _ * y ↦¹ _ * vx+vy≥0
//@    * break(this.I, 1, true);
//@ ensures initialised(lx)
//@    * initialised(ly) * holds(this.I, 1);
 Point(int vx, int vy){
   x=vx;
   y=vy;
   lx = new Lock/*@<x ↦¹ _>@*/();
   ly = new Lock/*@<y ↦¹ _>@*/();
{x ↦¹ vx * y ↦¹ vy * this.I * fresh(lx)
    * fresh(ly) * break(this.I, 1, true)}
//@ pack this;
{x ↦¹ vx * y ↦¹ vy * fresh(lx)
    * fresh(ly) * holds(this.I, 1)}
   commit lx;
   commit ly;
{initialised(lx) * initialised(ly)
    * holds(this.I, 1)}
 }

//@ requires break(this.I, π, false)
//@     * initialised(lx) * initialised(ly);
//@ ensures break(this.I, π, false);
void move(){
   acquire lx;  acquire ly;
   //@ unpack this.I {
{break(this.I, π, true) * x ↦¹ vx * y ↦¹vy}
     x = x+1;
     y = y−1;
{break(this.I, π, true)
    * x ↦¹ vx+1 * y ↦¹ vy−1}
   //@ }
{break(this.I, π, true)}
   release lx;  release ly;
}
```

```
//@ requires break(this.I, π, false)
//@     * initialised(lx);
//@ requires break(this.I, π, false);
void moveX(){
   acquire lx;
{break(this.I, π, false) * x ↦¹ vx}
   //@ unpack this.I {
{break(this.I, π, true) * x ↦¹ vx}
     x = x+1;
{break(this.I, π, true) * x ↦¹ vx+1}
   //@ }
{break(this.I, π, false) * x ↦¹ vx+1}
   release lx;
{break(this.I, π, false)}
}

//@ requires break(this.I, π, false)
//@     * initialised(ly);
//@ requires break(this.I, π, false);
void moveY(){
   acquire ly;
{break(this.I, π, false) * y ↦¹ vy}
   //@ unpack this.I {
{break(this.I, π, true) * y ↦¹ vy}
     y = y+1;
{break(this.I, π, true) * y ↦¹ vy+1}
   //@ }
{break(this.I, π, false) * y ↦¹ vy+1}
   release ly;
{break(this.I, π, false)}
}
```

Listing 5.10: Simultaneous breaking, the Point class

```
   class PointClient{
2    void main(){
       Point p = new Point;
4  {p.x ↦¹ 0 * p.y ↦¹ 0 * break(p.I, 1, true)}
       p.Point(2,3);
6  {initialised(p.lx) * initialised(p.ly) * holds(p.I,1)}   // the invariant p.I holds
   {initialised(p.lx) * initialised(p.ly) * initialised(p.lx) * initialised(p.ly)
8       * break(p.I, 1, false)}
     p.move();
10 {initialised(p.lx) * initialised(p.ly) * break(p.I, 1, false)}
   {initialised(p.lx) * initialised(p.ly) * holds(p.I, 1)}  // the invariant p.I holds
12 {initialised(p.lx) * initialised(p.ly) * break(p.I, 1, false)}
   {initialised(p.lx) * initialised(p.ly) * break(p.I, 0.5, false) * break(p.I, 0.5, false)}
14   fork t1;   //thread t1 calls p.moveX()
   {initialised(p.ly) * break(p.I, 0.5, false)}
16   fork t2;   //thread t2 calls p.moveY()
   {true}
18   join t1;
   {break(p.I, 0.5, false)}
20   join t2;
   {break(p.I, 1, false)}
22 {holds(p.I, 1)}   // the invariant p.I holds
    }
24 }
```

Listing 5.11: Simultaneous breaking, the PointClient class

*gives right to the current thread to modify the footprint locations of the invariant. After the object initialisation (line 5), the object is valid, and the client has the* holds(p.I, 1) *predicate which ensures that the invariant holds.*

*The client first calls the method* p.move() *(line 9) to update both* x *and* y*. The contract of the* move() *method (Listing 5.10) describes that the invariant might have been broken, but only temporary. After the method* move()*, the client has the complete* break(p.I, 1, **false**) *predicate, which ensures that the invariant holds (note that this predicate has the same semantics as* holds(p.I, 1)*).*

*Thereafter, the client forks two parallel threads: thread* t1 *calls* p.moveX() *and* t2 *calls* p.moveY()*. Each thread breaks the invariant and thus, requires a fraction of the break predicate (lines 47 and 65 in Listing 5.10). Each thread returns the fraction of the break predicate (lines 49 and 67 in Listing 5.10), which guarantees that any breaking of the invariant has been only temporarily.*

*At the end, the client needs to join both threads to obtain the complete break predicate, and ensure that the invariant is again in a stable state.*

*Note that if the invariant I was defined as $x + y == 0$, it is not possible for a thread to increase $x$ only within the unpacked segment. In this case, the same thread must also change the value of $y$ in order to re-establish the invariant. Thus, breaking of the invariant cannot be done simultaneously by multiple threads.*

**Example 5.9.** *Class* UniqueList *in Listing 5.12 represents a list of elements, each of them protected by a separate lock. The elements in the list must be unique; this requirement is expressed by the invariant* I *in line 6.*

*The* UniqueList *class contains a method* switchItems(**int** *i,* **int** *j), which exchanges the values of the elements at positions* i *and* j*. To execute this method, it is sufficient for a thread to acquire only the locks that protect the elements at the i-th and j-th position. The other locks remain available and can be used by other threads to switch elements at the other positions. Thus, multiple threads may break the invariant simultaneously, but if each thread promises to re-establish the invariant after its breaking, we can conclude that when no thread operates on the list, the invariant is preserved.*

*The client class is represented in Listing 5.13. It initialises a new* UniqueList *object* list *(line 6), and therewith, obtains the* holds(list.I,1) *predicate. It then trades this predicate for a break predicate which is split and distributed to two parallel threads. Thread* t1 *needs a fraction of the break predicate to call the method* list.switchItems(0,1) *(line 9), while* t2 *needs a fraction to call the method* list.switchItems(8,9) *(line 12). Both threads re-establish the invariant after breaking; therefore, when threads join (line 16), the client obtains the full predicate* break(list.I, 1, **false**) *and concludes that the invariant holds.*

## 5.8 Conclusions and Related Work

We discussed in this chapter about verification of *functional properties* in concurrent programs. Although crucially important, these properties are notoriously difficult to verify. A functional property describes what the program is actually expected to do; thus it needs to be manually specified. Moreover, a practical verification technique should be modular, which requires specifying the behaviour of every component (method/thread). Unfortunately, this causes problems in a concurrent program, because any external thread can change the behaviour of the thread that we describe.

```
    class UniqueList {
2     int[] items;
      Lock[] locks;
4     int size=10; //we assume the field is final (readonly)
                   //and no permissions are needed to read it
6   //@ Invariant I: ∀ i=0...size-1. ∀ j=i+1...size-1. items[i] ≠ items[j];

8   //@ requires items↦^1_ * break(I, 1, true);
    //@ ensures holds(I, 1) * ⊛_{i=0...size−1}.initialised(locks[i]);
10    public UniqueList() {
        items = new items[size];
12  {items↦^1_ * ⊛_{i=0...size−1}. items[i]↦^1_ * break(I, 1, true)}
        for (int i = 0; i < size; i++) {
14        items[i] = i;
          locks[i] = new /*@<items↦^{1/size}_ * items[i]↦^1_>@*/Lock();
16      }
    {items↦^1_ * ⊛_{i=0...size−1}. items[i]↦^1 i * fresh(locks[i]) * break(I, 1, true)}
18      pack this;
    {items↦^1_ * ⊛_{i=0...size−1}. items[i]↦^1 i * fresh(locks[i]) * holds(I, 1)}
20      for (int i = 0; i < size; i++) {
          commit locks[i];
22      }
    {holds(I, 1) * ⊛_{i=0...size−1}.initialised(locks[i])}
24    }
    //@ requires break(this.I, π, false) * 0≤i<size * 0≤j<size
26  //@     * initialised(locks[i]) * initialised(locks[j]);
    //@ requires break(this.I, π, false);
28    void switchItems(int i, int j) {
        acquire locks[i];
30      acquire locks[j];
    {break(this.I, π, false) * items↦^π_ * items[i]↦^1_ * items[j]↦^1_}
32      //@ unpack this.I{
    {break(this.I, π, true) * items↦^π_ * items[i]↦^1_ * items[j]↦^1_}
34      int k = items[i];
        items[i] = items[j];
36      items[j] = k;
    {break(this.I, π, true) * items↦^π_ * items[i]↦^1_ * items[j]↦^1_}
38    //@ }
    {break(this.I, π, false) * items↦^π_ * items[i]↦^1_ * items[j]↦^1_}
40    release locks[i];
      release locks[j];
42  {break(this.I, π, false)}
    }
44  }
```

Listing 5.12: Simultaneous breaking, the UniqueList class

```
class UniqueListClient {
  void main(){
    UniqueList list = new UniqueList;
{list.items ↦^1 _ * ⊛_{i=0...size−1}. list.items[i] ↦^1 0 * break(I, 1, true)}
    list.UniqueList();
{⊛_{i=0...size−1}.initialised(locks[i]) * holds(list.I, 1)}   // the invariant list.I holds
{⊛_{i=0...size−1}.initialised(locks[i]) * initialised(locks[0]) * initialised(locks[1]) *
     break(list.I, 1, false)}
    fork t1;   //thread t1 calls list.switchItems(0,1)
{⊛_{i=0...size−1}.initialised(locks[i]) * initialised(locks[8]) * initialised(locks[9]) *
     break(list.I, 0.5, false)}
    fork t2;   //thread t2 calls list.switchItems(8,9)
{⊛_{i=0...size−1}.initialised(locks[i]) * break(list.I, 0.25, false)}
    join t1;
    join t2;
{⊛_{i=0...size−1}.initialised(locks[i]) * break(list.I, 1, false)}
{holds(list.I, 1)}   // the invariant list.I holds
  }
}
```

Listing 5.13: Simultaneous breaking, the UniqueListClient class

**Related Work** The problem of modular verification of functional properties has been investigated by Jacobs and Piessens [JP11]. They proposed a technique (which we already discussed in Section 5.1 (page 123) based on the Owicki-Gries method, which allows passing auxiliary update code to methods. This results in a kind of a higher-order programming that allows modular reasoning about fine-grained data structures. As discussed, this logic is expressive but it requires the user to define an invariant that remains stable under the execution of the parallel threads, which is usually a complicated task.

Another similar approach to reason about the functional behaviour of concurrent programs is by using *Concurrent Abstract Predicates (CAP)* [DYDG$^+$10], which extends separation logic with *shared regions*. A specification of a shared region describes possible interference, in terms of actions and permissions to actions. These permissions are given to the client thread to allow them to execute the predefined actions according to a hardcoded usage protocol. A more advanced logic is the extension of this work to iCAP (Impredicative CAP) [SB14], where a CAP may be parameterised by a protocol defined by the client.

Compared to these approaches, histories are in a way a ghost code that keeps

track of the local contributions. We use process algebra to combine the local histories: this allows avoiding the need to specify the behaviour of the threads in an invariant. We do use invariants related to every lock, but by using histories, we intend to use these invariants for storing permissions only. Therefore, we believe histories allow more natural specifications.

Strongly related to our work is the recently proposed prototype logic of Ley-Wild and Nanevski [LN13], the *Subjective Concurrent Separation Logic (SCSL)*. They allow modular reasoning about coarse-grained concurrent programs by verifying the thread's local contribution with respect to its *local view*. When views are combined, the local contributions are combined. To this end, the logic contains the *subjective separating conjunction* operator, $\circledast$, which splits (merges) a heap such that the contents of a given location may also be split: $l \mapsto a \oplus b$ is equivalent to $l \mapsto a \circledast l \mapsto b$. The user specifies a *partial commutative monoid (PCM)*, $(\mathbb{U}, \oplus, \nvdash)$, with a commutative and associative operator $\oplus$ that combines the effect of two threads and where $\nvdash$ describes no effect. To solve the Owicki-Gries example, a PCM $(\mathbb{N}, +, 0)$ is chosen: threads local contributions are combined with the + operator. However, if we extend this example with a third parallel thread that for example multiplies the shared variable by 2, we expect that the choice of the right PCM will become troublesome.

In contrast to their technique, our histories are stored as parallel processes of actions that are resolved later. In a way we use a PCM where contributions of threads are expressed via histories, and these threads effects are combined by the process algebra operator $\|$. This makes our approach easily applicable to various examples (including the example described above). Moreover, our method is also suited to reason about programs with dynamic thread creation.

Furthermore, also closely related to our approach is the work on *linearisability* [Vaf07, Vaf10]. A method is linearisable if the system can observe it as if it is atomically executed. Linearisability is proved by identifying *linearisation points*, i.e., points where the method takes effect. Linearisation points roughly correspond to our action specifications. Using linearisation points allows one to specify a concurrent method in the form of sequential code, which is inlined in the client's code (replacing the call to the concurrent method). In a similar spirit, Elmas et al. [EQT09] abstract away from reasoning about fine-grained thread interleavings, by transforming a fine-grained program into a corresponding coarse-grained program. The idea behind the code transformation is that consecutive actions are merged to increase atomicity up to the desired level. Recently, a more powerful form of linearisation has been proposed, where multiple synchronisation commands can be abstracted into one single linearisation action [HR14]. It might be worth investigating if these ideas carry over to our

approach, by adding different synchronisation actions to the histories.

Recently, some very promising parameterisable logics have been introduced [DYBG+13, JSS+] to reason about multithreaded programs. The concepts that they introduce are very close to our proof logic. Reusing such a framework will be useful to simplify the formalisation and justify soundness of our system, as well as to show that the concept of histories is more general and applicable in other variations of separation logic. However, in their current form, they can be used as a foundation only for simplified versions of our logic. In particular, to the best of our knowledge, they are not directly applicable to our language as it contains dynamic thread creation instead of the parallel $\|$ operator.

**Our technique** We proposed a novel history-based technique for modular verification of functional behaviour of concurrent programs. This technique allows one to provide intuitive method specifications that describe only the *local effect* of a thread, in terms of abstract (user-specified) *actions*, which reduces the need to reason about fine-grained thread interleavings. The technique is an extension of permission-based separation logic. It is particularly suited to reason about programs with internal synchronisation, and notably, when access to certain locations is protected by multiple locks. Support for the approach is added to the VerCors tool set [BH14].

Currently, we can identify two potential weak points of our approach:

i) First, while a history $H$ is built modularly, reasoning about $H$ is non entirely modular because it involves calculating thread interleavings. However, we not consider this as a significant problem because: i) the history abstracts away all unnecessary details and makes the abstraction simpler than the original program; ii) the history mechanism is integrated in a standard modular program logic, such that histories can be employed to reason only about parts of the program where modular reasoning is troublesome; and iii) we allow the global history to be *reinitialised* (to be emptied), and moreover, to be destroyed. Thus, the management of histories allows keeping the abstract parts small, which makes reasoning more manageable.

ii) Second, histories increase the amount of specifications in the program; concretely, the user has to specify actions, processes and specification commands for managemenet of histories. The positive side of this point is that these specifications are very intuitive, easy to understand, and are not more complex than specifications of a sequential program.

*Future work* As a further step, it would be interesting to study how this approach can be applied to programs with fine-grained concurrency. For these programs, verification is more challenging, but very important because these are programs highly prone to errors.

Another point is investigating when process algebra simplifications can be applied during the construction of the history, without comprising soundness. We expect that such simplifications will be possible only in certain scenarios, and in these cases the result will be a simpler process algebra term, which is easier to reason about.

Finally, we believe that our history-based approach can be used to reason about distributed software. This will require more variations in how the global history can be derived from the local histories, but we expect that apart from this, most of the approach directly carries over.

# Chapter 6

# Verification of Programs with Guarded Blocks

In this chapter we move a step towards total correctness of concurrent programs. Total correctness means *correctness and termination*. In a concurrent program, there might be various reasons why a program does not terminate. Proving total correctness is therefore a difficult task; instead, verification techniques normally focus on proving only a specific liveness property. A typical example of a liveness property is *deadlock-freedom*.

Concretely, this chapter studies liveness properties of programs with *guarded blocks* [Lea99]. Guarded blocks are used on top of a lock-based mechanism to allow threads to synchronise on the value of the shared data. Intuitively, a guarded block is a block of code "guarded" by some condition: if the condition holds, the thread may enter the block; otherwise, it has to wait until another thread brings the program into a state in which the condition is satisfied. A common pattern for the use of guarded blocks is a producer/consumer pattern using a shared buffer, where producers need information about the buffer being non-full, and consumers need information about the buffer being non-empty.

Guarded blocks are useful, but they make reasoning about programs rather challenging. First, they affect the behaviour of the program because the guards may determine the order of thread executions and second, they may cause a thread to *block*, waiting forever to enter a specific guarded block.

In Chapter 1 we formulated the question:

*Challenge 3: How to verify functional and non-blocking properties of programs with guarded blocks?*

This chapter addresses this challenge with a novel technique for reasoning about programs with *guarded blocks*. Our technique builds on our history-related approach presented in Chapter 5. While in Chapter 5 we used histories (defined as process algebra terms) to abstract the functional behaviour of the program, here a history captures both the functional and blocking behaviour of the program. Moreover, we also allow reasoning about programs, where threads may have infinite executions. In that case, the abstract model is called a *future*: it has to be predicted in advance, while local reasoning is used to show that the program indeed respects the predicted abstract future.

To keep our presentation simple, we use a simplified procedural language, where parallelism is provided by the parallel construct $\|$ (instead of the more complicated fork/join constructs used in Chapters 4 and 5), while synchronisation is maintained via synchronised blocks (instead of acquire/release locks). The approach however can be generalised to a more complicated language. We give a complete formalisation of the approach and prove it sound.

**Outline** This chapter is structured as follows. Section 6.1 illustrates on a simple example why reasoning about programs with guarded blocks is difficult, and gives an overview of our reasoning strategy. Section 6.2 provides a detailed informal explanation of our approach, discussing separately history-based reasoning in Section 6.2.1 and future-based reasoning in Section 6.2.2. We formalise the approach in Section 6.3: Section 6.3.1 shows the language syntax, Section 6.3.2 presents its semantics, Section 6.3.3 illustrate the complete proof system, and Section 6.3.4 discusses how one can reason about the abstract model. Section 6.3.5 discusses soundness of the system. Finally in Section 6.4 we discuss related work and conclude.

## 6.1 The Problem of Reasoning about Guarded Blocks

In multithreaded programs, threads share resources and cooperate with each other. To synchronise threads and establish controlled access to shared data, typically a *lock-based mechanism* is used. In some cases, threads additionally synchronise on the value of some data. This is achieved using *guarded blocks*.

A guarded block is a block of code, synchronised on a lock $l$, which can be entered by a given thread only if a certain condition, i.e., *a guard*, holds. If

```
    void m1(){                    8  void m2(){
2     sync(lock){                      sync(lock){                 16  void client(){
        wait ([x] ≠1, lock);      10      wait ([x] ≠ 0, lock);           x = cons(0);
4         [x] = 1;                          [x] = 0;              18    m1() || m2();
        notifyAll(lock);          12      notifyAll(lock);              }
6     }                                 }
    }                             14  }
```

Listing 6.1: Guarded blocks

the guard is not satisfied, the thread has to wait: it releases the lock $l$, goes into $l$'s *waiting room* and waits for a notification. When another thread sends a notification on $l$, the waiting thread leaves the waiting room and tries to re-obtain the lock $l$; after successfully accessing the lock, the thread tries again to enter the guarded block by checking again the validity of the guard. A lock may also be associated with a set of *condition variables*, each of them defining its own waiting room. Waiting and notifying is then done on the condition variable, and not directly on the lock.

Normally programming languages include two mechanisms for sending a notification: i) *notifyAll*: wakes up all threads from the waiting room, such that all of them start to compete for the lock; ii) *notify*: sends the notification to only a single thread from the waiting room, chosen by a specific strategy. We consider in our language only notifications of the type *notifyAll*.

**Example 6.1.** *Consider for example the program in Listing 6.1. We use* x=cons(v) *to initialise a new location* x *to a value* v *and square brackets to denote pointer dereferencing. The client initialises a shared location* x *to* 0 *(line 17) and starts two threads to execute methods* m1() *and* m2() *in parallel. The command* wait([x]≠0, lock) *in line 10 forces the thread to wait in the* lock*'s waiting room if it tries to execute the block when the guard* x≠0 *does not hold. Thus, for this particular program, the guards in* m1() *and* m2() *determine the order of thread execution: the update of* x *in* m1() *must precede the one in* m2()*.*

Guarded blocks make verification difficult. First of all, they impede verification of functional properties. We discussed in Chapter 5 that to reason about values of shared variables (i.e., functional properties), we need to consider all possible thread interleavings. However, guarded blocks practically disallow certain interleavings. Thus, in order to characterise precisely the possible data values, one needs to identify the set of possible interleavings precisely. Secondly, guarded blocks may lead the program to a *blocking state*, i.e., a state in which all

threads wait forever. Whether the program will block depends on the functional behaviour and the values of the shared data.

*Our solution* The method that we propose in this chapter addresses the two challenges mentioned above: it allows verifying functional properties while considering the possible interleavings only, and moreover, it allows one to prove non-blocking of the program. For example, using our technique we can prove for the program in Listing 6.1 that: i) both threads will terminate; and ii) at the end of the program, the value of x is 0.

The technique uses the ideas of the history-based approach presented in Chapter 5. Local reasoning is used to capture the blocking behaviour of parts of the program in an abstract model, i.e., *a history*. By reasoning about the history, we prove properties about the original program. Moreover, we extend this approach with *future-based reasoning* where the model is predicted in advance and is called a *future*; by using local reasoning we verify whether the program behaves as prescribed in the model. Future-based reasoning is useful to reason about programs with non-terminating threads.

## 6.2 Abstracting programs to process algebra terms

This section gives an informal explanation about our history/future-based local reasoning used to build the abstract model. Later in Section 6.3 we formalise the approach and also explain the process of reasoning about the abstract model.

**Locks and conditions** To reason about locks, we use a similar protocol as in Chapters 4 and 5, but here adapted to a procedural language. It is inspired by the program logic of Gotsman et al. [GBC+07]. The command $\mathsf{initLock}(x, K)$ converts the heap location $x$ to a *lock*, associating a *lock label* $K$ to it. The thread that creates the lock must provide a write permission for the location $x$, which is then consumed and converted into a predicate $\mathsf{Lock}_K(x, 1, \mathsf{false})$:

$$\{x \overset{1}{\mapsto} \_\}\ \mathsf{initLock}(x, K)\ \{\mathsf{Lock}_K(x, 1, \mathsf{false})\}$$

The second parameter in $\mathsf{Lock}_K(x, 1, \mathsf{false})$ is used to make it a splittable predicate, i.e., a group. A fraction $\mathsf{Lock}_K(x, \pi, \mathsf{false})$ can be understood as a permission to the lock $x$; it does not give a thread a permission to access the location $x$, but it gives the right to acquire the lock. The fact that the predicate is splittable means that it can be distributed to multiple threads that can compete to win

the lock. The last boolean parameter is used to indicate whether the lock $x$ is associated to a model (a history or a future).

For every label $K$, a *resource invariant* $I_K$ is specified, i.e., a predicate expressing the permissions protected by the lock. Reasoning is done by transferring the resource invariant between the lock and threads that obtain the lock as explained in Chapter 4.

Similarly, the command $\mathsf{initCond}(x, y)$ is used to convert the location $x$ into a condition, associating it to an existing lock $y$. A splittable predicate $\mathsf{Cond}_K(x, 1)$ is then produced, where $K$ is the lock label of the lock $y$. A fraction $\mathsf{Cond}_K(x, \pi)$ is used as a permission to use the condition $x$.

$$\{x \overset{1}{\mapsto} \_ * \mathsf{Lock}_K(y, \pi, b)\}\ \mathsf{initCond}(x, y)\ \{\mathsf{Cond}_K(x, 1) * \mathsf{Lock}_K(y, \pi, b)\}$$

Given an expression $e$, we call the set of locations referred to by $e$ the *footprint of* $e$, denoted $fp(e)$. The footprint of a lock $l$ is the set of locations referred to by the resource invariant of $l$, *i.e.* $fp(l) = fp(res\_inv(l))$. For a set of locks $L$, we define $fp(L) = \bigcup_{l \in L} fp(l)$.

**Synchronised blocks** Each action recorded in the process algebra term represents a concrete *synchronised block* in the program, which respects the form in Figure 6.1. The *wait* and *notify* locations, $x_w$ and $x_n$ respectively, can either be the lock $x$, or conditions associated to the lock $x$. The *guard* $g$ may only refer to locations protected by $x$, $fp(g) \subseteq fp(x)$. The *wait* and *notifyAll* operations are optional. If *wait* is absent, we say that the guard of the block is *true* and the wait location is *null*. Similarly, the notify location is *null* if there is no *notifyAll* operation. The program segment has a restricted syntax and does not allow nested synchronised blocks (we discuss this more extensively on page 177).

```
sync (x) {
  wait (g, x_w); // optional command
    // program segment
  notifyAll(x_n); // optional command
}
```

Figure 6.1: A synchronised block

### 6.2.1 Abstracting to Histories

We use histories when program threads have finite executions only. A history refers to a set of locks $L$; we call it a *history over* $L$. It is used to trace all synchronised blocks over locks from $L$, such that each block is recorded in the history in the form of an action. This approach allows more histories to exist in the program at the same time, but their set of locks must be disjoint. This partitioning of the memory makes reasoning simpler because the abstractions can only relate to smaller parts of the program.

A history over $L$ is described by the predicate $\mathsf{Hist}(L, 1, R, M)$ where: $R$ is the *initial property*, i.e., a predicate that holds in the *initial state of the history*, i.e., the state in which the history is empty, while $M$ is a process algebra term representing the abstract model, in this case the history.

**Creating a history** To create a history over $L$, we use the specification command $\mathsf{crhist}(L, R)$, where $R$ is a predicate that must hold in the current state and that refers to locations protected by the locks in $L$, i.e, $\mathit{fp}(R) \subseteq \mathit{fp}(L)$. This command requires a full predicate $\mathsf{Lock}_K(x, 1, \mathsf{false})$ for every lock $x \in L$, converts it to $\mathsf{Lock}_K(x, 1, \mathsf{true})$, and produces a predicate $\mathsf{Hist}(L, 1, R, \epsilon)$ with an empty process $\epsilon$. This prevents the co-existence of more than one history over the same lock, and ensures that all actions are recorded in the same history.

**Splitting and merging histories** To build the history in a modular way, the history predicate may be split into parts. A fraction $\pi$ of the predicate is denoted by $\mathsf{Hist}(L, \pi, R, M)$, $\pi \in (0, 1]$. If $\pi = 1$, the history is *global* (complete), while for $\pi < 1$ the history is *local* (incomplete). When several threads running in parallel operate on the history over $L$, each thread obtains an empty local history, i.e., a fraction $\mathsf{Hist}(L, \pi, R, \epsilon)$ to record their actions to. When threads join, the client combines all fractions by merging the histories:

$$\mathsf{Hist}(L, \pi_1, R, M_1) * \mathsf{Hist}(L, \pi_2, R, M_2) \mathrel{*\!\!-\!\!*} \mathsf{Hist}(L, \pi_1 + \pi_2, R, M_1 \parallel M_2)$$

If the client has a global history $\mathsf{Hist}(L, 1, R, M)$, $M$ represents the abstraction of the program between the initial state of the history and the current state. We reason about the program using the initial property $R$, and the model $M$.

**Recording actions** To record synchronised blocks in the history, each block $c$ should be specified by an action identifier and a list of parameter values:

```
sync /*@< a(v_1, ..., v_n) >@*/ (x) {
  c;
}
```

This associates the block of code $c$ with a predefined action $a(x_1, ..., x_n)$; we call $c$ *the action implementation*. The predefined actions are part of the program specification, defined as abstract methods: they have a contract (pre- and postcondition) and no body.

The precondition $F$ and postcondition $F'$ of the associated action may only contain locations protected by the lock, $fp(F) \cup fp(F') \subseteq fp(x)$. It is required that the block $c$ satisfies the contract of the action $a(v_1, ..., v_n)$. In the poststate of the synchronised block, the action is recorded in the history in the form $a(v_1, .., v_n, a_g, a_w, a_n)$, where the last three parameters represent respectively: the guard, the wait location, and the notify location of the synchronised block. The action $a$ is recorded to a history over $L$, where $x \in L$. Thus, to start the synchronised block, a local history $\mathsf{Hist}(L, \pi, R, M)$ is required, where $x \in L$.

**Reinitialising a history** When the global history $\mathsf{Hist}(L, 1, R, M)$ is obtained, we have the knowledge $R$ of the state $\sigma_0$ when the history has been created and the complete abstracted model $M$ between $\sigma_0$ and the current state $\sigma$. The model can then be used to reason about the original program. In particular, if for some property $R'$ we prove that the model $M$ *terminates from* $R$ *to* $R'$, denoted $M{\downarrow_{R,R'}}$, this ensures that up to the state $\sigma$ the original program is non-blocking with respect to $L$, and $R'$ holds in the new state $\sigma$.

The global history may be reinitialised (emptied) by using the specification command $\mathsf{reinit}(L, R')$, where $R'$ is a boolean formula expected to hold over the current state. After reinitialisation, the $\mathsf{Hist}(L, 1, R, M)$ predicate is converted into $\mathsf{Hist}(L, 1, R', \epsilon)$. Reinitialising is allowed only if $M{\downarrow_{R,R'}}$. The current state becomes the new initial state of the history, and $R'$ becomes the new initial property. In this way, the history can safely forget about the previous behaviour of the program, and after the restarting point, the new actions will be recorded to a fresh empty history. Reinitialising allows to keep the abstract parts smaller, which makes reasoning more manageable.

**Example 6.2.** *In Listing 6.2 and Listing 6.3 we illustrate the history-based reasoning on the common example: a producer and a consumer are using a shared queue (represented by an array), while synchronising on the size of the buffer.*

*The client (Listing 6.2): In line 5 the client initialises an array* q *with capacity* 5*, and stores the current size of the array* size=0*. All values in the array*

```
2  //@ I_K: ⊛_{i=0...4}. q+i ↦¹v * size ↦¹ _;

4  void client{
     q = cons(0,0,0,0,0);
6    size = cons(0);
     lock = cons(0);
8  { ⊛_{i=0...4}. q+i ↦¹ 0 * size ↦¹ 0 * lock ↦¹ 0}
     initLock(lock, K);
10   nF = cons(0);     // notFull
     nE = cons(0);     //notEmpty
12   initCond(nF, lock);
     initCond(nE, lock);
14 {⊛_{i=0...4}. q+i ↦¹ 0 * size ↦¹ 0
       * Cond_K(nE,1) * Cond_K(nF,1) * Lock_K(lock,1,false)}
16 //@ crhist({lock}, size==0)
   {⊛_{i=0...4}. q+i ↦¹ 0 * size ↦¹ 0
18     * Cond_K(nE,1) * Cond_K(nF,1) * Lock_K(lock,1,true) * Hist(lock, 1, size==0, ϵ) }
     commit(lock);
20 {Cond_K(nE,1) * Cond_K(nF,1) * Lock_K(lock,1,true) * Hist(lock, 1, size==0, ϵ)}
   {Cond_K(nE,1) * Cond_K(nF,1) * Lock_K(lock,1,true)
22   * Hist(lock, 0.5, size==0, ϵ) * Hist(lock, 0.5, size==0, ϵ)}
     produce(10) || consume(7);
24 {Cond_K(nE,1) * Cond_K(nF,1) * Lock_K(lock,1,true)
       * Hist(lock, 1, size==0, pa(10) || pr(7))}
26 //@ reinit({lock}, size==3);
   {Cond_K(nE,1) * Cond_K(nF,1) * Lock_K(lock,1,true) * Hist(lock, 1, size==3, ϵ)}
28 }
```

Listing 6.2: History-based specifications, the client

*are initially set to* 0*, which indicates no value. Access to the array is protected by a lock* lock*, initialised as a location in line 7 and converted to a lock in line 9. Its associated invariant is* $I_K$ *(line 2). In lines 12 and 13 locations* nF *and* nE *are converted to conditions associated to* lock *(this produces a handle* Cond *for every condition). The client creates a global history (line 16) with initial knowledge* size=0*, which has to trace all synchronised blocks on* lock*. Afterwards, two threads are started (line 23): the producer adds the elements* 10 *to* 1 *to the queue; and the consumer removes* 7 *elements from the queue.*

*Producer/consumer (Listing 6.3): Both the producer and consumer threads obtain a local history to record its actions to (see lines 10 and 25). The synchronised blocks at lines 15, 30 are associated with predefined actions (lines 2, 5). In addition to actions, we also define* processes *(lines 7, 8),* i.e. *process algebra terms with a concrete definition. The contracts of the methods* produce *and* consume *describe the local contribution as an extension of the local history with a recursive* process *(lines 12, 27).*

*The client (Listing 6.2): At the end, when the client joins both threads (line 25), local histories are merged to a global history* M=pa(10) ‖ pr(7)*. Because* M *terminates from* R: size==0 *to* R': size==3 *($M{\downarrow_{R,R'}}$), we prove that the program is non-blocking and that* R' *holds in the current state. Finally, we can reinitialise the history, so that reasoning about the continuation of the program is done on an empty history with* R' *as a new initial property.*

### 6.2.2 Abstracting to Futures

In programs where threads have infinite executions, the abstract model $M$ has *infinite* traces and is called a *future*. A future is represented by a predicate $\mathsf{Future}(L, 1, R, M, b)$. The predicate has the same meaning as $\mathsf{Hist}(L, 1, R, M)$ except that $M$ represents the predicted abstraction of the program starting from the current state. The boolean parameter $b$ indicates whether the future is newly created and thus, $M$ is *complete*, i.e., equal to the initially predicted future.

**Creating a future** A future over locks $L$ is created using the specification command $\mathsf{crfut}(L, R, M)$, where $M$ is the predicted model. The command converts the predicates $\mathsf{Lock}_K(x, 1, \mathsf{false})$ for every $x \in L$, to $\mathsf{Lock}_K(x, 1, \mathsf{true})$ and provides a predicate $\mathsf{Future}(L, 1, R, M, \mathsf{true})$, which contains the complete future model ($b = \mathit{true}$). We can then prove that the original program is non-blocking by proving that the model $M$ *is not blocking from* $R$, denoted $M{\downarrow_R}$.

**Compliance with futures** Because the future is predicted in advance, it has to be verified that the program conforms to the predicted future. Similar to using histories, global futures may also be split into local futures that can be used by parallel threads simultaneously. Notably, the *precondition* of a given method describes the predicted contribution of the method by defining the local future. Each thread has to show that it executes the actions as prescribed in the local future. Thus, in the prestate of the synchronised block in Figure 6.1, it is required that the local future has the form $a(v_1, ..., v_n, g, x_w, x_n) \cdot M$. In

```
//@ ensures size=\old(size) + 1
//@ action a();

//@ ensures size=\old(size) - 1
//@ action r();

//@ proc pa(n) = a([size]<5, nF, nE)·pa(n - 1) ◁ n>0 ▷ ϵ;
//@ proc pr(n) = r([size]>0, nE, nF)·pr(n-1) ◁ n>0 ▷ ϵ;

//@ requires History(L, π, R, H) * lock ∈ L * Lock_K(lock, π1, true)
//@         * Cond_K(nF, π2) * Cond_K(nE, π3) * x>0;
//@ ensures History(L, π, R, H·pa(x)) * Lock_K(lock, π1, true)
//@         * Cond_K(nF, π2) * Cond_K(nE, π3);
void produce (x) {
  sync /*@<a()>@*/(lock) {
    wait([size] < 5, nF);
     y = [size];
     [q+y] = x;
     [size] = y+1;
    notifyAll(nE);
  }
  if (x>1) produce(x - 1);
}

//@ requires History(L, π, R, H) * lock ∈ L * Lock_K(lock, π1, true)
//@         * Cond_K(nF, π2) * Cond_K(nE, π3) * x>0;
//@ ensures History(L, π, R, H·pr(x)) * Lock_K(lock, π1, true)
//@         * Cond_K(nF, π2) * Cond_K(nE, π3);
void consume (x) {
  sync /*@<r()>@*/(lock) {
    wait ([size] >0, nE);
     y = [size];
     [q+y - 1] = 0;
     [size] = y - 1;
    notifyAll(nF);
  }
  if (x>1) consume(x - 1);
}
```

Listing 6.3: History-based specifications, producer/consumer

the poststate of the action, the action $a$ is removed from the future. Moreover, every synchronised block must respect the contract of the associated action. Any change of the initially predicted future (splitting the predicate or removing an action) changes the boolean parameter to *false* and thereafter, no reasoning about the abstract model may be performed.

**Example 6.3.** *To illustrate the future-based reasoning, we modify Example 6.2 such that: the producer thread now adds an element to the queue infinitely often, while the consumer infinitely often removes an element (see Listing 6.5).*

*Listing 6.4 illustrates the client. In this case the client creates a global future (line 22) with initial property* R: size=0, *and a predicted model* M=pa() ‖ pr(). *The future is then complete (*Future(lock, 1, size==0, pa() ‖ pr(), **true**)*) and therefore, we can prove non-blocking of the program, by proving that* M *is non-blocking from* R, $M{\downarrow_R}$. *The client then splits the future into two local futures* pa() *and* pr() *and distributed them to both threads. Each thread has to behave as prescribed in the local future.*

**Nested synchronised blocks** Using guarded blocks within a nested synchronised block is risky and can easily lead to a deadlock. Assume that the synchronised block in Figure 6.1 is nested into another synchronised block over a lock $y$. If the guard $g$ in line 2 does not hold, the current thread releases the inner-most lock only, i.e., $x$, and starts to wait while still holding $y_m$. If another thread needs the lock $y$ to change the state of the guard, the program is deadlocked.

To simplify the presentation, we restricted our formalisation to non-nested synchronised blocks only. However, it can be extended to programs with nested blocks. In particular, within a synchronised block it is safe to allow any code (including also other synchronised blocks) under the condition that this code terminates. This requires one to prove that the code is non-blocking with respect to any lock. This requirement is important to avoid deadlocks, because it ensures that a thread never holds a lock that is necessary by another thread to change the state of the guard.

```
2
   I_K: ⊛_{i=0...4}. q+i ↦^1 _ * size ↦^1 _;
4

6  void client{
    q = cons(0,0,0,0,0);
8   size = cons(0);
    lock = cons(0);
10 {⊛_{i=0...4}. q+i ↦^1 0 * size ↦^1 0 * lock ↦^1 0}
    initLock(lock, K);
12 {⊛_{i=0...4}. q+i ↦^1 0 * size ↦^1 0 * lock ↦^1 0 * Lock_K(lock, 1, false)}
    nF = cons(0);    //notFull
14  nE = cons(0);   //notEmpty
   {⊛_{i=0...4}. q+i ↦^1 0 * size ↦^1 0 * nF↦^1 0 * nE ↦^1 0
16       * lock ↦^1 0 * Lock_K(lock, 1, false)}
    initCond(nF, lock);
18  initCond(nE, lock);
   {⊛_{i=0...4}. q+i ↦^1 0 * size ↦^1 0 * lock ↦^1 0 * Cond_K(nE,1) * Cond_K(nF,1)
20      * Lock_K(lock, 1, false)}

22 //@ crfut({lock}, size==0, pa() || pr());
   {⊛_{i=0...4}. q+i ↦^1 0 * size ↦^1 0 * lock ↦^1 0 * Cond_K(nE,1) * Cond_K(nF,1)
24     * Lock_K(lock, 1, false)} * Future(lock, 1, size==0, pa() || pr(), true)}
   commit (lock);
26 {Cond_K(nE,1) * Cond_K(nF,1) * Lock_K(lock, 1, false)}
       * Future(lock, 1, size==0, pa() || pr(), true)}
28 {Cond_K(nE,1) * Cond_K(nF,1) * Lock_K(lock, 1, false)}
       * Future(lock, 1, size==0, pa(), false) * Future(lock, 1, size==0, pr(), false)}
30

    produceForever(1) || consumeForever();
32 }
```

Listing 6.4: Future-based specifications

```
  //@ ensures size=\old(size)+1
2 //@ action a();

4 //@ ensures size=\old(size)−1
  //@ action r();
6
  //@ proc pa() = a([size]<5, nF, nE)·pa();
8 //@ proc pr() = r([size]>0, nE, nF)·pr();

10 //@ requires Future(L, π, R, pa(), b) ∗ lock ∈ L ∗ Lock_K(lock, π1, true)
   //@          ∗ Cond_K(nF, π2) ∗ Cond_K(nE, π3) ∗ x>0;
12 void produceForever (x) {
    sync /*@<a()>@*/(lock) {
14    wait ([size] < 5, nF);
      y = [size];
16    [q+y] = x;
      [size] = y+1;
18    notifyAll(nE);
    }
20  if (x<100) x=x+1 else x=1;
    produceForever(x);
22 }

24 //@ requires Future(L, π, R, pr(), b) ∗ lock ∈ L ∗ Lock_K(lock, π1, true)
   //@          ∗ Cond_K(nF, π2) ∗ Cond_K(nE, π3) ∗ x>0;
26 void consumeForever () {
    sync /*@<r()>@*/(lock) {
28    wait ([size] >0, nE);
      y := [size];
30    [q+y - 1] = 0;
      [size] = y - 1;
32    notifyAll(nF);
    }
34  consumeForever();
  }
```

Listing 6.5: Future-based specifications

| | | | |
|---|---|---|---|
| Variables | $x, y, z \in \mathsf{Var}$ | | |
| Lock labels | $K \in \mathcal{L}$ | | |
| Values | $v \in \mathsf{Value}$ | $::=$ | $n \mid b \mid \pi \mid M$ |
| Program def | $pgm$ | $::=$ | $\overline{md}\ \overline{pd}\ \overline{act}\ \overline{proc}$ |
| Method def | $md$ | $::=$ | $\mathsf{requires}\ F\ \mathsf{ensures}\ F\ \mathsf{void}\ m(\overline{x})\{c\}$ |
| Integer expr | $e$ | $::=$ | $n \mid x \mid [e] \mid e + e \mid e - e \mid \ldots$ |
| Bool expr | $g, R$ | $::=$ | $b \mid e == e \mid e \neq e \mid e \geq e \mid \ldots$ |
| Commands | $c$ | $::=$ | $x = e \mid x = [e] \mid [e] = e \mid x = \mathsf{cons}(\overline{e}) \mid c; c$ |
| | | | $\mid \mathsf{if}\ g\ \mathsf{then}\ c\ \mathsf{else}\ c \mid \mathsf{while}\ g\ \{c\} \mid m(\overline{v}) \mid c \parallel c$ |
| | | | $\mid \mathsf{initLock}(e, K) \mid \mathsf{initCond}(e, e) \mid \mathsf{commit}\ e$ |
| | | | $\mid \mathsf{crhist}(e, R) \mid \mathsf{crfut}(e, R, M) \mid \mathsf{reinit}(e, R)$ |
| | | | $\mid \mathsf{sync}\ \langle a(\overline{v})\rangle\ (\overline{e})\{sb\} \mid \mathsf{assert}\ F$ |
| Sync. block | $sb$ | $::=$ | $sf; sc; sl$ |
| | $sf$ | $::=$ | $[] \mid \mathsf{wait}(g, e)$ |
| | $sl$ | $::=$ | $[] \mid \mathsf{notifyAll}(e)$ |
| | $sc$ | $::=$ | $x = e \mid x = [e] \mid [e] = e \mid x = \mathsf{cons}(\overline{e}) \mid sc; sc$ |
| | | | $\mid \mathsf{if}\ e\ \mathsf{then}\ sc\ \mathsf{else}\ sc$ |

Figure 6.2: Language syntax

## 6.3 Formalisation

This section formalises our approach. We use the same formalisation technique as in Section 4.3 and Section 5.5, but here adapted to a procedural language. This section presents the complete formalisation, but for a more detailed understanding of the underlying concepts, we refer to Section 4.3.

### 6.3.1 Language Syntax

Figure 6.2 presents the syntax of the language. With $\mathcal{L}$ we denote the set of lock labels. Values can be integers $n$, booleans $b$, permissions $\pi$, or process algebra terms $M$, i.e., abstract models. A program is composed of a set of methods (each provided by a contract), and program specifications: predicates, actions and processes. We use arithmetic expressions $e$ and boolean expressions $g, R$.

| | | | |
|---|---|---|---|
| Predicates | $pd$ | ::= | $\mathsf{pred}\ P = F(P \neq res\_inv) \mid \mathsf{pred}\ res\_inv_K = F$ |
| Actions | $act$ | ::= | $\mathsf{requires}\ F\ \mathsf{ensures}\ F\ \mathsf{action}\ a(\overline{x})$ |
| Processes | $proc$ | ::= | $p(\overline{x}) = M$ |
| Models | $M$ | ::= | $\epsilon \mid a(\overline{v}, g, x, x) \mid M \cdot M \mid M + M \mid M \parallel M$ $\mid M_1 \triangleleft g \triangleright M_2 \mid p(\overline{x})$ |
| Actions | $a(\overline{v}, g, x, x)$ | $\in$ | $\mathcal{A}$ |
| Formulas | $F$ | ::= | $g \mid F \oplus F \mid qt\ x.F \mid x \overset{\pi}{\mapsto} e \mid \mathsf{Lock}_K(e, \pi, b) \mid$ $\mathsf{Cond}_K(e, \pi) \mid \mathsf{Hist}(L, \pi, R, M) \mid \mathsf{Future}(L, \pi, R, M, b)$ |

Figure 6.3: Specification language

*Commands* With $[x]$ we denote pointer dereferencing. We distinguish three types of locations: *cell locations*, *locks*, and *conditions*. We use the command $\mathsf{cons}(e_1, ..., e_n)$ to allocate an array of cell locations on $n$ consecutive memory addresses, initialised with values $e_1, ..., e_n$. The command $\mathsf{initLock}(e, K)$ converts the cell location $e$ into a lock, attaching a lock label $K \in \mathcal{L}$ to it, while $\mathsf{initCond}(e_1, e_2)$ converts the cell location $e_1$ into a condition, associated to the lock $e_2$. A newly initialised lock is released with the command $\mathsf{commit}\ e$.

Commands $\mathsf{crhist}(e, R)$, $\mathsf{crfut}(e, R, M)$ and $\mathsf{reinit}(e, R)$ are specification commands for management of abstract models. The $c_1 \parallel c_2$ construct is used for parallel composition. A block of code $sb$ is synchronised on a lock $e$ with the command $\mathsf{sync}\ \langle a(v_1, ...v_n)\rangle\ (e)\{sb\}$ where $a(v_1, ...v_n)$ is an associated action, while $sb$ is the action implementation: $sb$ optionally starts and ends respectively with $\mathsf{wait}$ and $\mathsf{notifyAll}$ commands ([] means no command). With $sb^g$, $sb^w$ and $sb^n$ we denote respectively the guard, wait and notify location of the block $sb$.

*Specification formulas* Figure 6.3 illustrates the specification language. Special type of predicates are resource invariants. For every lock associated with a lock label $K$, a resource invariant $res\_inv_K$ should be defined; the default definition of the resource invariant is $res\_inv_K = true$. Actions defined in the program are specified with a contract, while processes are interpreted as a process algebra term $M$, composed of actions $a(\overline{v}, g, x, x)$. Predicates $\mathsf{Lock}_K(e, \pi, b)$ and $\mathsf{Cond}_K(e, \pi)$ respectively serve as a lock and a condition handle, while predicates $\mathsf{Hist}(L, \pi, R, M)$ and $\mathsf{Future}(L, \pi, R, M, b)$ represent respectively a history and a future over a set of locks $L$.

### 6.3.2 Language Semantics

*Program state* We model a state as triple:

$$\sigma \in \mathsf{State} = \mathsf{Heap} \times \mathsf{ThreadPool} \times \mathsf{ModelMap}$$

The first two components are used to express the semantics of the programming commands, while ModelMap is a *ghost* component: the specification commands operate on this component only. We store the state of the locks in the Heap component (locks and conditions are special types of locations) and avoid using a separate LockTable component (as was done in Chapters 4 and 5).

i) $h \in \mathsf{Heap}$ describes the shared memory, mapping every location to a *location value*. Locations are memory addresses, represented by integers, while location values vary depending on the type of the location:

$$\begin{array}{l} h \in \mathsf{Heap} = \mathsf{Loc} \rightharpoonup \mathsf{LocVal} \\ n \in \mathsf{Loc} \subseteq \mathsf{Value} \\ lv \in \mathsf{LocVal} = (\{\mathsf{Cell}\} \times \mathsf{Value}) \uplus (\{\mathsf{Lock}\} \times (\mathsf{ThrId} \uplus \{0\}) \times \mathcal{L}) \uplus (\{\mathsf{Cond}\} \times \mathsf{Value}) \\ t \in \mathsf{ThrId} = \mathsf{Value} \setminus \{0\} \end{array}$$

Therefore, given a location $n$:

- if $n$ is a *cell location* with a value $v$, $h(n) = (\mathsf{Cell}, v)$;
- if $n$ is a *lock*, $h(n) = (\mathsf{Lock}, t, K)$, where $t$ is the identifier of the owner thread or 0 if the lock is free, and $K$ is the associated lock label;
- if $n$ is a *condition*, $h(n) = (\mathsf{Cond}, n)$ where $n$ is the memory address of the associated lock.

ii) $tp \in \mathsf{ThreadPool}$ defines all threads operating on the heap, mapping thread identifiers to their local memory and a command to execute:

$$\begin{array}{l} tp \in \mathsf{ThreadPool} = \mathsf{ThrId} \rightharpoonup \mathsf{Stack}(\mathsf{Frame}) \times \mathsf{Cmd} \\ fr \in \mathsf{Frame} = \mathsf{Var} \rightharpoonup \mathsf{Value} \end{array}$$

iii) $mm \in \mathsf{ModelMap}$ describes the models that exist in the program state:

$$mm \in \mathsf{ModelMap} = \mathsf{Set}(\mathsf{Loc}) \rightharpoonup (\mathsf{InitHeap} \times \mathsf{Traces})$$

For every model $M$ over set of locks $L \in \mathsf{Set}(\mathsf{Loc})$, $mm$ stores:

- the *initial heap*, i.e., a mapping from all footprint locations of $L$ to their initial values:

$$ih \in \mathsf{InitHeap} = \mathsf{Loc} \rightharpoonup \mathsf{Value}$$
$$dom(mm(L)) = fp(L)$$

- *traces of actions*, i.e, the concrete (finite) sequence of actions recorded to the history if $M$ is a history; or a set of all possible (finite or infinite) sequences of execution ($\mathsf{TS}(M)$) if $M$ is a future. Every action in the trace is a tuple composed of: action identifier, action parameters, a boolean formula over cell locations (the guard), and wait and notify locations:

$$trace \in \mathsf{Traces} = (\{\mathsf{Hist}\} \times \overline{\mathsf{Action}}) \uplus (\{\mathsf{Future}\} \times \mathsf{Set}(\overline{\mathsf{Action}}))$$
$$act \in \mathsf{Action} = \mathsf{ActId} \times \overline{\mathsf{Value}} \times F \times (\mathsf{Loc} \uplus \{\mathsf{null}\}) \times (\mathsf{Loc} \uplus \{\mathsf{null}\})$$

**Operational semantics** The semantics of the commands is defined in terms of transitions $\sigma \rightsquigarrow \sigma'$. Figure 6.4 presents the meaning of the basic commands in the language. We discuss the more interesting cases:

- [*New*] : The command $x = \mathsf{cons}(n_0, ..., n_k)$ searches $k+1$ consecutive available addresses on the heap, allocates the value $n_i$ to the $i$-th location, and sets the variable $x$ to point to the first location.
- [*Call*] and [*Return*] : The call to a method extends the local stack with a new empty frame, executes the body of the method, followed by the command return, which removes the top frame of the stack.
- [*Parallel*] and [*Done*] : When two commands start to execute in parallel, the thread pool is extended with two new threads, each with a local empty memory. We use the command done to indicate the end of the thread execution and remove the thread from the pool.
- [*InitLock*] : This command converts the cell location $n$ to a lock, associating $K$ as its lock label and the current thread $t$ as an owner of the lock.
- [*InitCond*] : Converts the cell location $n_1$ to a condition associated to a lock $n_2$.

Figure 6.5 combines commands that are directly related to our history/future-based reasoning.

- [*CreateH*] and [*CreateF*] : Creating a new history extends the model map with a new empty trace, while a new future adds the set of all possible traces of the predicted future. Additionally, both commands copy the

$$
\begin{array}{ll}
[\mathit{VarSet}] & (h, tp.(t, fr \cdot s, x = n; c), mm) \rightsquigarrow (h, tp.(t, fr[x \mapsto n] \cdot s, c), mm) \\
[\mathit{Read}] & (h, tp.(t, fr \cdot s, x = [n]; c), mm) \rightsquigarrow \\
 & \quad h(n) = (\mathsf{Cell}, \_) \triangleright (h, tp.(t, fr[x \mapsto h(n)^2] \cdot s, c), mm) \\
[\mathit{Write}] & (h, tp.(t, s, [n] = v; c), mm) \rightsquigarrow \\
 & \quad h(n) = (\mathsf{Cell}, \_) \triangleright (h[n \mapsto (\mathsf{Cell}, v)], tp.(t, s, c), mm) \\
[\mathit{New}] & (h, tp.(t, fr \cdot s \ \mathsf{in} \ x = \mathsf{cons}(n_0, ..., n_k); c), mm) \rightsquigarrow \\
 & \quad (h[(n+i) \mapsto (\mathsf{Cell}, n_i)]_{\forall i=1..k}, tp.(t, fr[x \mapsto n] \cdot s, c), mm), \\
 & \quad \text{where} \quad \forall i = 0..k.n + i \cap dom(h) = \emptyset \\
 & \\
[\mathit{If}] & (h, tp.(t, s, \mathsf{if}\, b\, \mathsf{then}\, c_1\, \mathsf{else}\, c_2; c), mm) \rightsquigarrow \\
 & \quad (h, tp.(t, s, c'; c), mm), \text{where } b \Rightarrow c' = c_1 \text{ else } c' = c_2 \\
[\mathit{While}] & (h, tp.(t, s, \mathsf{while}\ g\ \{c_1\}; c), mm) \rightsquigarrow (h, tp.(t, s, c'), mm), \text{where} \\
 & \quad \text{if } [\![g]\!]^h_s = \mathsf{true} \Rightarrow c' = c_1; \mathsf{while}\ g\ \{c_1\} \\
 & \quad \text{if } [\![g]\!]^h_s = \mathsf{false} \Rightarrow c' = c \\
 & \\
[\mathit{Call}] & (h, tp.(t, s, m(\overline{v}); c), mm) \rightsquigarrow (h, tp.(t, \emptyset \cdot s, c_m[\overline{v}/\overline{x}]; \mathsf{return}; c), mm), \\
 & \quad \text{where } m(\overline{x}) = c_m \\
[\mathit{Return}] & (h, tp.(t, fr \cdot s, \mathsf{return}; c), mm) \rightsquigarrow (h, tp.(t, s, c), mm) \\
 & \\
[\mathit{Parallel}] & \dfrac{t_1, t_2 \notin dom(tp), dom(tp') \quad (h, tp.(t_1, \emptyset, c_1; \mathsf{done}).(t_2, \emptyset, c_2; \mathsf{done}), mm) \rightsquigarrow^\star (h', tp', mm');}{(h, tp.(t, s, c_1 \parallel c_2; c), mm) \rightsquigarrow^\star (h', tp'.(t, s, c), mm');} \\
[\mathit{Done}] & (h, tp.(t, s, \mathsf{done}), mm) \rightsquigarrow (h, tp, mm) \\
 & \\
[\mathit{InitLock}] & (h, tp.(t, s, \mathsf{initLock}(n, K); c), mm) \rightsquigarrow \\
 & \quad h(n) = (\mathsf{Cell}, \_) \triangleright (h[n \mapsto (\mathsf{Lock}, t, K)], tp.(t, s, c), mm) \\
[\mathit{InitCond}] & (h, tp.(t, s, \mathsf{initCond}(n_1, n_2); c), mm) \rightsquigarrow h(n_1) = (\mathsf{Cell}, \_) \wedge \\
 & \quad h(n_2) = (\mathsf{Lock}, \_, \_) \triangleright (h[n_1 \mapsto (\mathsf{Cond}, n_2)], tp.(t, s, c), mm) \\
[\mathit{Commit}] & (h, tp.(t, s, \mathsf{commit}\ n; c), mm) \rightsquigarrow \\
 & \quad h(n) = (\mathsf{Lock}, t, K) \triangleright (h[n \mapsto (\mathsf{Lock}, 0, K)], tp.(t, s, c), mm)
\end{array}
$$

Figure 6.4: Operational semantics of basic commands

$$
\begin{array}{ll}
[CreateH] & (h, tp.(t, s, \mathsf{crhist}(L, R); c), mm) \rightsquigarrow (h, tp.(t, s, c), mm[L \mapsto (ih, (\mathsf{Hist}, \mathsf{nil}))]) \\
& \quad \text{where } dom(ih) = fp(L) \wedge \forall n \in dom(ih).ih(n) = h(n) \\
[CreateF] & (h, tp.(t, s, \mathsf{crfut}(L, R, M); c), mm) \rightsquigarrow \\
& \quad (h, tp.(t, s, c), mm[L \mapsto (ih, (\mathsf{Future}, \mathsf{TS}(M)))]) \\
& \quad \text{where } dom(ih) = fp(L) \ \forall l \in dom(ih).ih(l) = h(l) \\
[Reinit] & (h, tp.(t, s, \mathsf{reinit}(L, R); c), mm) \rightsquigarrow (h, tp.(t, s, c), mm[L \mapsto (ih, (\mathsf{Hist}, \mathsf{nil}))]) \\
& \quad dom(ih) = fp(L) \ \forall l \in dom(ih).ih(l) = h(l) \\
\\
[Wait] & (h, tp.(t, s, \mathsf{wait}(g, n); c), mm) \rightsquigarrow (h', tp.(t, s, c'), mm), \text{ where} \\
& \quad [\![g]\!]^h_s = \mathsf{true} \Rightarrow h' = h \wedge c' = c \\
& \quad [\![g]\!]^h_s = \mathsf{false} \Rightarrow h' = h[n_m \mapsto (\mathsf{Lock}, 0, K_t)] \wedge \\
& \qquad\qquad c' = \mathsf{waiting}(n); \mathsf{resume}(n_l); \mathsf{wait}(g, n); c \\
& \quad h(n) = (\mathsf{Lock}, \_, \_) \Rightarrow n_l = n \qquad h(n) = (\mathsf{Cond}, n') \Rightarrow n_l = n' \\
[Resume] & (h, tp.(t, s, \mathsf{resume}(n); c), mm) \rightsquigarrow \\
& \quad (h[n \mapsto (\mathsf{Lock}, t, K)], tp.(t, s, c), mm) \ \text{ if } h(n) = (\mathsf{Lock}, 0, K) \\
[Notify] & (h, tp.(t, s, \mathsf{notifyAll}(n); c).(t_1, s_1, \mathsf{waiting}(n); c_1)\ldots. \\
& .(t_k, s_k, \mathsf{waiting}(n); c_k), mm) \rightsquigarrow (h, tp.(t, s, c).(t_1, s_1, c_1)..(t_k, s_k, c_k), mm) \\
\\
[Sync] & (h, tp.(t, s, \mathsf{sync}\ \langle a(\overline{v})\rangle\ (n)\{sb\}; c), mm) \rightsquigarrow \\
& \quad (h[n \mapsto (\mathsf{Lock}, t, K)], tp.(t, s, sb; \mathsf{endsync}(a(\overline{v}), n, sb); c), mm) \\
& \quad \text{if } h(n) = (\mathsf{Lock}, 0, K) \\
[EndSync] & (h, tp.(t, s, \mathsf{endsync}(a(\overline{v}), n, sb); c), mm) \rightsquigarrow (h[n \mapsto (0, t, K)], tp.(t, s, c), mm') \\
& \quad \text{where } n \in L \wedge L \in dom(mm) \quad A_1 = (a, \overline{v}, sb^g, sb^w, sb^n) \\
& \quad \text{if } mm(L) = (\mathsf{Hist}, \overline{A}) \Rightarrow mm' = mm[L \mapsto (\mathsf{Hist}, A_1{+\!+}\overline{A})] \\
& \quad \text{if } mm(L) = (\mathsf{Future}, S) \Rightarrow mm' = mm[L \mapsto (\mathsf{Future}, S')] \\
& \quad S' = \{\overline{A} | A_1{+\!+}\overline{A} \in S\}
\end{array}
$$

Figure 6.5: Operational semantics of guarded blocks-related commands

values of the locations from the heap to the initial heap.

- [*Reinit*] : Reinitialisation is possible only for histories: the semantics of reinitialisation of a history is equivalent to the semantics for creating a history.
- [*Wait*] : To model the $\mathsf{wait}(g, e)$ command, we introduce two special commands: i) $\mathsf{waiting}(e)$ describes that the thread has tried to execute $\mathsf{wait}(g, e)$ when the guard has been false and is now waiting for a notification on $e$;

and ii) resume($e$) means that the waiting thread has been notified and is now competing for the lock $e$.

- [*Resume*] : The semantics of the resume($e$) command is equivalent as acquiring a lock $e$.
- [*Notify*] : When a thread executes a notifyAll($e$) command, all threads that are waiting move into a *resume* state.
- [*Sync*] : and [*EndSync*] : Both commands describe the execution of a synchronised block (we use endsync to indicate the end of the block). When entering the block, the current thread obtains the lock and thereafter, executes the body of the block. At the end of the block the lock is released and the model map is updated: in case of a history, a new representative action $A_1$ is added; in case of a future, all traces that appeared to be a wrong prediction (those that do not start with $A_1$) are removed from the set, while the initial action $A_1$ is removed from the remaining traces.

*Initial program state* The initial global program state of a program *pgm* with a main method $c_{main}$ is defined as:

$$\sigma_0 = (\emptyset, (t_{main}, \emptyset, c_{main}), \emptyset)$$

The thread pool contains the main thread only $t_{main}$ with an empty local stack and the code $c_{main}$ that needs to execute. The heap and the model map are empty.

**Non-blocking of a program** Having defined the operational semantics of our programming language, now we are ready to give a definition for *non-blocking* of a program.

We say that *a thread $t$ is waiting on $x$* if $t$ is in the prestate of wait $(g, x)$. If $t$ is in a poststate of wait $(g, x)$, we say that *$t$ has passed $x$*. We define a *maximal fair run* as an infinite run under fair scheduling, or as a finite run ending in a final state from which no transition is enabled.

**Definition 6.1.** *A program is* non-blocking with respect to a set of locks $L$ *if: for any maximal fair run, if in a state $\sigma$ some thread is waiting on $x$, where $x \in L$ or $x$ is a condition associated to a lock from $L$, then in a later state $\sigma'$ some thread has passed $x$.*

Note that Definition 6.1 does not ensure that the same thread that is waiting on $x$ will pass $x$. If in the example in Listing 6.5, there were one infinite producer $p$ and two infinite consumers $c_1$ and $c_2$, the program is non-blocking even if only

$p$ and $c_1$ were active (while $c_2$ waits forever on $nE$). In that case, $c_2$ will be obtaining the lock (the run is fair), but not in a state where the related guard holds. Importantly, while in programs with infinite executions, non-blocking guarantees *global progress*, in programs with finite executions only, it guarantees *termination*.

**Resources** We do not reason directly on the global program state, but on a resource $\mathcal{R}$, i.e., a *partial* abstraction of the state. Later, we express the semantics of formulas over a given resource. The concepts of using resources is described more extensively in Section 4.3.

We define a resource as a tuple composed of an *abstract heap* $\mathcal{H}$ and an *abstract model map* $\mathcal{M}$:

$$\mathcal{R} = (\mathcal{H}, \mathcal{M})$$

For a resource $\mathcal{R}$ that is an abstraction of the global state $\sigma$, the relation $\mathit{abstracts}(\sigma, \mathcal{R})$ holds (*see* Apendix D for the formal definition).

The *compatibility relation* and the *joining operator* between two resources is defined component-wise:

$$\begin{array}{lcl} (\mathcal{H}_1, \mathcal{M}_1)\#(\mathcal{H}_2, \mathcal{M}_2) & \Leftrightarrow & \mathcal{H}_1\#\mathcal{H}_2 \wedge \mathcal{M}_1\#\mathcal{M}_2 \\ (\mathcal{H}_1, \mathcal{M}_1)*(\mathcal{H}_2, \mathcal{M}_2) & = & (\mathcal{H}_1*\mathcal{H}_2, \mathcal{M}_1*\mathcal{M}_2) \end{array}$$

*Abstract heap* The component $\mathcal{H}$ represents an abstraction of the global heap; it is a *masked heap*: every heap location is associated with a fraction. This fraction defines the permission to the location owned by the resource (for cell locations), or the fraction of the $\mathsf{Lock}$ or $\mathsf{Cond}$ predicates (for locks or conditions respectively). Formally, an abstract heap is represented as:

$$\mathcal{H} \in \mathsf{Loc} \rightharpoonup \mathsf{LocVal} \times [0, 1]$$

Furthermore, two abstract heaps are compatible when they map each location to the same value, and the sum of the permissions (fractions) associated to a location does not exceed 1:

$$\begin{array}{lcl} \mathcal{H}_1\#\mathcal{H}_2 & \Leftrightarrow & \mathit{dom}(\mathcal{H}_1) = \mathit{dom}(\mathcal{H}_2) \wedge \\ & & \forall n \in \mathit{dom}(\mathcal{H}_1).\mathcal{H}_1(n)^1 = \mathcal{H}_2(n)^1 \wedge \mathcal{H}_1(n)^2 + \mathcal{H}_2(n)^2 \leq 1 \\ \mathcal{H}_1*\mathcal{H}_2 & = & \lambda n.(\mathcal{H}_1(n)^1, \mathcal{H}_1(n)^2 + \mathcal{H}_2(n)^2) \end{array}$$

*Abstract model map* The second component represents a *masked model map*: every action from the global model map is marked with a boolean flag in-

dicating whether the action is owned by the resource, while the value $[0,1]$ is the fraction of the history/future predicate owned by the resource. For futures, an additional boolean value is stored, defining whether the model is newly created. Formally, we define an abstract model map as:

$$\mathcal{M} \in \mathsf{Set}(\mathsf{Loc}) \rightharpoonup \mathsf{InitHeap} \times (\mathsf{Hist}(\overline{\mathcal{A} \times \mathsf{bool}}) \cup \mathsf{Future}(\mathsf{Set}(\overline{\mathcal{A} \times \mathsf{bool}}) \times \mathsf{bool})) \times [0,1]$$

The definition of the compatibility and joining operator describes that: two history maps are compatible when for every model they keep the same initial heap; the same action does not occur in both resources and the sum of the associated fraction does not exceed 1. Furthermore, if the fraction for a given model in the resource is 0, then no action from that model belongs to that resource (this is expressed via the *soundness of an abstract model map*, $\mathcal{M} : \diamond$). Formally:

$$\begin{array}{rcl}
\mathcal{M} : \diamond & \Leftrightarrow & \forall L \in \mathit{dom}(\mathcal{M}).\mathcal{M}(L)^3 = 0 \Rightarrow \\
& & (\mathcal{M}(L)^2 = (\mathsf{Hist}, S) \Rightarrow (_, \mathsf{true}) \notin S) \wedge \\
& & (\mathcal{M}(L)^2 = (\mathsf{Future}, SS, b) \Rightarrow \forall S \in SS.(_, \mathsf{true}) \notin S) \\
\mathcal{M}_1 \# \mathcal{M}_2 & \Leftrightarrow & \mathcal{M}_1 : \diamond \wedge \mathcal{M}_2 : \diamond \wedge \\
& & \mathit{dom}(\mathcal{M}_1) = \mathit{dom}(\mathcal{M}_2) \wedge \\
& & \forall L \in \mathit{dom}(\mathcal{M}_1). \\
& & \quad \mathcal{M}_1(L)^1 = \mathcal{M}_2(L)^1 \wedge \mathcal{M}_1(L)^3 + \mathcal{M}_2(L)^3 \leq 1 \wedge \\
& & \quad \mathcal{M}_1(L)^2 = (\mathsf{Hist}, S_1) \Rightarrow (\mathcal{M}_2(L)^2 = (\mathsf{Hist}, S_2) \wedge S_1 \# S_2) \wedge \\
& & \quad \mathcal{M}_1(L)^2 = (\mathsf{Future}, SS_1, b) \Rightarrow (\mathcal{M}_2(L)^2 = (\mathsf{Future}, SS_2, b) \wedge \\
& & \qquad\qquad SS_1 \# SS_2) \\
S_1 \# S_2 & = & |S_1| = |S_2| \wedge \forall i. S_1\,[i]^1 = S_2\,[i]^1 \wedge \neg(S_1\,[i]^2 \wedge S_2\,[i]^2) \\
SS_1 \# SS_2 & = & |SS_1| = |SS_2| \wedge \forall S_1 \in SS_1 \exists. S_2 \in SS_2. S_1 \# S_2 \\
\\
(\mathcal{M}_1 * \mathcal{M}_2) & = & \lambda L.(\mathcal{M}_1(L), \mathcal{M}_1(L)^2 * \mathcal{M}_2(L)^2, \mathcal{M}_1(L)^3 + \mathcal{M}_2(L)^3) \\
& & (\mathsf{Hist}, S_1) * (\mathsf{Hist}, S_2) = (\mathsf{Hist}, S_1 * S_2) \\
& & (\mathsf{Future}, SS_1, b_1) * (\mathsf{Future}, SS_2, b_2) = (\mathsf{Future}, SS_1 * SS_2, b_1 \vee b_2) \\
S_1 * S_2 & = & S, |S| = |S_1| \wedge \forall i. S[i]^1 = S_1[i]^1 \wedge S[i]^2 = S_1[i]^2 \vee S_2[i]^2 \\
SS_1 * SS_2 & = & \{SS \mid \ |SS| = |SS_1| \wedge \exists S_1 \in SS_1, \exists S_2 \in SS_2. S = S_1 * S_2\}
\end{array}$$

$$\text{where } S : \overline{\mathsf{Action} \times \mathsf{bool}}, \quad SS : \mathsf{Set}(\overline{\mathsf{Action} \times \mathsf{bool}})$$

**Semantics of formulas** Figure 6.6 shows the semantics of formulas, where the relation $\mathcal{R}; s \models F$ denotes that the formula $F$ is valid with respect to a stack $s$ and a resource $\mathcal{R}$.

$$
\begin{array}{rlcl}
[Exp] & \mathcal{R}; s \models g & \Leftrightarrow & [\![g]\!]_s^{\mathcal{H}} = \mathsf{true} \\
[Conj] & \mathcal{R}; s \models F \wedge G & \Leftrightarrow & \mathcal{R}; s \models F \wedge \mathcal{R}; s \models G \\
[Disj] & \mathcal{R}; s \models F \vee G & \Leftrightarrow & \mathcal{R}; s \models F \vee \mathcal{R}; s \models G \\
[Forall] & \mathcal{R}; s \models \forall x.F & \Leftrightarrow & \forall v.\mathcal{R}; s \models F[v/x] \\
[Exists] & \mathcal{R}; s \models \exists x.F & \Leftrightarrow & \exists v.\mathcal{R}; s \models F[v/x] \\
[SepConj] & \mathcal{R}; s \models F{*}G & \Leftrightarrow & \exists \mathcal{R}_1, \mathcal{R}_2.\mathcal{R} = \mathcal{R}_1{*}\mathcal{R}_2 \wedge \\
 & & & \mathcal{R}_1; s \models F \wedge \mathcal{R}_2; s \models G \\
[Perm] & \mathcal{R}; s \models e \overset{\pi}{\mapsto} e' & \Leftrightarrow & \mathcal{H}([\![e]\!]_s) = (\mathsf{Cell}, [\![e']\!]_s, \pi'), \pi' \geq \pi \\
[Lock] & \mathcal{R}; s \models \mathsf{Lock}_K(e, \pi, b) & \Leftrightarrow & \mathcal{H}([\![e]\!]_s) = (\mathsf{Lock}, \_, K, \pi'), \pi' \geq \pi \\
[Cond] & \mathcal{R}; s \models \mathsf{Cond}_K(e, \pi) & \Leftrightarrow & \mathcal{H}([\![e]\!]_s) = (\mathsf{Cond}, n, \pi'), \pi' \geq \pi \wedge \\
 & & & \mathcal{H}(n) = (\mathsf{Lock}, \_, K, \_)
\end{array}
$$

$$
\begin{array}{rl}
[Hist] & \mathcal{R}; s \models \mathsf{Hist}(\{x_1, .., x_k\}, \pi, R, M) \Leftrightarrow \\
 & \mathcal{M}(\{[\![x_1]\!]_s, .., [\![x_k]\!]_s\}) = (h_i, (\mathsf{Hist}, S), \pi') \wedge \pi' \geq \pi \wedge \\
 & [\![R]\!]_s^{h_i} = \mathsf{true} \wedge \mathsf{filter}(S) \in [\![TS(M)]\!]_s \\
[Future] & \mathcal{R}; s \models \mathsf{Future}(\{x_1, .., x_k\}, \pi, R, M, b) \Leftrightarrow \\
 & \mathcal{M}(\{[\![x_1]\!]_s, .., [\![x_k]\!]_s\}) = (h_i, \mathsf{Future}(\{S_1, ...S_n\}, [\![b]\!]_s), \pi') \wedge \pi' \geq \pi \wedge \\
 & [\![R]\!]_s^{h_i} = \mathsf{true} \wedge \bigcup_{i=1..n} \mathsf{filter}(S_i) = [\![TS(M)]\!]_s
\end{array}
$$

Figure 6.6: Semantics of formulas, $\mathcal{R} = (\mathcal{H}, \mathcal{M})$

The semantics of most of the formulas is clear from the definition of a resource. The predicate $\mathsf{Hist}(L, \pi, R, H)$ (and similarly $\mathsf{Future}(L, \pi, R, H, b)$) is valid when the abstract model map $\mathcal{M}$ contains a mapping for the set of locks $L$ such that: the fraction in the mapping is at least $\pi$, the predicate $R$ holds over the values stored in the initial heap; and the model $M$ is consistent with the model viewed by the resource: in case of a history, $\mathcal{M}$ contains a trace that belongs to the set of traces of $M$; in case of a future, $\mathcal{M}$ contains the whole set of traces of $M$. The function $\mathsf{filter}(S)$ returns the subsequence of $S$ ordered in the same order, with actions marked with the flag $\mathsf{true}$.

### 6.3.3 Proof System

As explained in Section 4.3.3, our proof system consists of: i) *proof theory*, i.e., *logical consequence rules* represented in the form $F \vdash F'$ and *axioms* in the form $\vdash F$; and ii) *Hoare triples*, i.e., *axioms* and *inference rules* that describe the

| | |
|---|---|
| [*SplitMerge Perm*] | $e \overset{\pi_1+\pi_2}{\mapsto} v \mathrel{*\!\!-\!\!*} e \overset{\pi_1}{\mapsto} v * e \overset{\pi_2}{\mapsto} v$ |
| [*SplitMerge Lock*] | $\mathsf{Lock}_K(x, \pi_1 + \pi_2, b) \mathrel{*\!\!-\!\!*} \mathsf{Lock}_K(x, \pi_1, b) * \mathsf{Lock}_K(x, \pi_2, b)$ |
| [*SplitMerge Cond*] | $\mathsf{Cond}_K(x, \pi_1 + \pi_2) \mathrel{*\!\!-\!\!*} \mathsf{Cond}_K(x, \pi_1) * \mathsf{Cond}_K(x, \pi_2)$ |
| [*SplitMerge Hist*] | $\mathsf{Hist}(L, (\pi_1 + \pi_2), R, M_1 \parallel M_2) \mathrel{*\!\!-\!\!*}$ $\mathsf{Hist}(L, \pi_1, R, M_1) * \mathsf{Hist}(L, \pi_2, R, M_2)$ |
| [*SplitMerge Future*] | $\mathsf{Future}(L, (\pi_1 + \pi_2), R, M_1 \parallel M_2, b) \mathrel{*\!\!-\!\!*}$ $\mathsf{Future}(L, \pi_1, R, M_1, \mathsf{true}) * \mathsf{Future}(L, \pi_2, R, M_2, \mathsf{true})$ |

Figure 6.7: Axioms from the proof system

behaviour of the language commands.

*Proof theory* Figure 6.7 gives a set of axioms that illustrate splitting and merging of the predicates in our language (the group predicates). The other rules in the proof theory are standard and are omitted in this presentation.

*Hoare triples* Figure 6.8 contains the complete set of Hoare triples. We describe the most interesting cases:

- [*InitLock*] : Converting a location $x$ to a lock consumes the write permission to the location $x$ and produces a predicate $\mathsf{Lock}_K(x, 1, \mathsf{false})$; the parameter $\mathsf{false}$ indicates that there exists no model that refers to the lock $x$.
- [*InitCond*] : Similarly, initialising a new condition $x$ converts a write permission to $x$ to a predicate $\mathsf{Cond}_K(x, 1)$.
- [*Commit*] : Committing a lock consumes the resource invariant associated to the lock.
- [*CreateH*] and [*CreateF*]: To create a history/future over a set $L$, every $x \in L$ must be a lock for which no model already exists: $\mathsf{Lock}_K(x, 1, \mathsf{false})$ is required and converted to $\mathsf{Lock}_K(x, 1, \mathsf{true})$.
- [*MergeH*]: When threads join, their local histories are merged and added to the history of the receiver thread.
- [*Reinit*]: When the history predicate is complete and the model that it currently stores terminates from $R$ to a $R'$, the history may be emptied, setting the predicate $R'$ as a new initial property.

- [*Wait*], [*Notify*] and [*Sync*]: The rules [*Wait*] and [*Notify*] describe only partial correctness and state nothing about blocking of the program. Instead, the information about blocking is captured by an abstract action and the [*Sync*] rule ensures that this action is part of the abstract model. Reasoning about non-blocking is then performed on the model in the moment when the model is complete; non-blocking of the model guarantees non-blocking of the original program as shown later in Theorem 6.2.

### 6.3.4 Reasoning about the Abstract Model

We define the semantics of process algebra terms $M$ as a *labelled transition system* (LTS) over states

$$s = (h, W, M) \quad \text{where:}$$

i) $h \in \mathsf{Var} \rightharpoonup \mathsf{Value}$ is a *heap* mapping state variables to values;

ii) $W \in \mathsf{Var} \rightharpoonup \mathsf{Set}(\mathsf{ActId})$ is a *waiting room* mapping variables (which represent waiting locations) to a set of action identifiers (which represent the actions waiting on each location);

iii) $M$ is the process algebra term composed of actions $a(\overline{v}, a_g, a_w, a_n)$. We require each action to have a unique identifier.

We call the state $(h, \emptyset, \epsilon)$ a *final state*. The transitions of the LTS are written in the form $s \xrightarrow{l} s'$, where $l \in \mathsf{ActId} \cup \{\tau\}$. Figure 6.9 presents the transition rules. We describe the first two rules, while the semantics of the others is standard.

- [*Fail*] : The rule produces a silent transition $\tau$ when the guard $a_g$ is not satisfied, while moving the action to the waiting room.
- [*Success*] : Executes the action $a$, if $a_g$ holds on the heap $h$; additionally, if the action contains a notify location $a_n$, all actions waiting on $a_n$ are removed from the waiting room $W$ (note that $x \notin W$ entails $W(x) = \emptyset$). The $\mathcal{E}[\![\mathtt{h}]\!]\mathtt{a}$ function evaluates the action $a$ on the heap $h$ (note that $a$ is defined as an abstract method with a contract).

**Non-blocking of abstract models** Having defined the semantics of abstract models, we are now ready to give a formal definition for *non-blocking of abstract models*.

**Definition 6.2.** *Let $M$ be an abstract model and $R$ be a boolean formula. We say that:*

$[VarSet]$ $$\frac{}{\vdash \{\mathsf{true}\}\ y = x\ \{y = x\}}$$

$[Read]$ $$\frac{}{\vdash \{x \overset{\pi}{\mapsto} v\}\ y = [x]\ \{x \overset{\pi}{\mapsto} v * y = v\}}$$

$[Write]$ $$\frac{}{\vdash \{x \overset{1}{\mapsto} \_\}\ [x] = v\ \{x \overset{1}{\mapsto} v\}}$$

$[New]$ $$\frac{}{\vdash \{\mathsf{true}\}\ x = \mathsf{cons}(v_0, ..., v_n)\ \{\circledast_{\forall i=0..n} x + i \overset{1}{\mapsto} v_i\}}$$

$[Sequence]$ $$\frac{\vdash \{F_1\}\ c_1\ \{F_2\} \quad \{F_2\}\ c_2\ \{F_3\}}{\vdash \{F_1\}\ c_1; c_2\ \{F_3\}}$$

$[If]$ $$\frac{\vdash \{F * b\}\ c\ \{G\} \quad \vdash \{F * \neg b\}\ c'\ \{G\}}{\vdash \{F\}\ \mathsf{if}\ b\ \mathsf{then}\ c\ \mathsf{else}\ c'\ \{G\}}$$

$[While]$ $$\frac{\vdash \{F * b\}\ c\ \{F\}}{\{F\}\ \mathsf{while}\ b\ \{c\}\ \{\neg b * F\}}$$

$[Call]$ $$\frac{\mathsf{requires}\ F\ \mathsf{ensures}\ F'\ m(\overline{x})\{c\} \quad \vdash \{F\}\ c[\overline{v}/\overline{x}]\ \{F'\}}{\vdash \{F\}\ m(\overline{v})\ \{F'\}}$$

$[Parallel]$ $$\frac{\vdash \{P_1\}\ c_1\ \{Q_1\} \quad \vdash \{P_2\}\ c_2\ \{Q_2\}}{\vdash \{P_1 * P_2\}\ c_1 \parallel c_2\ \{Q_1 * Q_2\}}$$

$[InitLock]$ $$\frac{}{\vdash \{x \overset{1}{\mapsto} \_\}\ \mathsf{initLock}(x, K)\ \{\mathsf{Lock}_K(x, 1, \mathsf{false})\}}$$

$[InitCond]$ $$\frac{}{\vdash \{x \overset{1}{\mapsto} \_ * \mathsf{Lock}_K(y, \pi, b)\}\ \mathsf{initCond}(x, y)\ \{\mathsf{Cond}_K(x, 1) * \mathsf{Lock}_K(y, \pi, b)\}}$$

$[Commit]$ $$\frac{}{\vdash \{\mathsf{Lock}_K(x, 1, b) * I(K)\}\ \mathsf{commit}\ x\ \{\mathsf{Lock}_K(x, 1, b)\}}$$

$$[CreateH]\quad \frac{fp(R) \subseteq fp(L)}{\begin{array}{c}\{\circledast_{\forall x \in L}\mathsf{Lock}_K(x,1,\mathsf{false}) * I(K) * R\} \\ \vdash \quad \mathsf{crhist}(L,R) \\ \{\circledast_{\forall x \in L}\mathsf{Lock}_K(x,1,\mathsf{true}) * I(K) * \mathsf{Hist}(L,1,R,\epsilon)\}\end{array}}$$

$$[CreateF]\quad \frac{fp(R) \subseteq fp(L)}{\begin{array}{c}\{\circledast_{\forall x \in L}(\mathsf{Lock}_K(x,1,\mathsf{false}) * I(K)) * R\} \\ \vdash \quad \mathsf{crfut}(L,R,M) \\ \{\circledast_{\forall x \in L}\mathsf{Lock}_K(x,1,\mathsf{true}) * I(K) * \mathsf{Future}(L,1,R,M,\mathsf{true})\}\end{array}}$$

$$[MergeH]\quad \frac{\begin{array}{c}\vdash \{\mathsf{Hist}(L,\pi_1,R,\epsilon)\}c_1\{\mathsf{Hist}(L,\pi_1,R,M_1)\} \\ \vdash \{\mathsf{Hist}(L,\pi_2,R,\epsilon)\}c_2\{\mathsf{Hist}(L,\pi_2,R,M_2)\}\end{array}}{\vdash \{\mathsf{Hist}(L,\pi_1+\pi_2,R,M)\}c_1 \parallel c_2\{\mathsf{Hist}(L,\pi_1+\pi_2,R,M\cdot(M_1 \parallel M_2))\}}$$

$$[Reinit]\quad \frac{M{\downarrow_{R,R'}}}{\vdash \{\mathsf{Hist}(L,1,R,M)\}\ \mathsf{reinit}(L,R')\ \{\mathsf{Hist}(L,1,R',\epsilon)\}}$$

$$[Wait]\quad \vdash \{\mathsf{Cond}_K(x,\pi) * I(K)\}\ \mathsf{wait}(g,x)\ \{\mathsf{Cond}_K(x,\pi) * I(K) * g\}$$

$$[Notify]\quad \vdash \{\mathsf{Cond}_K(x,\pi)\}\ \mathsf{notifyAll}(x)\ \{\mathsf{Cond}_K(x,\pi)\}$$

$$[Sync]\quad \frac{\begin{array}{c}act ::= \mathsf{requires}\ F\ \mathsf{ensures}\ F'\ \mathsf{action}\ a(\overline{y}) \quad x \subseteq L \quad \sigma = (\overline{w}/\overline{y}) \\ fp(F,F',g) \subseteq fp(x) \quad A = a(\overline{w}, sb^g, sb^w, sb^n) \\ P_1 = \mathsf{Hist}(L,\pi_1,R,M)\ \text{and}\ P_2 = \mathsf{Hist}(L,\pi_1,R,M\cdot A)\ \text{or} \\ P_1 = \mathsf{Future}(L,\pi_1,R,A\cdot M,\_)\ \text{and}\ P_2 = \mathsf{Future}(L,\pi_1,R,M,\mathsf{false}) \\ \vdash \{I_K * F[\sigma]\}sb\{I_K * F'[\sigma]\}\end{array}}{\begin{array}{c}\{\mathsf{Lock}_K(x,\pi,\mathsf{true}) * F[\sigma] * P_1\} \\ \vdash \mathsf{sync}\,\langle a(\overline{w})\rangle\ (x)\{sb\} \\ \{\mathsf{Lock}_K(x,\pi,\mathsf{true}) * F'[\sigma] * P_2\}\end{array}}$$

$$[Conseq]\quad \frac{\vdash F_1 \Rightarrow F_1' \quad \vdash \{F_1'\}\ c\ \{F_2'\} \quad F_2' \Rightarrow F_2}{\vdash \{F_1\}\ c\ \{F_2\}}$$

$$[Frame]\quad \frac{\vdash \{F_1\}c\{F_2\} \quad freevars(F_3) \cap writes(c) = \emptyset}{\vdash \{F_1 * F_3\}c\{F_2 * F_3\}}$$

Figure 6.8: Hoare triples

$$[\mathit{Fail}] \quad \frac{h \models \neg a_g \quad a \notin W(a_w) \quad W' = W[a_w \mapsto W(a_w) \cup \{a\}]}{\langle h, W, a(\overline{v}, a_g, a_w, a_n)\rangle \xrightarrow{\tau} \langle h, W', a(\overline{v}, a_g, a_w, a_n)\rangle}$$

$$[\mathit{Success}] \quad \frac{h \models a_g \quad a \notin W(a_w) \quad h' \in \mathcal{E}[\![\mathtt{h}]\!]\mathtt{a} \quad W' = \begin{cases} W[a_n \mapsto \emptyset] & \text{if } a_n \neq \mathsf{null} \\ W & \text{otherwise} \end{cases}}{\langle h, W, a(\overline{v}, a_g, a_w, a_n)\rangle \xrightarrow{a} \langle h', W', \epsilon\rangle}$$

$$[\mathit{Alt}] \quad \frac{\langle h, W, M_i\rangle \xrightarrow{l} \langle h', W', M_i'\rangle \quad i \in \{1,2\}}{\langle h, W, M_1 + M_2\rangle \xrightarrow{l} \langle h', W', M_i'\rangle}$$

$$[\mathit{Par}] \quad \frac{\langle h, W, M_i\rangle \xrightarrow{l} \langle h', W', M_i'\rangle \quad i,j \in \{1,2\}\ i \neq j}{\langle h, W, M_1 \parallel M_2\rangle \xrightarrow{l} \langle h', W', M_i' \parallel M_j\rangle}$$

$$[\mathit{Seq}] \quad \frac{\langle h, W, M_1\rangle \xrightarrow{l} \langle h', W', M_1'\rangle}{\langle h, W, M_1 \cdot M_2\rangle \xrightarrow{l} \langle h', W', M_1' \cdot M_2\rangle}$$

$$[\mathit{If}] \quad \frac{\langle h, W, M_i\rangle \xrightarrow{l} \langle h', W', M_i'\rangle \quad i = \begin{cases} 1 & \text{if } h \models b \\ 2 & \text{otherwise} \end{cases}}{\langle h, W, M_1 \triangleleft b \triangleright M_2\rangle \xrightarrow{l} \langle h', W', M_i'\rangle}$$

Figure 6.9: Transition rules for abstract models

*i)* $M$ terminates *from* $R$ *in* $R'$*, written* $M{\downarrow_{R,R'}}$*, if every path starting from an initial state* $s_0 = (h_0, \emptyset, M)$ *such that* $h_0 \models R$ *ends in a final state* $s_{fin} = (h_{fin}, \emptyset, \epsilon)$ *such that* $h_{fin} \models R'$*.*

*ii)* $M$ *is* non-blocking *from* $R$*, written* $M{\downarrow_R}$*, if for every path starting from an initial state* $s_0 = (h_0, \emptyset, M)$ *such that* $h_0 \models R$ *it holds: if* $s_1 = (h', W', M')$ *is a state reachable from* $s_0$ *(*$s_0 \rightarrow^* s_1$*) and* $\exists a.\ W'(a_w) \neq \emptyset$*, then* $\exists s_2, s_3$ *such that* $s_1 \rightarrow^* s_2$ *and* $s_2 \xrightarrow{a} s_3$*, where* $a_w$ *is the wait location of action* $a$*.*

To check these conditions, we encode the LTS in terms of µCRL2 process terms, by adding $h$ and $W$ as process parameters. The properties are then checked using symbolic model checking provided by the µCRL2 toolset [CGK+13]. Moreover, when the desired condition i) or ii) is not provable, the

tool is able to provide the path in the LTS that causes the blocking, as feedback to the user to locate the potential blocking error in the original program.

### 6.3.5 Soundness

In this section we state soundness of our system via the following two properties:

i) verified programs are partially correct (Theorem 6.1);

ii) non-blocking of an abstracted model of a program guarantees non-blocking of the program (Theorem 6.2).

We prove partial correctness in the same way as this was done in Section 4.4. Here we discuss only details that are relevant to the non-blocking reasoning, while for other information we refer to Section 4.4.

**Theorem 6.1** (Partial Correctness)**.** *If $c$ is a verified program with an initial state $\sigma_0$ and $\sigma_0 \rightsquigarrow^* \sigma$ where $\sigma = (h, tp.(t, s, \mathsf{assert}\ F), mm)$, then there exists a resource $\mathcal{R}$ such that abstracts$(\sigma, \mathcal{R})$ and $\mathcal{R}, s \models F$.*

*Proof.* Partial correctness follows directly from Invariant 4.1 (page 110), which states that every reachable state is valid. To ensure correctness of this invariant, we need to prove validity of the proof rules from Section 6.3.3. We sketch only the proof of the [*Reinit*] rule, while proving the other rules is straightforward.

$$[\mathit{Reinit}] \quad \frac{M{\downarrow_{R,R'}}}{\vdash \{\mathsf{Hist}(L, 1, R, M)\}\mathsf{reinit}(L, R')\{\mathsf{Hist}(L, 1, R', \epsilon)\}}$$

Let $\sigma$ and $\sigma'$ be the pre- and poststate of the $\mathsf{reinit}(L, R')$ command, respectively:

$$\sigma = (h, tp.(t, s, \mathsf{reinit}(L, R'); c), hm) \rightsquigarrow (h, tp.(t, s, c), hm') = \sigma'$$

Let $\mathcal{R}_\sigma$ be a resource that abstracts $\sigma$. Because $\sigma : \diamond$ we have:

$$\mathcal{R}_\sigma = \mathcal{R}_t * \mathcal{R}_{tp} \tag{6.1}$$

$$\mathcal{R}_t, s \models \mathsf{Hist}(L, 1, R, H) \tag{6.2}$$

$$\mathcal{R}_{tp} \vdash tp : \diamond \tag{6.3}$$

To prove $\sigma' : \diamond$, we need to find a resource $\mathcal{R}_{\sigma'}$ that abstracts the state $\sigma'$

such that:

$$\mathcal{R}_{\sigma'} = \mathcal{R}'_t * \mathcal{R}'_{tp} \tag{6.4}$$

$$\mathcal{R}'_t, s \models \mathsf{Hist}(L, 1, R', \epsilon) \tag{6.5}$$

$$\mathcal{R}'_{tp} \vdash tp : \diamond \tag{6.6}$$

Further, let $\sigma_0$ be the last initial state of the history, i.e., the poststate of the last $\mathsf{reinit}$ command, if any, or otherwise the poststate of the creation of the history. From (6.2), we have that in the state $\sigma$ there exists a history over $L$

$$\exists L \in dom(hm).hm(L) = (ih, [act_1, ..., act_n])$$

and the predicate $R$ holds over the initial heap $ih$, i.e., $R[ih(x)/x]_{\forall x \in R} = \mathsf{true}$. We write $ih \models R$. Therefore, $R$ also holds on the heap $h_0$ in $\sigma_0$, because then the values from the heap have been copied to the initial heap:

$$h_0 \models R \tag{6.7}$$

Furthermore, because from (6.2) the complete predicate $\mathsf{Hist}(L, 1, R, M)$ holds over $\mathcal{R}_t$, from the definition of compatibility of abstract model maps, we have that the abstract model map $\mathcal{M}_t$ in $\mathcal{R}_t$ is equivalent to the abstract model map $\mathcal{M}_\sigma$ in the resource $\mathcal{R}_\sigma$, i.e., the abstraction of the global state. Therefore, the trace $[act_1, ..., act_n]$ represents the complete trace recorded in the global history. Therefore, the term $M$ abstracts all updates in the program over locations in $fp(L)$ between states $\sigma_0$ and $\sigma'$; this is ensured because an update of a location $x \in fp(L)$ must be within a synchronised block, and synchronised blocks behave as if they are atomic.

From the premise of the [*Reinit*] rule, i.e., $M \downarrow_{R,R'}$, Definition 6.2 (page 191), and the statement (6.7), we have that every path in the LTS of $M$ that starts from $(h_0, \emptyset, M)$ ends in a final state $(h_{fin}, \emptyset, \epsilon)$ such that $h_{fin} \models R'$. The global trace is represented by one of these paths (where all silent transitions are eliminated), and thus $R'$ holds over the heap in $\sigma$. From the operational semantics of the $\mathsf{reinit}(L, R')$ command, the values from the heap in $\sigma$ are copied to the initial heap in the state $\sigma'$, while the global history is made empty; thus. $hm'(L) = (ih', \mathsf{nil})$, where $ih' \models R'$. The state $\sigma'$ can be abstracted by a resource $\mathcal{R}_{\sigma'} = \mathcal{R}'_t * \mathcal{R}'_{tp}$ such that: $\mathcal{R}'_{tp} = \mathcal{R}_{tp}$ (from where (6.3) is satisfied); while $\mathcal{R}'_t = (\mathcal{H}, \mathcal{M}')$, where $\mathcal{R}_t = (\mathcal{H}, \mathcal{M})$ and $\mathcal{M}' = \mathcal{M}[L \mapsto ih', 1, \mathsf{nil}]$. Proving 6.5 is then trivial. This concludes the proof. □

**Theorem 6.2** (Non-blocking). *i) Let* $\{\mathsf{true}\}c\{\mathsf{Hist}(L,1,R,M)\}$ *be derivable. If* $M{\downarrow_{R,\mathsf{true}}}$ *then the program* $c$ *is non-blocking with respect to* $L$.

*ii) Let* $\{\mathsf{Future}(L,1,R,M,\mathsf{true})\}c\{\mathsf{true}\}$ *be derivable. If* $M{\downarrow_R}$ *then the program* $c$ *is non-blocking with respect to* $L$.

*Proof.* This theorem states that non-blocking of the program is implied by non-blocking of its abstract model; soundness of i) and ii) is guaranteed under the following assumptions:

A1 The set $L$ is closed under *succession* of synchronised blocks: if in a given thread execution, a synchronised block on a lock $l_1 \in L$ is executed before a synchronised block on a lock $l_2$, then $l_2 \in L$;

A2 No thread in the program ends in an infinite loop that contains no synchronised block.

Below is the proof sketch of both parts separately:

i) Let $r$ be a maximal fair run of $c$ in which $\sigma_0$ is the initial state, $\sigma_1$ is the poststate of the command $\mathsf{crhist}(L,R)$ and there is no $\mathsf{reinit}(L,R')$ command. Let $\sigma_2$ be a state in $r$ in which a thread $t$ is waiting on $m$ ($m \in L$ or $m$ is a lock associated to a lock from $L$). To prove that $c$ is non-blocking with respect to $L$, we need to prove that in a later state $\sigma_3$, some thread has passed $m$.

   Thus, in the state $\sigma_2$, $t$ is in the prestate of a $\mathsf{wait}(g,m)$ command. If $g$ holds in $\sigma_2$, the above is trivially proved: the thread $t$ will pass the $\mathsf{wait}(g,m)$ command. If $g$ does not hold, the thread $t$ has to wait. Because $M$ abstracts the program, its LTS contains a path $p$ that represents $r$: $p$ starts from an initial state $s_0 = (h_0, \emptyset, M)$, (where $h_0$ is the initial heap created in state $\sigma_1$, $h_0 \models R$) and reaches a state $s_1 = (h, W', M')$, where $W'(m) \neq \emptyset$ (see rule [Act Fail] in Figure 6.9). From $M{\downarrow_{R,\mathsf{true}}}$, we have that $p$ will end in a final state in which $M = \epsilon$, and thus there will be a transition $a$ in $p$ such that $a_w = m$. This means that in the original program some thread (we know at least $t$) will pass $m$.

   Note that to ensure that the original program follows the behaviour of the model, it is important that threads can block only in a prestate of a *wait* command. This is ensured because of the assumptions A1 and A2 and because a thread may hold only a single lock at a time. This guarantees that no thread will hold a lock forever, and thus in a fair run a thread can not wait forever to enter a synchronised block.

In case $r$ contains the $\mathsf{reinit}(L, R)$ command, let $\sigma$ be the poststate of the last $\mathsf{reinit}(L, R)$ command. Then, we can view $\sigma$ as a poststate of the creation of the history, while the model $M$ abstracts the behaviour between $\sigma$ and the end state of the run $r$. The proof is then analogous.

ii) The proof is similar to i). In this case $M$ is the abstraction of the whole program $c$. Let $r$ be a maximal fair run of $c$ and $\sigma_0$ be the initial state of $r$. Further, let $\sigma_1$ be a state in which a thread $t$ is waiting on $m$, and we want to prove that in a later state $\sigma_2$, some thread has passed $m$. If the guard holds in $\sigma_1$, this is trivially proved. Otherwise, the thread $t$ has to wait. The LTS of $M$ contains a path $p$ that represents $r$: $p$ starts from an initial state $s_0 = (h_0, \emptyset, M)$, (where $h_0$ is the initial heap created in state $\sigma_0$, $h_0 \models R$) and reaches a state $s_1 = (h, W', M')$ where $W'(m) \neq \emptyset$. From $R{\downarrow_M}$ we know that there is a transition $a$ in $p$ such that $a_w = m$. This means that a thread $t'$ in the program $c$ will pass $m$, which concludes the proof.

□

## 6.4 Conclusions and Related Work

Guarded blocks are an efficient mechanism that allows threads to synchronise on the value of some shared data. However, this mechanism comes at a price. First, guarded blocks dictate the possible order of thread executions, which influences the program behaviour and the values of the shared data. Second, the values in the shared state may cause a thread to block, i.e., to wait forever to enter a guarded block. This interplay between the propagation of values and the interleaving of guarded blocks makes reasoning about these programs particularly difficult.

**Related work** We are not aware of much work done on verification of programs with guarded blocks. Leino et al. [LMS10] prove deadlock-freedom in programs with locking and message passing via channels. A blocking receive operation is allowed only when another thread has an obligation to send a message on the channel. This pattern is comparable with the *notify/wait* operations; however, a sent message on the buffer is not lost, but accessible to threads that try to receive later. In contrast, notifications are much more complicated to reason about (as observed by Leino et al.), because they can be lost if not send at the right time.

In [GCPV09], Gotsman et al. propose a fully-automated technique, based on RGSep logic [VP07], to prove that a non-blocking (CAS-based) algorithm *does not block*, i.e., its most general client satisfies one of the properties: *wait-freedom*, *lock-freedom* or *obstruction-freedom*. However, we target different programs, i.e., algorithms that use the wait/notifyAll pattern. For these algorithms, proving even the weakest of the above properties (obstruction-freedom) is inadequate; whether the program blocks depends on how the client uses this algorithm.

Our technique allows modular reasoning about the local contributions of threads to the global behaviour of a program. The analysis of non-blocking of the computed models, however, is not modular. Modularity could be achieved by extending the proof system with rules for rely-guarantee reasoning. In such a line of work, Popeea and Rybalchenko propose a reasoning system combining predicate abstraction and refinement with rely-guarantee rules, where automation is partially achieved by a procedure for solving recursion-free Horn clauses [PR12].

**Our technique** In this chapter we proposed a novel technique for reasoning about multithreaded programs with guarded blocks. The technique allows one to verify functional properties over the shared state and moreover, to prove that the program does not reach a state where threads are blocked, i.e., they wait forever.

Our technique breaks verification into two steps. First, we abstract parts of the program into representative abstract models, called *histories*, represented as process algebra terms. To build the history, the programmer provides modular and intuitive annotations to specify locally the effect of each method, while an automated analysis combines the local contributions into a complete (global) history. Second, we analyse the abstract model to prove functional and non-blocking properties of the original program.

We also showed how to reason about programs where threads may have infinite executions. The abstract program model is then called a *future*; it needs to be specified up-front, and local reasoning is used to show that the program indeed satisfies the behaviour prescribed in the predicted model.

*Future directions* We illustrated our approach on a simplified language, restricted to non-nested synchronised blocks with a specific form. We discussed in Section 6.2 that these restrictions are natural when using guarded blocks, and we sketched our ideas of how one can extend the approach to programs with nested blocks also. However, it will be worth to study these ideas more thoroughly, and to investigate which of these restrictions are beneficial to lift,

without causing unnecessary complications in the verification process.

Furthermore, it is useful to study how this approach can be applied to more complex languages, where the parallel constuct ($\|$) is replaced with the more complicated *fork*/*join* mechanism, and where synchronisation is also achieved via the *acquire*/*release* pattern.

Future work should also focus on tool support for the proposed verification technique and its practical evaluation on case studies.

# Chapter 7

# Conclusions

It has been almost 40 years since Owicki and Gries introduced the first method for reasoning about concurrent programs. Today, after years of active research in this area, concurrent verification is still a challenge.

Owicki and Gries gave insight in the nature of concurrent programs and showed that verification of these programs can be troublesome because of their unpredictable behaviour. This difficulty mainly arises because of the *stability problem*: the assertions used to reason about one thread do not necessarily stay stable because of the interleavings with other parallel threads. The approach of Owicki and Gries takes into account all possible interleavings between threads, but this makes the approach non-modular and hardly applicable in practice.

Since then, different concurrent verification techniques have been developed to allow modular reasoning. To overcome the stability problem, a modular technique provides a mechanism that protects an assertion from invalidation by interference with other threads; for example *permission-based separation logic* uses permissions to ensure stability of assertions. These techniques have shown to be sufficient for proving generic properties, such as data race-freedom or deadlock-freedom; however, proving *functional properties* that are application-specific remains a serious challenge.

## 7.1 Verification of Functional Properties

Functional properties of the program describe how the developer intends the program to behave. A program can be free of data races or free of null-pointer exceptions, but that does not guarantee that it does what we want it to do. To

prove a functional property, it is necessary that the requirements of the program are explicitly written in a formal specification language, describing the expected behaviour of the program. Verification of these properties is highly important but it is hard and faces many challenges.

This thesis contributes with novel techniques for the verification of concurrent programs, focusing mainly on functional behaviour of programs. Concretely, we focused on three verification challenges: i) verification of concurrent class invariants; ii) specification and verification of method contracts in a concurrent setting; and iii) verification of concurrent programs with guarded blocks.

**Class invariants** In a sequential setting, only method pre- and poststates are visible states where class invariants must hold, while breaking of an invariant is allowed within a method. However, in the presence of multiple threads, verification of class invariants becomes more complicated, because due to thread interleavings, internal method states are also visible.

We proposed a technique for verification of class invariants, which allows breaking of an invariant in explicitly specified segments. By using permission-based separation logic, we ensure that invariants are preserved outside of these segments. Our technique is flexible and permissive. It allows a thread to break the invariant without holding all permissions to locations referred by the invariant. Simultaneous breaking of the same class invariant by multiple threads is also possible, if all threads promise that they do not harm the invariant. The technique is modular, because it integrates the rules from the ownership-based type system.

**Method contracts** The stability problem brings a lot of complications when specifying and verifying method contracts. In particular, when a method uses internal synchronisation (which is usually the case), it becomes troublesome to write an expressive method contract via stable assertions.

This thesis addresses this problem with a technique based on using *histories*. A history is a process algebra term that traces contributions of all threads in the form of actions. We use the power of process algebras to allow splitting the history into local histories, and merging local histories into a global history. A local history can be used to trace the behaviour of the local thread, while a method contract is expressed in terms of actions added to the local history. One cannot deduce any information about the shared state from a local history, because it is incomplete and the actions from the local history might have been interleaved with actions from a local history from another thread. However,

the local history remembers the past and allows one to postpone the reasoning for later, when the program is in a stable state. In such a state, local histories are combined to a complete global history, and by analysing the traces from the global history, we deduce properties about the shared state. Importantly, actions in the history have their own contracts, and thus reasoning about the traces in the history is equivalent to reasoning about a sequential program.

The history machinery requires additional history-based specifications, but the advantage of the approach is that these specifications are intuitive and easy to write. The local specifications are no more complicated than specifications of a sequential program. To combine local histories, automated analysis based on process algebra is used and therefore, no logic is required from the specification side. Moreover, the method allows permissive specifications and importantly, there is no restriction of how the client can use a given specified data structure.

Reasoning about the history requires calculating its traces. However, in practice, reasoning about the abstract model is much easier that reasoning about the original program. First, the model abstracts all unnecessary details from the original code; second, we allow multiple abstractions, each of them gathering information about a specific set of shared locations only; and third, history reinitialisation is allowed to keep these abstractions small and manageable. Moreover, our approach is integrated on top of a standard reasoning system based on permission-based separation logic. Our idea is that histories do not interfere with the standard way of reasoning. A history may be used only in a specific part of the program when reasoning otherwise is complicated.

Furthermore, we expect that in certain scenarios, it is possible to allow simplifying the process algebra term during the construction of the history, which will simplify the process of reasoning about the history.

Nevertheless, it is worth to investigate more into making the approach more efficient and easy to use. We believe that the core idea of using process algebra-based histories is useful because it solves some crucial problems in functional verification and therefore, can be applicable in different verification domains.

**Verification of liveness properties** An additional complication for reasoning about concurrent programs is brought by the fact that using synchronisation mechanisms may easily cause the program to block. Blocking and non-termination may happen for various reasons, and therefore, proving that the program terminates is a very difficult task.

In this thesis we concentrate on programs with *guarded blocks* and developed a technique for proving that threads in the program cannot block waiting forever

to enter a given guarded block. The core idea of this technique is history-based reasoning: parts of the program are abstracted by process algebra terms, and reasoning about non-blocking of the original program is done via reasoning about the abstract model. Moreover, we extended the approach with future-based reasoning to allow proving non-blocking about programs where threads have infinite executions.

What is especially challenging about verification of programs with guarded blocks is that there is a subtle interplay between the propagation of values and the interleaving of guarded blocks: the set of possible interleavings restricts the possible values of the variables, while these in turn affect the possible future interleavings. In a way, guarded blocks disable some of the thread interleavings. Therefore, the contribution of our approach is not only that we can prove non-blocking, but also that we allow one to prove more precise functional properties over the share state, i.e., properties that can be a result of the possible thread interleavings only.

Our approach is restricted to programs written in a simple procedural language. We do allow non-nested synchronised blocks only, but we believe that the approach is also useful for programs with nested blocks, if one can prove that the code within a synchronised block is non-blocking. Given the nature and complexity of the problem, it is expected that certain restrictions and assumptions are necessary; however, it is worth to study whether some of them can be lifted so that the approach will be applicable to a broader range of programs.

## 7.2 Formal Verification in Practice

We showed in this thesis that axiomatic reasoning (and formal verification in general) faces many challenges, especially when the program consists of multiple threads. It is therefore understandable that in a practical industrial environment this style of reasoning has difficulties to win over the other already broadly accepted verification techniques, such as the standard testing techniques.

First, formal verification requires highly skilled engineers with considerable knowledge in a very specific area. At this moment this is a problem. Investment in this area of expertise is necessary but it also costs money and time. Second, the process of verification is slow. Writing formal specifications is time consuming and moreover, specifications are needed also for all library components that are used in the program. Furthermore, formal verification techniques are usually restricted to certain programs only and are not mature enough to support complex languages. These disadvantages increase the scepticism over

the benefits of formal verification.

Although the progress of formal verification techniques is evident, its pace is not sufficient to follow the new trends in software engineering. Different programming languages are being built with complex features that bring challenges to verification techniques. Furthermore, in practice the software is usually developed in a dynamic environment (like agile-style of development), where systems are exposed to fast changes. This is a problem for verification techniques, which are not very responsive to change. In [CDD+15] Calcagno et al. describe their experience in integrating a verification tool into the development process at Facebook. They state that in such an environment where the release cycle is very fast, a verification technique should offer features like: full automation, scalability, precision, and fast reporting. At this moment, the currently existing verification techniques find it difficult to cope with these requirements.

But formal verification has its place. Its strong side is the fact that it uses the power of mathematical logic and deduces conclusions that are supported by mathematical proofs. In practice of course, correctness is not absolute but still, once a property is proven, we can safely rely on its correctness. Verification techniques are expensive, limited in their possibilities, but are strong and do guarantee safety. Therefore, the target of these techniques are environments where safety and correctness of software are critical requirements.

Using formal verification techniques in isolation is hard, and these are often combined with standard testing techniques. For example, the core functionality of the system can be formally verified, and combined with testing techniques to verify the system completely. Experience shows that the use of verification techniques significantly increase the software quality of these systems. This gives us the right to believe that formal verification is indeed promising.

Finally, I believe that in the near future we will have more available formal and semi-formal verification tools and that many promising scientific ideas will be integrated as part of the development process in various software systems. Formal verification will not guarantee absolute correctness of software, but it will move us much *closer to reliable software*.

# Appendices

# Appendix A

# Common auxiliary definitions

**Tuple projection,** $T^k$

$$\frac{T = (X_1, X_2, ..., X_n)}{T^k = X_k \text{ where } k \in \{1, ..., n\}}$$

**Separation conjunction,** $\circledast_{i \in S} F_i$

$$\circledast_{i \in \{1,2,...,n\}} F_i = F_1 * F_2 * ... * F_n$$

# Appendix B

# Auxiliary definitions for Chapter 4

**Class invariants,** $inv(C)$

$$\frac{\mathsf{class}\ C\{\ \overline{fd}\ \overline{md}\ \overline{\mathsf{Invariant}\ I}\ \overline{pd}\}}{inv(C) = \overline{\mathsf{Invariant}\ I}}$$

**Class fields,** $fld(C)$

$$\frac{\mathsf{class}\ C\{\ \overline{fd}\ \overline{md}\ \overline{inv}\ \overline{pd}\}}{fld(C) = \overline{fd}}$$

**Relevant fields of a class,** $relFld(C)$

$$\frac{\mathsf{class}\ C\{\overline{fd}\ \overline{md}\ \overline{inv}\ \overline{pd}\}}{relFld(C) = \bigcup_{Tf \in \overline{fd}, T^1 = \mathsf{rep}} relAux(T^2)}$$

$$\frac{\mathsf{class}\ C\{\overline{fd}\ \overline{md}\ \overline{inv}\ \overline{pd}\}}{relAux(C) = \bigcup_{Tf \in \overline{fd}, T^1 \in \{\mathsf{self},\mathsf{rep},\mathsf{peer}\}} (Tf) \cup relFld(T^2)}$$

**Class invariant fields** $\mathit{fld}(C, I)$

$$\frac{\mathsf{class}\ C\{\overline{fd}\ \overline{md}\ inv_1\ \overline{inv}\ \overline{pd}\} \quad inv_1 = \mathsf{Invariant}\ I : F_{inv}}{\mathit{fld}(C, I) = \mathit{fld}(F_{inv})}$$

$$\mathit{fld}(F_{inv}) = \begin{cases} \bigcup_{i=1..n}(T_i\ e_i) \cup (U\ f) & \text{if } F_{inv} ::= e_1.e_2...e_n.f\ (f : U\ \forall i = 1..n.e_i : T_i) \\ \bigcup_{i=1..n} \mathit{fld}(e_{inv_i}) & \text{if } F_{inv} ::= op(e_{inv_1}, ..., e_{inv_n}) \\ \mathit{fld}(F_{inv_1}) \cup \mathit{fld}(F_{inv_2}) & \text{if } F_{inv} ::= F_{inv_1} \oplus F_{inv_2} \\ \mathit{fld}(F_{inv_1}) & \text{if } F_{inv} ::= (qt\ T\ x)(F_{inv_1}) \end{cases}$$

**Resource invariant fields,** $\mathit{fldResInv}(C)$

$$\frac{\mathsf{class}\ C\{\ \overline{fd}\ \overline{md}\ \overline{inv}\ \overline{pd}\ res\_inv = F_{res}\}}{\mathit{fldResInv}(C) = \mathit{fld}(F_{res})}$$

$$\mathit{fld}(F_{res}) = \begin{cases} (T\ f) & \text{if } F_{res} ::= e.f \overset{\pi}{\mapsto} e, f : T \\ \mathit{fld}(F'_{res}) & \text{if } F_{res} ::= e.P(\overline{v})\ (\mathsf{pred}\ P(\overline{T}\ \overline{x}) = F'_{res}) \\ \mathit{fld}(F_{res_1}) \cup \mathit{fld}(F_{res_2}) & \text{if } F_{res} ::= F_{res_1} \oplus F_{res_2} \\ \mathit{fld}(F_{res_1}) & \text{if } F_{res} ::= (qt\ T\ x)(F_{res_1}) \\ \emptyset & \text{otherwise} \end{cases}$$

**Class of field** $\mathit{classOf}(C, f)$

$$\frac{\mathsf{class}\ C\{fd_1\ \overline{fd}\ \overline{md}\ \overline{inv}\ \overline{pd}\} \quad fd_1 = T\ f}{\mathit{classOf}(C, f) = T^2}$$

**Method inline,** $m\_inline(C, m, o, \overline{v})$

$$\frac{\mathsf{class}\ C\ \{\overline{fd}\ md_1\ \overline{md}\ \overline{inv}\ \overline{pd}\} \quad md_1 = spec\ \mathsf{void}\ m(\overline{T}\ \overline{x})\{c\}}{m\_inline(C, m, o, \overline{v}) = c[o/\mathsf{this}, \overline{v}/\overline{x}]}$$

**Thread inline**

$$\frac{\mathsf{class}\ C\ \{\overline{fd}\ md_1\ \overline{md}\ \overline{inv}\ \overline{pd}\} \quad md_1 =\ spec\ \mathsf{void}\ run()\{c\}}{t\_inline(C, o) = c[o/\mathsf{this}]}$$

**Predicate inline**

$$\frac{\mathsf{class}\ C\ \{\overline{fd}\ \overline{md}\ \overline{inv}\ pd_1\ \overline{pd}\} \quad pd_1 = \mathsf{pred}\ P(\overline{T}\ \overline{x}) = F}{p\_inline(C, P, o, \overline{v}) = F[o/\mathsf{this}, \overline{v}/\overline{x}]}$$

**InitStore**

$$\frac{st \in \mathsf{Store} = \mathsf{FieldId} \rightharpoonup \mathsf{Value} \quad \mathsf{class}\ C\ \{T_1\ fd_1, ..., T_n\ fd_n\ \overline{md}\ \overline{inv}\ \overline{pd}\} \quad dom(st) = \{fd_1, ..., fd_n\} \quad \forall i = 1..n.st(i) = df(T_i)}{initStore(C) = st}$$

**Method pre- and postcondition** $precond(C, m)$

$$\frac{\mathsf{class}\ C\ \{\overline{fd}\ m \cup \overline{md}\ \overline{inv}\ \overline{pd}\} \quad \mathsf{requires}\ F\ \mathsf{ensures}\ F'\ T\ m(\overline{T}\ \overline{x})\{c\}}{precond(C, m) = F \quad postcond(C, m) = F'}$$

**Writeable locations,** $writeLocs(F)$

$$writeLocs(F) = \{o.f \mid o.f \in locs(F), perm(F, o.f) \geq 1\}$$

$$locs(F) = \begin{cases} o.f & \text{if } F ::= o.f \overset{\pi}{\mapsto} v \\ locs(F_1) \cup locs(F_2) & \text{if } F ::= F_1 \oplus F_2 \\ \bigcup_{v:T} locs(F[v/x]) & \text{if } F ::= (qt\ T\ x)(F) \\ locs(F'[o/\mathsf{this}, \overline{v}/\overline{x}]) & \text{if } F ::= o.P(\overline{v})\ (\mathsf{pred}\ P(\overline{T}\ \overline{x}) = F') \\ \emptyset & \text{otherwise} \end{cases}$$

$$perm(F, o.f) = \begin{cases} \pi & \text{if } F = o.f \overset{\pi}{\mapsto} v \\ perm(F_1, o.f) + perm(F_2, o.f) & \text{if } F = F_1 * F_2 \\ max(perm(F_1, o.f), perm(F_2, o.f)) & \text{if } F = F_1 \oplus F_2, \oplus \in \{\wedge, \vee\} \\ perm(\bigwedge_{v \in T \setminus \{x\}} F[v/x], o.f) & \text{if } F = (qt\ T\ x)(F) \\ perm(F'[o'/\mathsf{this}, \overline{v}/\overline{x}], o.f) & \text{if } F = o'.P\ (\mathsf{pred}\ P(\overline{T}\ \overline{x}) = F') \\ 0 & \text{otherwise} \end{cases}$$

**Abstraction function,** $abstracts(\sigma, \mathcal{R})$

$$\sigma = (h, tp, lt, it), \quad \mathcal{R} = (\mathcal{H}, \mathcal{L}, \mathcal{T}, \mathcal{J})$$

$$\begin{array}{lcl}
abstracts(\sigma, \mathcal{R}) & \Leftrightarrow & abstracts(h, \mathcal{H}) \wedge abstracts(lt, \mathcal{L}) \wedge \\
 & & abstracts(it, \mathcal{T}) \wedge abstracts(tp, \mathcal{J}) \\
abstracts(h, \mathcal{H}) & \Leftrightarrow & dom(h) = dom(\mathcal{H}) \wedge \forall o \in dom(h).h(o)^1 = \mathcal{H}(o)^1 \\
 & & \wedge dom(h(o)^2) = dom(\mathcal{H}(o)^2) \wedge \\
 & & \forall f \in dom(h(o)^2).h(o.f) = \mathcal{H}(o.f)^1 \\
abstracts(lt, \mathcal{L}) & \Leftrightarrow & dom(lt) = dom(\mathcal{L}) \wedge \forall o \in dom(lt). \\
 & & lt(o) = \mathsf{Fresh} \Rightarrow \mathcal{L}(o) = (\mathsf{true}, \mathsf{false}, \mathsf{false}) \wedge \\
 & & lt(o) = \mathsf{Free} \Rightarrow \mathcal{L}(o) = (\mathsf{false}, \mathsf{true}, \mathsf{false}) \wedge \\
 & & lt(o) = t \Rightarrow \mathcal{L}(o) = (\mathsf{false}, \mathsf{true}, \mathsf{true}) \\
abstracts(it, \mathcal{T}) & \Leftrightarrow & dom(it) = dom(\mathcal{T}) \wedge \forall (o, I) \in dom(it).it(o, I) = \mathcal{T}(o, I)^1 \\
abstracts(tp, \mathcal{J}) & \Longleftrightarrow & dom(tp) = dom(\mathcal{J})
\end{array}$$

# Appendix C

# Auxiliary definitions for Chapter 5

**Action footprint,** $fp(C, a, o)$

$$\frac{\begin{array}{c}\mathsf{class}\ C\ \{\overline{fd}\ \overline{md}\ \overline{inv}\ \overline{pd}\ act_1\ \overline{act}\ \overline{proc}\} \quad \sigma = o/\mathsf{this} \\ act_1 = \mathsf{accessible}\ \{e_1.f_1, ..., e_n.f_n\}\ \mathsf{assignable}\ L'\ \mathsf{requires}\ F\ \mathsf{ensures}\ F'\ \mathsf{action}\ a(\overline{T}\ \overline{r});\end{array}}{fp(C, a, o) = \{e_1[\sigma].f_1, ..., e_n[\sigma].f_n\}}$$

**Action body,** $\mathsf{abody}(C, a, \overline{T})$

$$\frac{\begin{array}{c}\mathsf{class}\ C\ \{\overline{fd}\ \overline{md}\ \overline{inv}\ \overline{pd}\ act_1\ \overline{act}\ \overline{proc}\} \\ act_1 = \mathsf{accessible}\ L\ \mathsf{assignable}\ L'\ \mathsf{requires}\ F\ \mathsf{ensures}\ F'\ \mathsf{action}\ a(\overline{T}\ \overline{r});\end{array}}{\mathsf{abody}(C, a, \overline{T}) = \mathsf{accessible}\ L\ \mathsf{assignable}\ L'\ \mathsf{requires}\ F\ \mathsf{ensures}\ F'\ \mathsf{action}\ a(\overline{T}\ \overline{r})}$$

**Filtered actions,** $\mathsf{filter}(S)$

$$\mathsf{filter}(S) = \begin{cases} \mathsf{filter}(S') +\!\!+ [act] & \text{if } S = S' +\!\!+ [act, \mathsf{true}] \\ \mathsf{filter}(S') & \text{if } S = S' +\!\!+ [act, \mathsf{false}] \\ \mathsf{nil} & \text{if } S = \mathsf{nil} \end{cases}$$

**Abstraction function,** $abstracts(\sigma, \mathcal{R})$

$$\sigma = (h, tp, lt, it, hm, sa), \quad \mathcal{R} = (\mathcal{H}, \mathcal{L}, \mathcal{T}, \mathcal{J}, \mathcal{M}, \mathcal{A})$$

$$\begin{array}{lcl}
abstracts(\sigma, \mathcal{R}) & \Leftrightarrow & abstracts(h, \mathcal{H}) \wedge abstracts(lt, \mathcal{L}) \wedge \\
 & & abstracts(it, \mathcal{T}) \wedge abstracts(tp, \mathcal{J}) \\
abstracts(h, \mathcal{H}) & \Leftrightarrow & dom(h) = dom(\mathcal{H}) \wedge \forall o \in dom(h).h(o)^1 = \mathcal{H}(o)^1 \\
 & & \wedge dom(h(o)^2) = dom(\mathcal{H}(o)^2) \wedge \\
 & & \forall f \in dom(h(o)^2).h(o.f) = \mathcal{H}(o.f)^1 \\
abstracts(lt, \mathcal{L}) & \Leftrightarrow & dom(lt) = dom(\mathcal{L}) \wedge \forall o \in dom(lt). \\
 & & lt(o) = \mathsf{Fresh} \Rightarrow \mathcal{L}(o) = (\mathsf{true}, \mathsf{false}, \mathsf{false}) \wedge \\
 & & lt(o) = \mathsf{Free} \Rightarrow \mathcal{L}(o) = (\mathsf{false}, \mathsf{true}, \mathsf{false}) \wedge \\
 & & lt(o) = t \Rightarrow \mathcal{L}(o) = (\mathsf{false}, \mathsf{true}, \mathsf{true}) \\
abstracts(it, \mathcal{T}) & \Leftrightarrow & dom(it) = dom(\mathcal{T}) \wedge \\
 & & \forall (o, I) \in dom(it).it(o, I) = \mathcal{T}(o, I)^1 \\
abstracts(tp, \mathcal{J}) & \Leftrightarrow & dom(tp) = dom(\mathcal{J}) \\
abstracts(hm, \mathcal{M}) & \Leftrightarrow & dom(hm) = dom(\mathcal{M}) \wedge \forall \bar{l} \in dom(hm). \\
 & & hm(\bar{l})^1 = \mathcal{M}(\bar{l})^1 \wedge abstractsList(hm(\bar{l})^2, \mathcal{M}(\bar{l})^3) \\
abstracts(sa, \mathcal{A}) & \Leftrightarrow & abstractsList(sa, \mathcal{A}) \\
abstractsList(\overline{X}, \overline{Y}) & \Leftrightarrow & |\overline{X}| = |\overline{Y}| \wedge \forall i.\overline{X}[i] = (\overline{Y}[i])^1
\end{array}$$

# Appendix D

# Auxiliary definitions for Chapter 6

**Filtered actions,** $\mathsf{filter}(S)$

$$\mathsf{filter}(S) = \begin{cases} \mathsf{filter}(S') +\!\!+ [act] & \text{if } S = S' +\!\!+ [act, \mathsf{true}] \\ \mathsf{filter}(S') & \text{if } S = S' +\!\!+ [act, \mathsf{false}] \\ \mathsf{nil} & \text{if } S = \mathsf{nil} \end{cases}$$

**Abstraction function,** $abstracts(\sigma, \mathcal{R})$

$$\begin{array}{lcl} & & \sigma = (h, tp, mm), \quad \mathcal{R} = (\mathcal{H}, \mathcal{M}) \\ abstracts(\sigma, \mathcal{R}) & \Leftrightarrow & abstracts(h, \mathcal{H}) \wedge abstracts(mm, \mathcal{M}) \\ abstracts(h, \mathcal{H}) & \Leftrightarrow & dom(h) = dom(\mathcal{H}) \wedge \forall x \in dom(h).h(x)^1 = \mathcal{H}(x)^1 \\ abstracts(mm, \mathcal{M}) & \Leftrightarrow & dom(mm) = dom(\mathcal{M}) \wedge \forall L \in dom(mm). \\ & & mm(L)^1 = \mathcal{M}(L)^1 \wedge \\ & & [mm(L)^2 = (\mathsf{Hist}, H_m) \wedge \mathcal{M}(L)^2 = (\mathsf{Hist}, H_{\mathcal{M}}) \wedge \\ & & abstractsList(H_m, H_{\mathcal{M}})] \vee \\ & & [mm(L)^2 = (\mathsf{Future}, F_m) \wedge \mathcal{M}(L)^2 = (\mathsf{Future}, F_{\mathcal{M}}, b) \\ & & \wedge abstractsSet(F_m, F_{\mathcal{M}})] \\ abstractsList(X, Y) & \Leftrightarrow & |X| = |Y| \wedge \forall i.X[i] = Y[i]^1 \\ abstractsSet(SX, SY) & \Leftrightarrow & |SX| = |SY| \wedge \forall X \in SX. \exists Y \in SY. \\ & & abstractsList(X, Y) \end{array}$$

# List of papers by the author


[1] Stefan Blom, Marieke Huisman, and Marina Zaharieva-Stojanovski. History-based verification of functional behaviour of concurrent programs. In *Software Engineering and Formal Methods, SEFM, York, UK*, 2015.

[2] Marina Zaharieva-Stojanovski, Stefan Blom, Dilian Gurov, and Marieke Huisman. Reasoning about concurrent programs with guarded blocks. Submitted.

[3] Marina Zaharieva-Stojanovski and Marieke Huisman. Verifying class invariants in concurrent programs. In *17th International Conference, FASE, Grenoble, France*, pages 230–245, 2014.

[4] Afshin Amighi, Stefan Blom, Marieke Huisman, Wojciech Mostowski, and Marina 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*, pages 725–733, 2014.

[5] Afshin Amighi, Stefan Blom, Saeed Darabi, Marieke Huisman, Wojciech Mostowski, and Marina 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*, pages 172–216, 2014.

[6] Marina Zaharieva-Stojanovski, Marieke Huisman, and Stefan Blom. Verifying functional behaviour of concurrent programs. In *Proceedings of 16th Workshop on Formal Techniques for Java-like Programs*, FTfJP'14. ACM, 2014.

[7] Marina Zaharieva-Stojanovski, Marieke Huisman, and Stefan Blom. A history of BlockingQueues. In *Sixth Workshop on Formal Languages and Analysis of Contract-Oriented Software, FLACOS 2012, Bertinoro, Italy*, pages 31–35, 2012.

[8] Afshin Amighi, Stefan Blom, Marieke Huisman, and Marina Zaharieva-Stojanovski. The vercors project: setting up basecamp. In *Proceedings of the sixth workshop on Programming Languages meets Program Verification, PLPV 2012, Philadelphia, PA, USA*, pages 71–82, 2012.

[9] S. C. C. Blom, M. Huisman, and M. Zaharieva-Stojanovski. History-based verification of functional behaviour of concurrent programs. Technical Report TR-CTIT-15-02, Centre for Telematics and Information Technology, University of Twente, 2015.

[10] M. Zaharieva-Stojanovski and M. Huisman. Verifying class invariants in concurrent programs. Technical Report TR-CTIT-13-10, Centre for Telematics and Information Technology, University of Twente, 2014.

# References


[ABH14a] Afshin Amighi, Stefan Blom, and Marieke Huisman. Resource protection using atomics - patterns and verification. In *Programming Languages and Systems - 12th Asian Symposium, APLAS, Singapore*, pages 255–274, 2014.

[ABH+14b] Afshin Amighi, Stefan Blom, Marieke Huisman, Wojciech Mostowski, and Marina Zaharieva-Stojanovski. Formal specifications for Java's synchronisation classes. In *22nd Euromicro International Conference on Parallel, Distributed, and Network-Based Processing, PDP Torino, Italy*, pages 725–733, 2014.

[AdBO09] Krzysztof R. Apt, Frank S. de Boer, and Ernst-Rüdiger Olderog. Disjoint parallel programs. In *Verification of Sequential and Concurrent Programs*, Texts in Computer Science, pages 243–266. Springer London, 2009.

[AGVY11] Edward Aftandilian, Samuel Z. Guyer, Martin T. Vechev, and Eran Yahav. Asynchronous assertions. In *Proceedings of the 26th Annual ACM SIGPLAN Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA, part of SPLASH Portland, OR, USA*, pages 275–288, 2011.

[AHHH14] Afshin Amighi, Christian Haack, Marieke Huisman, and Clément Hurlin. Permission-based separation logic for multithreaded Java programs. *CoRR*, abs/1411.0851, 2014.

[AKC02] Jonathan Aldrich, Valentin Kostadinov, and Craig Chambers. Alias annotations for program understanding. In *Proceedings of the 2002 ACM SIGPLAN Conference on Object-Oriented*

*Programming Systems, Languages and Applications, OOPSLA, Seattle, Washington, USA*, pages 311–330, 2002.

[Amd67] Gene M. Amdahl. Validity of the single processor approach to achieving large scale computing capabilities. In *Proceedings of the Spring Joint Computer Conference*, AFIPS '67 (Spring), pages 483–485, New York, NY, USA, 1967. ACM.

[Apt81] Krzysztof R. Apt. Ten years of hoare's logic: A survey - part I. *ACM Trans. Program. Lang. Syst.*, 3(4):431–483, 1981.

[Apt83] Krzysztof R. Apt. Ten years of hoare's logic: A survey - part II: Nondeterminism. *Theoretical Computer Science*, 28(12):83 – 109, 1983.

[BCO05] Josh Berdine, Cristiano Calcagno, and Peter W. O'Hearn. Smallfoot: Modular automatic assertion checking with separation logic. In *Formal Methods for Components and Objects, 4th International Symposium, FMCO, Amsterdam, The Netherlands, Revised Lectures*, volume 4111 of *Lecture Notes in Computer Science*, pages 115–137. Springer, 2005.

[BCOP05] Richard Bornat, Cristiano Calcagno, Peter O'Hearn, and Matthew Parkinson. Permission accounting in separation logic. In *POPL*, pages 259–270. ACM, 2005.

[BDF+04] Michael Barnett, Robert DeLine, Manuel Fähndrich, K. Rustan M. Leino, and Wolfram Schulte. Verification of object-oriented programs with invariants. *Journal of Object Technology*, 3(6):27–56, 2004.

[BDH15] Stefan Blom, Saeed Darabi, and Marieke Huisman. Verification of loop parallelisations. In *Fundamental Approaches to Software Engineering - 18th International Conference, FASE Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS London, UK*, pages 202–217, 2015.

[BH14] Stefan Blom and Marieke Huisman. The VerCors Tool for verification of concurrent programs. In *Formal Methods*, volume 8442 of *LNCS*, pages 127–131. Springer, 2014.

[BHM14] Stefan Blom, Marieke Huisman, and Matej Mihelcic. Specification and verification of GPGPU programs. *Sci. Comput. Program.*, 95:376–388, 2014.

[BHS07] Bernhard Beckert, Reiner Hähnle, and Peter H. Schmitt, editors. *Verification of Object-Oriented Software: The KeY Approach*. Number 4334. 2007.

[BHZ15] Stefan Blom, Marieke Huisman, and Marina Zaharieva-Stojanovski. History-based verification of functional behaviour of concurrent programs. Technical report, Enschede, 2015.

[BHZS15] Stefan Blom, Marieke Huisman, and Marina Zaharieva-Stojanovski. History-based verification of functional behaviour of concurrent programs. In *SEFM*, 2015.

[BK84] Jan A. Bergstra and Jan Willem Klop. Process algebra for synchronous communication. *Information and Control*, 60(1-3):109–137, January/February/March 1984.

[BLR02] Chandrasekhar Boyapati, Robert Lee, and Martin C. Rinard. Ownership types for safe programming: preventing data races and deadlocks. In *OOPSLA*, pages 211–230, 2002.

[BLS05] Mike Barnett, K. Rustan M. Leino, and Wolfram Schulte. The spec# programming system: An overview. In *CASSIS 2004, Construction and Analysis of Safe, Secure and Interoperable Smart devices*, volume 3362 of *Lecture Notes in Computer Science*, pages 49–69. Springer, 2005.

[BMR95] Alexander Borgida, John Mylopoulos, and Raymond Reiter. On the frame problem in procedure specifications. *IEEE Trans. Software Eng.*, 21(10):785–798, 1995.

[Boy03] J. Boyland. Checking interference with fractional permissions. In R. Cousot, editor, *Static Analysis Symposium*, volume 2694 of *LNCS*, pages 55–72. Springer, 2003.

[ByECD+06] Mike Barnett, Bor yuh Evan Chang, Robert Deline, Bart Jacobs, and K. Rustanm. Leino. Boogie: A modular reusable verifier for object-oriented programs. In *Formal Methods for Components and Objects: 4th International Symposium, FMCO 2005,*

*volume 4111 of Lecture Notes in Computer Science*, pages 364–387. Springer, 2006.

[CC14] Patrick Cousot and Radhia Cousot. Abstract interpretation: past, present and future. In *Joint Meeting of the Twenty-Third EACSL Annual Conference on Computer Science Logic (CSL) and the Twenty-Ninth Annual ACM/IEEE Symposium on Logic in Computer Science (LICS), CSL-LICS '14, Vienna, Austria*, page 2, 2014.

[CDD+15] Cristiano Calcagno, Dino Distefano, Jérémy Dubreil, Dominik Gabi, Pieter Hooimeijer, Martino Luca, Peter W. O'Hearn, Irene Papakonstantinou, Jim Purbrick, and Dulma Rodriguez. Moving fast with software verification. In *NASA Formal Methods - 7th International Symposium, NFM 2015, Pasadena, CA, USA*, pages 3–11, 2015.

[CDH+09a] Ernie Cohen, Markus Dahlweid, Mark Hillebrand, Dirk Leinenbach, MichałMoskal, Thomas Santen, Wolfram Schulte, and Stephan Tobies. Vcc: A practical system for verifying concurrent c. In *Proceedings of the 22Nd International Conference on Theorem Proving in Higher Order Logics*, TPHOLs '09, pages 23–42. Springer-Verlag, 2009.

[CDH+09b] Ernie Cohen, Markus Dahlweid, Mark A. Hillebrand, Dirk Leinenbach, Michal Moskal, Thomas Santen, Wolfram Schulte, and Stephan Tobies. VCC: A practical system for verifying concurrent C. In *TPHOLs*, pages 23–42, 2009.

[CGK+13] Sjoerd Cranen, JanFriso Groote, JeroenJ.A. Keiren, FrankP.M. Stappers, ErikP. de Vink, Wieger Wesselink, and TimA.C. Willemse. An overview of the mCRL2 toolset and its recent advances. In *TACAS*, volume 7795 of *LNCS*, pages 199–213. Springer, 2013.

[CK04] David R. Cok and Joseph R. Kiniry. ESC/Javail: Uniting esc/java and jml - progress and issues in building and using ESC/Java2. In *In Construction and Analysis of Safe, Secure and Interoperable Smart Devices: International Workshop, CASSIS*. SpringerVerlag, 2004.

[CMST10] Ernie Cohen, Michal Moskal, Wolfram Schulte, and Stephan Tobies. Local verification of global invariants in concurrent programs. In *CAV*, pages 480–494, 2010.

[Cok11] David R. Cok. OpenJML: JML for Javer 7 by extending OpenJDK. In *NASA Formal Methods - Third International Symposium, NFM 2011, Pasadena, CA, USA*, pages 472–479, 2011.

[DFMS08] Sophia Drossopoulou, Adrian Francalanza, Peter Müller, and Alexander J. Summers. A unified framework for verification techniques for object invariants. In *Types, Logics and Semantics for State*, 2008.

[DHA09] Robert Dockins, Aquinas Hobor, and Andrew W. Appel. A fresh look at separation algebras and share accounting. In *Proceedings of the 7th Asian Symposium on Programming Languages and Systems*, APLAS '09, pages 161–177, Berlin, Heidelberg, 2009. Springer-Verlag.

[Dij70] Edsger W. Dijkstra. Structured programming. In *Software Engineering Techniques*. NATO Science Committee, August 1970.

[Dij76] Edsger W. Dijkstra. *A Discipline of Programming*. Prentice-Hall, 1976.

[DM05] Werner Dietl and Peter Müller. Universes: Lightweight ownership for JML. *Journal of Object Technology*, 4(8):5–32, 2005.

[DM12] Werner Dietl and Peter Müller. Object ownership in program verification. In D. Clarke, J. Noble, and T. Wrigstad, editors, *Aliasing in Object-Oriented Programming*, LNCS. Springer-Verlag, 2012.

[DYBG+13] Thomas Dinsdale-Young, Lars Birkedal, Philippa Gardner, Matthew Parkinson, and Hongseok Yang. Views: Compositional reasoning for concurrent programs. In *Proceedings of POPL*, 2013.

[DYDG+10] Thomas Dinsdale-Young, Mike Dodds, Philippa Gardner, Matthew J. Parkinson, and Viktor Vafeiadis. Concurrent abstract predicates. In *ECOOP*, pages 504–528, 2010.

[EQT09] Tayfun Elmas, Shaz Qadeer, and Serdar Tasiran. A calculus of atomic actions. In *POPL*, pages 2–15, 2009.

[FFS07] Xinyu Feng, Rodrigo Ferreira, and Zhong Shao. On the relationship between concurrent separation logic and assume-guarantee reasoning. In *Programming Languages and Systems, 16th European Symposium on Programming, ESOP 2007, Held as Part of the Joint European Conferences on Theory and Practics of Software, ETAPS Braga, Portugal*, pages 173–188, 2007.

[Flo67] Robert W. Floyd. Assigning meanings to programs. *Proc. Symp. Appl. Math*, 19:19–31, 1967.

[FM12] Pietro Ferrara and Peter Müller. Automatic inference of access permissions. In *VMCAI*, pages 202–218, 2012.

[GBC+07] Alexey Gotsman, Josh Berdine, Byron Cook, Noam Rinetzky, and Mooly Sagiv. Local reasoning for storable locks and threads. In *APLAS*, 2007.

[GCPV09] Alexey Gotsman, Byron Cook, Matthew Parkinson, and Viktor Vafeiadis. Proving that non-blocking algorithms don't block. In *POPL*, pages 16–28. ACM, 2009.

[GMR+09] Jan Friso Groote, Aad Mathijssen, Michel A. Reniers, Yaroslav S. Usenko, and Muck van Weerdenburg. Analysis of distributed systems with mCRL2. *Process Algebra for Parallel and Distributed Processing*, 2009.

[GPU01] Jan Friso Groote, Alban Ponse, and Yaroslav S. Usenko. Linearization in parallel pCRL. *The Journal of Logic and Algebraic Programming*, 48(12):39 – 70, 2001.

[GR01] Jan Friso Groote and Michel A. Reniers. Algebraic process verification. In *Handbook of Process Algebra, chapter 17*, pages 1151–1208. Elsevier, 2001.

[HG12] Aquinas Hobor and Cristian Gherghina. Barriers in concurrent separation logic: Now with tool support! *Logical Methods in Computer Science*, 8(2:2):36, 2012.

[HH07] Marieke Huisman and Clément Hurlin. The stability problem for verification of concurrent object-oriented programs. In *VAMP*, 2007.

[HHH08] Christian Haack, Marieke Huisman, and Clément Hurlin. Reasoning about Java's reentrant locks. In *APLAS*, pages 171–187, 2008.

[HJMS03] Thomas A. Henzinger, Ranjit Jhala, Rupak Majumdar, and Grégoire Sutre. Software verification with BLAST. In *Model Checking Software, 10th International SPIN Workshop. Portland, OR, USA, May 9-10, 2003, Proceedings*, pages 235–239, 2003.

[HK00] Kees Huizing and Ruurd Kuiper. Verification of object oriented programs using class invariants. In *FASE*, pages 208–221, 2000.

[HLMS11] Stefan Heule, K. Rustan M. Leino, Peter Müller, and Alexander J. Summers. Fractional permissions without the fractions. In *Formal Techniques for Java-like Programs (FTfJP)*, 2011.

[HM15] Marieke Huisman and Wojciech Mostowski. A symbolic approach to permission accounting for concurrent reasoning, 2015. submitted.

[Hoa69] C. A. R. Hoare. An axiomatic basis for computer programming. *Commun. ACM*, 12(10):576–580, 1969.

[Hoa78] C. A. R. Hoare. Communicating sequential processes. *Commun. ACM*, 21(8):666–677, 1978.

[Hoa02] C. A. R. Hoare. Software pioneers. chapter Proof of Correctness of Data Representations, pages 385–396. Springer-Verlag New York, Inc., New York, NY, USA, 2002.

[HR14] Nir Hemed and Noam Rinetzky. Brief announcement: Contention-aware linearizability. In *PODC 2014*, 2014.

[HW73] C. A. R. Hoare and Niklaus Wirth. An axiomatic definition of the programming language PASCAL. *Acta Inf.*, 2:335–355, 1973.

[IO01] Samin S. Ishtiaq and Peter W. O'Hearn. Bi as an assertion language for mutable data structures. *SIGPLAN Not.*, 36(3):14–26, January 2001.

[JKM+14] Uri Juhasz, Ioannis T. Kassios, Peter Müller, Milos Novacek, Malte Schwerhoff, and Alexander J. Summers. Viper: A verification infrastructure for permission-based reasoning. Technical report, ETH Zurich, 2014.

[Jon83] Cliff B. Jones. Tentative steps toward a development method for interfering programs. *ACM Transactions on Programming Languages and Systems*, 5(4):596–619, 1983.

[JP11] Bart Jacobs and Frank Piessens. Expressive modular fine-grained concurrency specification. In *POPL*, pages 271–282, 2011.

[JPLS05] Bart Jacobs, Frank Piessens, K. Rustan M. Leino, and Wolfram Schulte. Safe concurrency for aggregate objects with invariants. In *SEFM*, pages 137–147, 2005.

[JPS+08] Bart Jacobs, Frank Piessens, Jan Smans, K. Rustan M. Leino, and Wolfram Schulte. A programming model for concurrent object-oriented programs. *ACM Trans. Program. Lang. Syst.*, 31(1), 2008.

[JSS+] Ralf Jung, David Swasey, Filip Sieczkowski, Kasper Svendsen, Aaron Turon, Lars Birkedal, and Derek Dreyer. Iris: Monoids and invariants as an orthogonal basis for concurrent reasoning. Accepted for publication at POPL 2015.

[Kan14] Jorne Kandziora. Runtime assertion checking of multithreaded Java programs - an extension of the strobe framework, August 2014.

[Kas06] Ioannis T. Kassios. Dynamic frames: Support for framing, dependencies and sharing without restrictions. In *FM*, pages 268–283, 2006.

[Kas11] Ioannis T. Kassios. The dynamic frames theory. *Formal Asp. Comput.*, 23(3):267–288, 2011.

[KL12] Jason Koenig and K. Rustan M. Leino. Getting started with dafny: A guide. In *Software Safety and Security - Tools for Analysis and Verification*, pages 152–181. 2012.

[LCC+05] Gary T. Leavens, Yoonsik Cheon, Curtis Clifton, Clyde Ruby, and David R. Cok. How the design of JML accommodates both runtime assertion checking and formal verification. *Sci. Comput. Program.*, 55(1-3):185–208, 2005.

[Lea99] Doug Lea. *Concurrent Programming in Javer. Second Edition: Design Principles and Patterns*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2nd edition, 1999.

[Lei98] K. Rustan M. Leino. Data groups: Specifying the modification of extended state. In *Proceedings of the 1998 ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages & Applications (OOPSLA '98), Vancouver, British Columbia, Canada.*, pages 144–153, 1998.

[Lei12] K. Rustan M. Leino. Developing verified programs with dafny. In *Verified Software: Theories, Tools, Experiments - 4th International Conference, VSTTE 2012, Philadelphia, PA, USA. Proceedings*, page 82, 2012.

[LG86] Barbara Liskov and John Guttag. *Abstraction and specification in program development*. MIT Press, Cambridge, MA, USA, 1986.

[LGH+78] Ralph L. London, John V. Guttag, James J. Horning, Butler W. Lampson, James G. Mitchell, and Gerald J. Popek. Proof rules for the programming language euclid. *Acta Inf.*, 10:1–26, 1978.

[LL05] K. Rustan M. Leino and Francesco Logozzo. Loop invariants on demand. In *Proceedings of the Third Asian Conference on Programming Languages and Systems*, APLAS'05, pages 119–134, Berlin, Heidelberg, 2005. Springer-Verlag.

[LLM07] Gary T. Leavens, K. Rustan M. Leino, and Peter Müller. Specification and verification challenges for sequential object-oriented programs. *Formal Asp. Comput.*, 19(2):159–189, 2007.

[LM04] K. Rustan M. Leino and Peter Müller. Object invariants in dynamic contexts. In *ECOOP 2004 - Object-Oriented Programming, 18th European Conference, Oslo, Norway, Proceedings*, pages 491–516, 2004.

[LM05] K. Rustan M. Leino and Peter Müller. Modular verification of static class invariants. In *FM 2005: Formal Methods, International Symposium of Formal Methods Europe, Newcastle, UK, Proceedings*, pages 26–42, 2005.

[LM09] K. Rustan Leino and Peter Müller. A basis for verifying multi-threaded programs. In *Proceedings of the 18th European Symposium on Programming Languages and Systems: Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2009*, ESOP '09, pages 378–393, Berlin, Heidelberg, 2009. Springer-Verlag.

[LMS09] K. Rustan Leino, Peter Müller, and Jan Smans. Foundations of security analysis and design v. chapter Verification of Concurrent Programs with Chalice, pages 195–222. Springer-Verlag, Berlin, Heidelberg, 2009.

[LMS10] K. Rustan M. Leino, Peter Müller, and Jan Smans. Deadlock-free channels and locks. In *ESOP*, pages 407–426, 2010.

[LN13] Ruy Ley-Wild and Aleksandar Nanevski. Subjective auxiliary state for coarse-grained concurrency. In *POPL*, pages 561–574, 2013.

[LPC+07] Gary T. Leavens, Erik Poll, Curtis Clifton, Yoonsik Cheon, Clyde Ruby, David Cok, Peter Müller, Joseph Kiniry, and Patrice Chalin. *JML Reference Manual*, February 2007.

[LPX07] Yi Lu, John Potter, and Jingling Xue. Validity invariants and effects. In *ECOOP*, pages 202–226, 2007.

[LW94] Barbara H. Liskov and Jeanette M. Wing. A behavioral notion of subtyping. *ACM Transactions on Programming Languages and Systems*, 16:1811–1841, 1994.

[MÖ2] Peter Müller. *Modular Specification and Verification of Object-oriented Programs*. Springer-Verlag, Berlin, Heidelberg, 2002.

[Mey92] Bertrand Meyer. Applying "design by contract". *Computer*, 25:40–51, October 1992.

[Mey97] Bertrand Meyer. *Object-Oriented Software Construction, 2nd Edition*. Prentice-Hall, 1997.

[Mil82] Robin Milner. *A Calculus of Communicating Systems*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 1982.

[Moo65] Gordon E. Moore. Cramming more components onto integrated circuits. *Electronics*, 38(8), April 1965.

[MPHL06] Peter Müller, Arnd Poetzsch-Heffter, and Gary T. Leavens. Modular invariants for layered object structures. *Sci. Comput. Program.*, 62(3):253–286, 2006.

[OG76a] Susan S. Owicki and David Gries. An axiomatic proof technique for parallel programs i. *Acta Inf.*, 6:319–340, 1976.

[OG76b] Susan S. Owicki and David Gries. Verifying properties of parallel programs: An axiomatic approach. *Commun. ACM*, 19(5):279–285, 1976.

[O'H04] Peter W. O'Hearn. Resources, concurrency and local reasoning. In *CONCUR*, pages 49–67, 2004.

[O'H07] Peter W. O'Hearn. Resources, concurrency, and local reasoning. *Theor. Comp. Sci.*, 375(1-3):271–307, 2007.

[Par05] Matthew J. Parkinson. Local reasoning for Java. Technical Report UCAM-CL-TR-654, University of Cambridge, Computer Laboratory, November 2005.

[PB05] Matthew J. Parkinson and Gavin M. Bierman. Separation logic and abstraction. In *POPL*, pages 247–258, 2005.

[PGB+05] Tim Peierls, Brian Goetz, Joshua Bloch, Joseph Bowbeer, Doug Lea, and David Holmes. *Java Concurrency in Practice*. Addison-Wesley Professional, 2005.

[PH97] Arnd Poetzsch-Heffter. *Specification and Verification of Object-Oriented Programs*. PhD thesis, Habilitation thesis, Technical University of Munich, 1997.

[PR12] Corneliu Popeea and Andrey Rybalchenko. Compositional termination proofs for multi-threaded programs. In *TACAS*, volume 7214 of *LNCS*. Springer, 2012.

[PS12] Matthew J. Parkinson and Alexander J. Summers. The relationship between separation logic and implicit dynamic frames. *Logical Methods in Computer Science*, 8(3), 2012.

[Rey02] John C. Reynolds. Separation logic: A logic for shared mutable data structures. In *17th IEEE Symposium on LICS 2002*, pages 55–74. IEEE Computer Society, 2002.

[RK07] Enric Rodríguez-Carbonell and Deepak Kapur. Generating all polynomial invariants in simple loops. *J. Symb. Comput.*, 42(4):443–476, 2007.

[RSSB98] Wolfgang Reif, Gerhard Schellhorn, Kurt Stenzel, and Michael Balser. Structured specifications and interactive proofs with KIV. In *Automated Deducation - A Basis for Applications*, 1998.

[SB14] Kasper Svendsen and Lars Birkedal. Impredicative concurrent abstract predicates. In *ESOP*, pages 149–168, 2014.

[Sch97] Fred B. Schneider. *On Concurrent Programming*. Graduate Texts in Computer Science. Springer, 1997.

[SDM09] Alexander J. Summers, Sophia Drossopoulou, and Peter Müller. The need for flexible object invariants. IWACO '09, pages 6:1–6:9, New York, NY, USA, 2009. ACM.

[SJP09] Jan Smans, Bart Jacobs, and Frank Piessens. Implicit dynamic frames: Combining dynamic frames and separation logic. In *ECOOP*, pages 148–172, 2009.

[SJP12] Jan Smans, Bart Jacobs, and Frank Piessens. Implicit dynamic frames. *ACM Trans. Program. Lang. Syst.*, 34(1):2, 2012.

[Vaf07] Viktor Vafeiadis. *Modular fine-grained concurrency verification*. PhD thesis, University of Cambridge, 2007.

[Vaf10] Viktor Vafeiadis. Automatically proving linearizability. In *CAV*, pages 450–464, 2010.

[VB99] Jan Vitek and Boris Bokowski. Confined types. In *Proceedings of the 1999 ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages & Applications (OOPSLA '99), Denver, Colorado, USA, November 1-5, 1999.*, pages 82–96, 1999.

[VP07] Viktor Vafeiadis and Matthew J. Parkinson. A marriage of rely/guarantee and separation logic. In *Proceedings of the 18th International Conference on Concurrency Theory (CONCUR 2007)*,

volume 4703 of *Lecture Notes in Computer Science*, pages 256–271, Berlin, 2007. Springer.

[Wei11] Benjamin Weiß. *Deductive Verification of Object-Oriented Software: Dynamic Frames, Dynamic Logic and Predicate Abstraction*. PhD thesis, Karlsruhe Institute of Technology, 2011.

[ZSBGH] Marina Zaharieva-Stojanovski, Stefan Blom, Dilian Gurov, and Marieke Huisman. Reasoning about concurrent programs with guarded blocks. Submitted.

[ZSH14] Marina Zaharieva-Stojanovski and Marieke Huisman. Verifying class invariants in concurrent programs. In *FASE*, pages 230–245, 2014.

# Samenvatting

Programmacorrectheid is een belangrijke kwaliteitseigenschap van elk softwaresysteem. Voordat een softwaretoepassing in productie gaat is het daarom van groot belang dat de software goed geverifieerd wordt, oftewel dat er gecontroleerd wordt dat de code aan de gestelde eisen voldoet. Een standaard vorm van verificatie is testen, waarbij het gedrag van het systeem wordt gecontroleerd door het uitvoeren van het programma met verschillende invoerdata. Deze vorm van verificatie kan echter nooit een garantie geven dat er geen bugs meer in de code te vinden zijn.

Aan safety-critical systemen (bijv. medische apparatuur) worden de eisen voor correctheid heel hoog gesteld, omdat een fout in het programma letterlijk levensgevaarlijk kan zijn. Hier zijn verificatietechnieken vereisd die met zeer hoog zekerheid de correctheid van het programma kunnen garanderen.

*Statische formele technieken* bieden deze hoge zekerheid. Dit zijn technieken gebaseerd op wiskundige logica. Door gebruik te maken van een formele methode, kunnen we bewijzen dat het programma aan de eisen voldoet zonder de code van het programma uit te voeren. Deze technieken zijn uitdagend, maar wel nuttig en aantrekkelijk wanneer correctheid belangrijk is.

Dit proefschrift gaat over statische formele verificatie, met name *axiomatische redenering*. We richten ons op *concurrente programma's* want deze worden tegenwoordig heel vaak gebruikt, terwijl er een gebrek is aan verificatietechnieken die concurrente code kunnen ondersteunen. Dat komt met name door de complexiteit van de statische analyse van concurrente code, omdat de aanwezigheid van een aantal threads leidt tot non-deterministisch gedrag van het programma.

We maken in dit proefschrift gebruik van *permission-based separation logic*, een logica ontwikkeld voor het redeneren over concurrente programma's. We breiden deze logica uit om verschillende eigenschappen van concurrente programma's te kunnen bewijzen. We zijn daarbij voornamelijk gericht op het

bewijzen van functionele eigenschappen van het programma.

In Hoofdstuk 1 van dit proefschrift laten we zien wat de belangrijkste uitdagingen in formele verificatie zijn die later in het proefschrift zijn behandeld. Daarna bespreken we in Deel I de achtergrond van het axiomatisch redeneren; we beschrijven met name separation logica, hoe deze tot stand is gekomen, en wat de voor- en nadelen zijn.

Deel II bevat een uitgebreide presentatie van de door ons ontwikkelde technieken. We richten ons op drie belangrijke uitdagingen. Ten eerste stellen we een nieuwe techniek voor voor het redeneren over concurrente programma's met *class invarianten*. Class invarianten zijn een belangrijk fundament voor programma verificatie, maar hun gebruik in concurrente programa's is nogal beperkt. We beschrijven hoe door gebruik te maken van onze techniek sommige van deze beperkingen vermeden kunnen worden. Ten tweede ontwikkelen we een nieuwe manier om het functionele gedrag van programma's te verifiëren. Dat beschrijft hoe de methodes in het programma op intuïtive manier gespecificeerd kunnen worden, en hoe we kunnen bewijzen dat de code aan deze specificatie voldoet. Ten derde stellen we een manier voor om over programma's met *guarded blocks* te kunnen redeneren. Hier richten we ons op het redeneren over zowel het functionele als het *blocking* gedrag van het programma.

Propositions accompanying the dissertation *"Closer to Reliable Software: Verifying functional behaviour of concurrent programs"*

Marina Zaharieva Stojanovski

1. Permissions in separation logic guarantee correct thread behaviour on a memory level, but other techniques are needed to verify correct thread behaviour on a more abstract level. [This thesis]
2. We can not expect push-button tools for verifying functional behaviour of programs; it is necessary for the user to describe the desired behaviour of the program. [This thesis]
3. Working in formal verification is very inspiring. It is a pity that we struggle to find a simple language to communicate our work to other people.
4. Artificial intelligence disrupts the way we live by combining magic and science.
5. I always admire mathematicians. Their ideas are timeless, applicable now or hundred years later.
6. What makes the communication between research and industry difficult is their different vision: research strives for innovation, which normally takes time, while industry needs solutions now.
7. We should not panic if software takes jobs away. It only leaves us more space for creative work.
8. A good research idea is like love; it always comes when you do not expect it.
9. Spend time for a cup of coffee with smart people who have different views than your own.
10. "Simple can be harder than complex; you have to work hard to get your thinking clean to make it simple." - Steve Jobs

Stellingen behorend bij het proefschrift *"Closer to Reliable Software: Verifying functional behaviour of concurrent programs"*

Marina Zaharieva Stojanovski

1. Permissions in separation logica garanderen dat threads correct gedrag hebben op geheugenniveau; er zijn echter andere technieken nodig om het gedrag van threads op een meer abstract niveau te kunnen verifiëren. [Dit proefschrift]

2. We moeten geen push-button tools verwachten voor verificatie van functioneel gedrag van programma's; het is noodzakelijk om de gebruiker het gewenste gedrag van het programma te laten beschrijven. [Dit proefschrift]

3. Het werk met formelle verificatie is heel inspirerend. Wat jammer is, is dat wij het lastig vinden om ons werk met anderen te bespreken.

4. Kunstmatige intelligentie verandert onze manier van leven door het combineren van magie met wetenschap.

5. Ik bewonder wiskundigen. Hun ideeën zijn tijdloos, toepasbaar nu of honderd jaar later.

6. De moelijke communicatie tussen wetenschap en industie komt door hun verschillende visie: wetenschap streeft naar innovatie waarvoor tijd meestal noodzakelijk is, terwijl industrie een instantane oplossing eist.

7. Als software onze banen inpikt zijn er geen redenen tot paniek. Wij zullen alleen meer ruimte hebben voor creatief werk.

8. Een goed wetenschappelijk idee is met verliefdheid te vergelijken; Het komt altijd als je het niet verwacht.

9. Besteed tijd voor een kop koffie met slimme mensen die andere denkbeelden hebben dan de jouwe.

10. "Simpel kan moeilijker zijn dan complex. Je moet hard werken om je gedachten schoon te krijgen om het simpel te maken." - Steve Jobs