SPECIFICATION AND VERIFICATION OF SHARED-MEMORY CONCURRENT PROGRAMS

LE DUY KHANH

(B.Eng.(Hons.), Ho Chi Minh City University of Technology)

A THESIS SUBMITTED FOR THE DEGREE OF
DOCTOR OF PHILOSOPHY

DEPARTMENT OF COMPUTER SCIENCE
NATIONAL UNIVERSITY OF SINGAPORE

2014

DECLARATION

I hereby declare that this thesis is my original work and it has been written by me in its entirety. I have duly acknowledged all the sources of information which have been used in the thesis.

This thesis has also not been submitted for any degree in any university previously.

Le Duy Khanh

8 December 2014

Acknowledgements

I am deeply grateful to my advisors, Professors Teo Yong Meng and Chin Wei Ngan. Without their invaluable technical and personal insight, guidance, and encouragement, none of the work presented in this thesis would have been possible. I am very grateful to Professors Wong Weng Fai, Roland Yap, and Peter Müller for being my thesis examiners and for giving me many insightful feedback. I am also thankful to Professor Dong Jin Song for his comments and feedback in the course of this thesis. I highly appreciate Professor Shengchao Qin for his critical comments on this thesis. I also would like to express my gratitude to Professor Thoai Nam for his guidance during my days as an undergraduate student at HCMUT and for his constant supports during my PhD journey at NUS.

I would like to thank my colleagues in the Systems & Networking Lab and Programming Languages & Software Engineering Lab, where I worked on this research. Many have contributed to the completion of this thesis, both academically and personally. Here I can only mention several (in no specific order): Verdi, Claudia, Marian, Saeid, Bogdan, Cristina, Xuyan, Seth, Dumi, Lavanya, An, Linh, Trang, Loc, Chanh, Trung, Thai, Andreea, Asankhaya, Cristian, Cristina, Yamilet. Many have graduated from the labs, but their presence made my PhD experience memorable. Other colleagues such as Khanh, Hiep, Mano (NUS), Hung (HCMUT), and Granville (HP Labs) helped me a lot during my research. I also appreciate all my friends in Singapore who made my PhD life fruitful.

Last but not least, I am indebted to my parents, my sister, and especially my wife, Thanh, who have always been by my side sharing my joys and sadness. I could not have finished this thesis without them.

ABSTRACT

The recent adoption of multi-core processors has accelerated the importance of formal verification for shared-memory concurrent programs. Understanding and reasoning about concurrent programs are more challenging than sequential programs because of the notoriously non-deterministic interleavings of concurrent threads. These interleavings may lead to violations of functional correctness, data-race freedom, and synchronization properties such as deadlock freedom. This results in low confidence in the reliability of software systems. Although recent advances in specification and verification have shown promise in increasing the reliability of shared-memory concurrent programs, they mainly focus on partial correctness and data-race freedom, and often ignore the verification of synchronization properties.

In shared-memory concurrent programs, threads, locks, and barriers are among the most commonly-used constructs and the most well-known sources of software bugs. The aim of this thesis is to develop methodologies for advancing verification of shared-memory concurrent programs, in particular to ensure partial correctness, data-race freedom, and synchronization properties of programs with these constructs.

First, we propose "*threads as resource*" to enable verification of first-class threads. Threads are first-class in existing programming languages, but current verification approaches do not fully consider threads as first-class. Reasoning about first-class threads is challenging because threads are dynamic and non-lexically-scoped in nature. Our approach considers threads as first-class citizens and allows the ownership of a thread (and its resource) to be flexibly split, combined, and (partially) transferred across procedure and thread boundaries. The approach also allows thread liveness to be precisely tracked. This enables verification of partial correctness and data-race freedom of intricate fork/join behaviors, including the multi-join pattern and threadpool idiom. The notion of "threads as resource" has recently inspired us to propose "*flow-aware resource predicate*" for more expressive verification of various concurrency mechanisms.

Second, threads and locks are widely-used, and their interactions could potentially lead to deadlocks that are not easy to verify. Therefore, we develop a framework for

ensuring *deadlock freedom* of shared-memory programs using fork/join concurrency and non-recursive locks. Our framework advocates the use of *precise locksets*, introduces *delayed lockset checking technique*, and integrates with the well-known concept of locklevel to form a *unified formalism* for verifying deadlock freedom of various scenarios, some of which are not fully studied in the literature. Experimental evaluation shows that, compared to the state-of-the-art deadlock verification system, our approach ensures deadlock freedom of programs with intricate interactions between thread and lock operations.

Lastly, we propose the use of *bounded permissions* for verifying *correct synchronization of static and dynamic barriers* in fork/join programs. Barriers are commonly-used in practice; hence, verifying correct synchronization of barriers is desirable because it can help improve the precision of compilers and analysers for their analyses and optimizations. However, static verification of barrier synchronization in fork/join programs is a hard problem and has mostly been neglected in the literature. This is because programmers must not only keep track of (possibly dynamic) number of participating threads, but also ensure that all participants proceed in correctly synchronized phases. To the best of our knowledge, ours is the first approach for verifying both static and dynamic barrier synchronization in fork/join programs. The approach has been applied to verify barrier synchronization in the SPLASH-2 benchmark suite.

List of Publications

1. **Threads as Resource for Concurrency Verification**

   Duy-Khanh Le, Wei-Ngan Chin, Yong-Meng Teo

   24th ACM SIGPLAN Symposium/Workshop on Partial Evaluation and Program (PEPM), Mumbai, India, Jan 13–14, 2015.

2. **An Expressive Framework for Verifying Deadlock Freedom**

   Duy-Khanh Le, Wei-Ngan Chin, and Yong Meng Teo

   11th International Symposium on Automated Technology for Verification and Analysis (ATVA), pp. 287–302, Springer LNCS 8172, Hanoi, Vietnam, Oct 15–18, 2013.

3. **Verification of Static and Dynamic Barrier Synchronization using Bounded Permissions**

   Duy-Khanh Le, Wei-Ngan Chin, and Yong Meng Teo

   15th International Conference on Formal Engineering Methods (ICFEM), pp. 232–249, Springer LNCS 8144, Queenstown, New Zealand, Oct 29 – Nov 1, 2013.

4. **Variable Permissions for Concurrency Verification**

   Duy-Khanh Le, Wei-Ngan Chin, and Yong Meng Teo

   14th International Conference on Formal Engineering Methods (ICFEM), pp. 5–21, Springer LNCS 7635, Kyoto, Japan, Nov 12–16, 2012.

Table of Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Formal Methods

Modern software is often large, complex, and error-prone. A recent study at Cambridge University research showed that the global cost of software bugs is approximately $312 billion annually [3]. This is a tremendous loss for companies and national economies. A software bug is used to describe an error, mistake or fault in a computer program's source code or design that produces unexpected results or causes the program to behave in unintended ways. Bugs make software systems less reliable. Therefore, ensuring reliability of software to reduce development and maintenance cost is of global interest and is also a grand challenge as pointed out by Tony Hoare [59].

Type checking is one of the very first techniques to ensure that a program only performs valid operations. An operation such as adding an integer to a string is invalid. Type-safe languages, such as Java and C♯ have greatly improved the reliability of software. Type systems in these high-level programming languages ensure that certain classes of errors never occur. Although type checking is completely automatic, it provides a low level of confidence because a type-checked program often does not imply its functional correctness.

Currently, in order to detect software bugs, the majority of software developers depend on testing; however, testing can only help show the presence of bugs, but hardly can prove the absence of them. In software testing, developers write input-output specifications in terms of unit tests and then execute this suite of tests to

check whether, with the given input, the program results in the desired output. The problem with this approach is that it may not discover all errors because it is difficult to write unit tests that foresee all possible execution paths [122]. Therefore, passing a test suite does not necessarily mean a program is error-free.

Formal methods are approaches to producing more reliable software systems. Formal methods, fundamentally, traverse all possible execution paths in a software program; therefore, they provide higher reliability by ensuring the absence of bugs. The essence of formal methods is to apply formal mathematical-based techniques for specification and verification of software systems. Cliff Jones, Peter O'Hearn, and Jim Woodcock [72] pointed out the importance of formal methods:

> *"Given the right computer-based tools, the use of formal methods could become widespread and transform software engineering."*

In their study, they showed that formal methods are popularly used in safety-critical domains such as banking and aviation. Big companies such as Microsoft [7, 28], Intel [75] and Compaq (now part of HP) [42] develop their own static verifiers to ensure the safety of their products.

Formal methods are divided into two main approaches: analysis and verification. Program analysis is designed for pre-defined properties that may not meet programmers' intentions. Program verification is directed towards users' needs. Users use a specification language to express their intention (a specification), a program verifier then checks if a program conforms to its specification. Given an annotated program as an input, a program verifier outputs proof obligations which are then discharged by theorem provers. This provides strong guarantee for correctness with respect to users' specifications.

Tony Hoare proposed the foundational use of logic for verification of sequential programs [57]. In Hoare logic, each program is associated with a triple $\{p\}C\{q\}$ which is interpreted as follows: given a program C beginning in state satisfying the pre-condition p, if it terminates, it will do so in a state satisfying q. This is called *partial correctness*. *Total correctness* additionally requires *program termination*, i.e.

it ensures that the program finally terminates. Hoare provided a complete set of axioms and rules for each sequential primitive which formed the foundation of program verification [57]. With the proliferation of shared-memory programs in the current multi-core era, new specification and verification methodologies are needed for ensuring the reliability of shared-memory concurrent programs.

1.2 Shared-Memory Concurrency in Multi-core Era

Historically, Moore's law [116] observed that the transistor density doubles roughly every two years. Nonetheless, due to the limit on the amount of heat a micro-processor chip could reasonably dissipate (which is known as the "power wall" [113]), increasing density is no longer used to increase clock rate. Instead, it is used to put multiple cores in a die. As a result, most computers and mobile devices today are "multi-core".

Multi-threading is a widespread programming model for concurrency. A concurrent program consists of multiple threads that can be created statically at compile time or dynamically at run time. These threads share the same address space and communicate with each other via shared memory. With the advent of multi-core systems, multi-threading is advantageous because well-written multi-threaded programs can run faster by exploiting parallelism on computer systems that have more cores. This is because a thread is a unit of execution, which can be scheduled to run on a processing core. Therefore, the more cores a system has, the more threads can be executed concurrently, and the more performance gains. In order to exploit parallelism, programmers use threading constructs (such as fork/join) for creating concurrent threads, and use synchronization constructs (such as locks and barriers) for synchronizing and coordinating concurrent accesses to shared resources.

Unfortunately, writing a correct concurrent program is generally difficult. Most programmers are used to thinking sequentially; however, concurrent programming forces them to consider interleavings among concurrent threads. Multiple interleavings can produce different results across different runs. Even worse, incorrectly-

synchronized programs could potentially incur concurrency bugs such as data races and deadlocks, which seriously reduce the reliability of concurrent programs. As pointed out by computer scientist Edward A. Lee [88] , threads are the culprit which discards the most essential and appealing properties of sequential computation such as understandability, predictability, and determinism. As a result, compared with sequential programs, concurrent programs are much harder to write.

1.3 Verification of Shared-Memory Concurrent Programs

Concurrent programs are difficult to write and it is even more difficult to check for their correctness [94]. The major challenge is that threads are notoriously nondeterministic; therefore, they may interleave with each other in an unexpected manner [14, 88]. As a result, in order to verify concurrent programs, we have to take into account an exponential number of different interleavings which causes a "state explosion" in both testing and model checking.

Fortunately, theoretical advances in program verification show promise when reasoning about shared-memory concurrent programs. In 1975, in her PhD thesis [109], Susan Owicki and her advisor, David Gries, came up with the very first tractable proof method for concurrent programs using Hoare-style parallel composition and conditional critical regions [58]. Owicki-Gries logic relies on the fact that concurrent threads are independent and they are allowed to communicate in critical regions to ensure mutual exclusion. The most complicated part of the logic is to check that each thread does not modify variables belonging to other threads. This requires global knowledge about the entire system. Another difficulty of this Hoare-style logic is aliasing. Aliasing arises if a memory location (e.g. a heap object or a stack variable) can be accessed through different symbolic names. This problem is even worse in the presence of arrays and other dynamically allocated data structures. More importantly, Owicki-Gries logic gears towards partial correctness and ignores other

properties such as data-race freedom and deadlock freedom.

Rely/Guarantee reasoning (RG) is another well-known approach to reasoning about concurrent programs proposed by Jones [69] in 1983. In contrast to Owicki-Gries logic which focuses on independence of threads, RG aims to specify possible interference among them. Each atomic step in a thread has to be captured in the rely and guarantee conditions to ensure that threads do not interfere with each other. The disadvantage of this approach is that it is difficult to capture all possible interference among threads because this requires global knowledge about all threads in the system. Additionally, RG is less memory-modular because it considers the entire memory as shared resources; therefore, it is usually hard to define global invariants for all these shared resources.

In the last decade, separation logic [64, 115, 132] has been proposed to advocate modular and local reasoning. The beauty of separation logic is the ability to exploit separation of resources in heap-manipulating programs using the separation connective $*$. A separation conjunction $p_1 * p_2$ states that a thread owns resources described by p_1 and at the same time but separately resources described by p_2. The local reasoning principle of separation logic is captured by the following frame rule:

$$\frac{\{p\}\ C\ \{q\}}{\{p * r\}\ C\ \{q * r\}}$$

This rule states that if we are able to verify a program C in a smaller memory state (described by $\{p\}\ C\ \{q\}$), it is safe for C to execute in a larger state as long as the extra state r does not interfere with the execution of C. This rule implicitly says that a thread only needs to care for its own business, which is described by p and q, and its specification can be attached to any specification r without redoing the proof.

Local reasoning is an important property for verifying shared-memory concurrent programs. It greatly improved modularity and motivated O'Hearn to propose Concurrent Separation Logic (CSL) [106]. CSL can be considered as a combination of Owicki-Gries logic and separation logic. CSL enables local reasoning principle by allowing threads to "mind their own business" [105]. In CSL, threads execute con-

currently using Hoare's parallel composition and communicate with each other only in conditional critical regions (CCRs) [58]. In the parallel composition, threads are independent from each other and no interference is allowed except in critical regions. Shared resources are captured by resource invariants. A thread entering a critical region obtains the invariant of the resource protected in the critical region. When it is inside the critical region, a thread views the shared resource as local without considering other threads. CSL was originally designed to handle heap resources and allow limited forms of concurrency in terms of parallel composition and CCR. Recent developments have extended CSL to deal with stack variables [16], dynamic locks and threads [45, 51, 52, 61], static barriers [62]. Although CSL and its variants [16, 45, 51, 52, 61, 62] guarantee partial correctness and race-freedom, they often ignore other synchronization properties such as deadlock freedom.

Because of local reasoning in separation logic, many works (RG^+) have applied it to rely/guarantee reasoning [35, 38, 39, 126]. The key idea is to split program states into shared states and private states. Shared states are treated in the same way with RG while private states are reasoned locally using the separation conjunction. This greatly reduces efforts to describe interference in shared states. RG^+ is considered more general than CSL because it is able to reason about concurrent programs with both disciplined concurrency and ad hoc synchronizations. However, it is still complicated to be adopted popularly compared with CSL because, besides pre- and post-conditions, RG^+ also requires interference specifications in terms of rely and guarantee conditions. Recently, Deny/Guarantee (DG) [35] is proposed to mitigate this drawback. In DG, deny and guarantee conditions become a part of pre- and post-conditions. Although DG and other RG^+ methods are expressive to reason about concurrent programs with dynamic creation of locks and threads, it is unclear how to extend them to verify other concurrency constructs such as barriers as well as to verify properties such as deadlock freedom and correct barrier synchronization.

Reasoning about program code is a very difficult task due to many different special exceptions and assumptions to ensure desired program behaviors. The proof can be

done by hand by abstracting the core algorithm of the program, writing its specification, and checking that the algorithm meets the specification. An apparent problem of this approach is that the core algorithm may interact with other components in unexpected ways. This indicates that the correctness of the core algorithm does not imply the correctness of the entire program. Besides, in case of large programs, it is not easy to extract their core algorithm. Especially, in the context of concurrent programs, threads may interleave non-deterministically. Therefore, it becomes much harder to abstract the core algorithm precisely and it is even more tedious to write proofs which account for all possible interleavings. As a result, computerized proofs (e.g. proofs generated by an automatic program verifier) are desirable.

Although fully automatic generation of verification proofs appears too difficult to achieve, programmers can help by annotating their intentions to guide the program verifiers. Therefore, a program verifier should come with an expressive specification logic allowing users to fully express their intention. However, expressiveness of the specification logic does not mean that it can be automated. The more expressive the logic is, the harder it is to automate the logic [71]. Often, a high degree of automation is a desirable property of program verifiers [59].

Though many program verifiers have implemented the above-mentioned logics in the last decade, they are of limited expressiveness or automation. Smallfoot [9] is among the very first CSL-based verifiers for concurrent programs. It comes with a complete decision procedure as well as excellent automation, but it only supports simplistic concurrency constructs such as parallel composition and conditional critical regions. Although Chalice [90] and Verifast [67] are expressive to reason about concurrent programs with fork/join and locks, they are of limited automation and require a lot of user annotations. For example, Verifast reported an annotation overhead which is in the order of 10 to 20 lines of annotation per line of code [65]. Furthermore, among existing verification systems, Chalice is the only system that could help prevent certain types of deadlocks. None of the above systems support verification of barrier synchronization.

In summary, although the literature has shown promise in specifying and verifying correctness of shared-memory programs, they mostly focus on partial correctness and data-race freedom, and often ignore the verification of synchronization properties such as deadlock freedom and correct barrier synchronization. Hence, in order to further improve the reliability of shared-memory concurrent software, methodologies are needed not only for reasoning about partial correctness and data-race freedom, but also for ensuring the synchronization properties.

1.4 Objective and Contributions

In view of the above review, it is worth noting that although existing works on specification and verification of shared-memory concurrent programs have achieved many promising advances, there remain the following research challenges:

- In mainstream languages, threads are first-class in that they can be dynamically created, stored in data structures, passed as parameters, and returned from procedures. However, current verification systems support reasoning about threads in a restricted way because threads are often represented by unique tokens that can neither be split nor shared. As such, the verification of first-class threads has not been fully investigated. Reasoning about first-class threads is challenging because threads are dynamic and non-lexically-scoped in nature. A thread can be dynamically created in a procedure (or a thread), but shared and joined in other procedures (or threads). Therefore, there is a need for expressive verification of first-class threads.

- Deadlock freedom is among the most desirable properties for concurrent programs. However, among existing specification and verification systems, only Chalice [89, 90] could prevent certain types of deadlocks such as those due to double lock acquisition and unordered locking. There are still other types of deadlocks that have almost been neglected in the literature such as those

due to the interactions between thread fork/join and lock acquire/release operations. With the profound use of threads and locks in large programs with many (possibly non-deterministic) execution branches, these interactions are not easy to follow [88]. These types of deadlocks are hard to verify by current approaches since the current pre-condition checking at the fork point is insufficient to prevent the deadlocks from happening. Therefore, it is desirable to have an expressive framework capable of verifying different deadlock scenarios, especially those due to the intricate interactions between fork/join and acquire/release operations

- Existing works focus mainly on concurrent programs manipulating (mutex) locks. Besides locks, barriers are among the most commonly-used synchronization constructs [13, 107]. Static verification of barrier synchronization is challenging because programmers must not only keep track of (possibly dynamic) number of participating threads, but also ensure that all participants proceed in correctly synchronized phases. As barriers are commonly used in practice [13, 107], correct barrier synchronization is a desirable property since it can provide compilers and analysers with important information for improving the precision of their analyses and optimizations such as reducing false sharing [68], may-happen-in-parallel analysis [93, 134], and data race detection [76]. However, verification of barrier synchronization has almost been neglected in the context of shared-memory fork/join programs.

The main objective of this thesis is to design a set of methodologies for reasoning about shared-memory programs, in terms of verifying partial correctness, data-race freedom, and synchronization properties such as deadlock freedom and correct barrier synchronization. Our expressive program logics, based on separation logic, are designed to reason about programs with first-class threads, locks, and barriers that are commonly used in shared-memory programming. The logics have been implemented into prototype tools and experimental evaluations demonstrate their capabilities for verifying many intricate programs. In particular, many of the programs implement

the multi-join pattern, intricate interactions between thread and lock operations, and dynamic barrier synchronization, which could not be verified by current verification approaches.

Specifically, towards automated verification of shared-memory programs, we make the following contributions:

- For reasoning about first-class threads, we propose "*threads as resource*" approach, allowing the ownership of a thread to be flexibly split, combined, and (partially) transferred across procedure and thread boundaries. We also allow thread liveness to be precisely tracked. This enables verification of partial correctness and data-race freedom of intricate fork/join behaviors such as multi-join pattern and threadpool idiom. The idea of "threads as resource" has also inspired our recently-proposed "*flow-aware resource predicate*" for more expressive verification of various concurrency mechanisms, including and beyond first-class threads.

- For ensuring deadlock-freedom of shared-memory programs manipulating fork/join concurrency and non-recursive locks, we develop an expressive framework that advocates the use of *precise locksets*, introduces *delayed lockset checking technique*, and integrates with the well-known notion of locklevel to form a *unified formalism* for verifying deadlock-freedom of various scenarios, including double lock acquisition, interactions between thread fork/join and lock acquire/release, and unordered locking. Specifically, compared to the state-of-the-art deadlock verification system, our approach ensures deadlock freedom of programs with intricate interactions between thread fork/join and lock acquire/release operations, which are not fully studied in the literature.

- Lastly, we present an approach for verifying correct synchronization of static and dynamic barriers in fork/join programs using *bounded permissions*. For verifying *static barriers*, the approach uses *bounded permissions* and *phase numbers* to keep track of the number of participants and barrier phases respectively. For

verifying *dynamic barriers*, the approach introduces *dynamic bounded permissions* to additionally keep track of the additions and/or removals of participants. Our approach has been proven sound, and a prototype of it has been applied to verify barrier synchronization in the SPLASH-2 benchmark suite.

Our methodologies proposed in this study advance the verification of shared-memory concurrent programs in multiple dimensions. First, we address different commonly-used concurrency constructs including fork/join, locks, and barriers. Our "threads as resource" approach enables reasoning about intricate fork/join concurrency and provides an infrastructure for reasoning about concurrent programs with locks and barriers. Based on "threads as resource", we advocate the use of *precise locksets*, introduce *delayed lockset checking technique* for reasoning about deadlock-free programs with locks. We also propose approaches for verifying correct synchronization of static and dynamic barriers. Second, we verify different program properties such as partial correctness, data-race freedom, deadlock freedom, and correct barrier synchronization. The proposed methodologies have been implemented into integrated tools for verifying concurrent programs.

We also addressed the issue of ensuring race-free accesses to program variables in the course of this research. Existing works often focus on ensuring safe (or race-free) concurrent accesses to heap data structures, but reasoning about concurrent accesses to program variables is not fully addressed. One solution is to apply the same permission system (e.g. fractional permissions [18]), designed for heap memory, to variables. "Variables as resource" [112] is such an approach. However, it is, in most cases, overly heavy [71]. We propose a new permission system, called variable permissions, which is simpler than existing permission systems in the literature. Therefore, it simplifies the verification and automatic inference of permissions. This contribution is not the major focus of this thesis, thus it is left in Appendix A.

This thesis focuses on methodologies for specifying and verifying shared-memory concurrent programs. Methods for program testing are not discussed in this thesis as testing is generally incomplete, i.e. it can show the presence of concurrency bugs, but

hardly can prove the absence of them. Similarly, techniques using model checking are not central to this study as they generally suffer from the "state explosion" problem. Furthermore, static analyses such as those based on type systems are only discussed briefly as they tend to be less expressive than specification logics. Comparative remarks between our work and these approaches will be presented in each chapter.

1.5 Organization of the Thesis

The organization of the thesis is as follows.

Chapter 2 discusses related theoretical advances in reasoning about shared-memory concurrent programs. The chapter also discusses open issues that motivate this thesis.

Chapter 3 introduces our "threads as resource" approach for reasoning about first-class threads. The main contribution is an expressive treatment of first-class threads to enable verification of more intricate fork/join behaviors. The chapter also presents "flow-aware resource predicate" for verifying various concurrency mechanisms.

Chapter 4 presents an expressive framework for verifying deadlock freedom. The main contributions of the framework are the use of precise locksets, the inroduction of delayed lockset checking technique, and the capability to verify various deadlock scenarios, some of which have not been adequately studied in the literature.

Chapter 5 presents our approach to verifying correct synchronization of both static and dynamic barriers in fork/join programs. The main contributions are the new permission system, called bounded permissions, and the use of this system for verifying synchronization of static and dynamic barriers.

Chapter 6 concludes the thesis and discusses future works.

Chapter 2

Related Work

In this chapter, we discuss theoretical advances and open issues in reasoning about shared-memory concurrent programs. More comprehensive comparisions between related works and our work will be presented in respective chapters.

Logics for specification and verification of shared-memory programs focus on two aspects of concurrent threads: independence and interference. Threads are independent if they access disjoint resources. Independence, therefore, enables local reasoning for each individual thread. Nonetheless, threads could interfere with each other in complicated ways, and hence require methodologies to describe their interference. Beside theoretical advances, automating the verification process is desirable as it reduces the manual (human) efforts for specification. We will also discuss some existing automatic verification systems in this chapter. Last but not least, we conclude this chapter with challenging open issues.

2.1 Reasoning about Independence among Threads

2.1.1 Owicki-Gries Logic

In 1969, Hoare [57] introduced an axiomatic approach for proving correctness of sequential programs. Hoare's triples are the basis of program verification. A triple $\{p\}\ C\ \{q\}$ states that given an execution of a program C beginning in a state sat-

isfying the pre-condition p, then if the execution terminates, it will do so in a state satisfying the post-condition q. Afterward, in [58], Hoare formalized concurrent execution of threads as a parallel composition with a resource r:

$$resource\ r:\ C_1\ ||\ \ldots\ ||\ C_n$$

Here, all threads $C_1, \ldots, C_n$ are executed in parallel. In order to cope with different interleavings among threads, Hoare proposed to protect shared resources in conditional critical regions (CCR):

$$with\ r\ when\ B\ do\ C$$

where r denotes a shared resource (i.e. a list of variables), B denotes the guard condition, and C denotes a piece of code that uses the resource r. Generally, a thread is allowed to test the state of the resource r by trying to acquire a semaphore associated with r. After successfully acquiring the semaphore, the thread checks condition B. If B is not satisfied, the thread will be placed on the queue of threads waiting for r and release the semaphore. If B is satisfied, it will enter the critical region, execute, and on completion invoke all processes in the waiting queue. The conditional critical region ensures that only one thread at a time has access to the shared resource r.

Following the work of Hoare, Owicki and Gries introduced the concept of non-interference among proofs of concurrent threads, which is known as Owicki-Gries Logic [110, 111, 109]. The logic assumes that a resource invariant $I(r)$ has been defined for each resource r. The proof rule of parallel composition is described as follows:

$$\frac{\{p_1\}\ C_1\ \{q_1\} \qquad \ldots \qquad \{p_n\}\ C_n\ \{q_n\} \qquad (\dagger)}{\{p_1 \wedge \ldots \wedge p_n \wedge I(r)\}\ resource\ r:\ C_1\ ||\ \ldots\ ||\ C_n\ \{q_1 \wedge \ldots \wedge q_n \wedge I(r)\}} \tag{2.1}$$

where the side condition (†) states that no thread C_i will interfere with the proof of thread C_j ($i \neq j$) and vice versa. More precisely, any intermediate assertions between atomic actions in the proof outline of C_j must be preserved by all atomic actions of C_i and vice versa. This ensures that threads do not interfere with each other during the execution.

The rule for conditional critical regions (CCRs) is formulated as follows:

$$\frac{\{I(r) \wedge p \wedge B\}\ C\ \{I(r) \wedge q\} \qquad \forall\ C_j{\neq}C : FV(p,q) \cap modifies(C_j){=}\phi}{\{p\}\ with\ r\ when\ B\ do\ C\ \{q\}}$$

where the side condition says that no variable in p or q is modified by other threads.

As pointed out by Owicki and Gries [111], the two above rules are inadequate even for simple programs. Therefore, they introduce auxiliary (or ghost) variables to capture additional information about concurrent threads. An auxiliary variable is a logical variable; it does not exist in the program but rather is to support proving the program's correctness. Auxiliary statements using auxiliary variables do not affect the control flow of the programs. Indeed, Owicki and Gries proved that auxiliary variables and their statements do not affect the correctness of verified programs.

Although elegant and easy to understand, Owicki-Gries logic has important limitations. The most important limitation is due to the side conditions mentioned in the two above rules for parallel composition and conditional critical region. As aforementioned, the side conditions require that a thread has to know the code of other threads in order to check for non-interference. This makes the method less compositional. Besides, in order to capture interference, the logic requires resource invariants and many auxiliary variables. These elements sometimes are difficult to specify precisely [126].

2.1.2 Concurrent Separation Logic

Separation logic (SL) [64, 115, 132] is an extension of Hoare's logic to support local reasoning of heap-manipulating programs. The strength of separation logic lies under the separation connective $*$. The separation conjunction $p_1 * p_2$ in an assertion

specifies heap states which can be split into two disjoint parts: the first part satisfies p_1 and the second part satisfies p_2. The most important benefit of separation logic is to allow local reasoning via the following frame rule:

$$\frac{\{p\}\ C\ \{q\} \qquad FV(r) \cap modifies(C)=\phi}{\{p * r\}\ C\ \{q * r\}}$$

The idea of local reasoning is that the specifications p and q of a module C only need to mention heap states accessed locally by C. This leads to clean verification of sequential heap-manipulating programs. The side condition is necessary to ensure that C does not modify stack variables mentioned in r. Separation logic is compositional in the sense that C can be composed with other modules in different contexts (i.e. different r) without re-doing the proof of C.

Discovering the strength of separation logic, O'Hearn [105, 106] proposed Concurrent Separation Logic (CSL) which extends separation logic to reason about concurrency. The parallel composition rule comes in naturally because of the separation nature of resources:

$$\frac{\{p_1\}\ C_1\ \{q_1\} \qquad \ldots \qquad \{p_n\}\ C_n\ \{q_n\} \qquad \forall\ i{\neq}j : FV(p_i, q_i) \cap modifies(C_j)=\phi}{\{p_1 * \ldots * p_n\}\ C_1\ ||\ldots||\ C_n\ \{q_1 * \ldots * q_n\}}$$

The rule states that a heap state can be split into multiple disjoint parts in such a way that threads only access their own part without interfering with the others. Verification of each individual thread is similar to that of a sequential program. In contrast to Owicki-Gries logic which always needs the side condition to ensure non-interference among threads (Equation 2.1), CSL by nature ensures non-interference in the heap. The side condition in this rule is to guarantee that stack variables mentioned in p_i and q_i of a thread C_i are not modified by other threads C_j $(i \neq j)$.

To support sharing of resources among concurrent threads, CSL adopts Hoare's conditional critical regions (CCRs) for mutual exclusion:

$$\frac{\{(I(r) * p) \wedge B\}\ C\ \{I(r) * q\} \qquad \forall\ C_j{\neq}C : FV(p,q) \cap modifies(C_j){=}\phi}{\{p\}\ with\ r\ when\ B\ do\ C\ \{q\}} \tag{2.2}$$

The rule is basically similar to that of Owicki-Gries except that it uses separation connective $*$ instead of conjunction $\wedge$ to ensure separation of heap resources. The side condition is to ensure that no stack variables mentioned in p and q is modified by other threads. A thread has full control over resource r when it is in the critical region C. The rule shows an important property of CSL: *ownership transfer*. Outside the critical region, the resource r is in a shared state and is owned by the invariant $I(r)$. The ownership of r is transferred to a thread when it acquires the semaphore to enter the critical region C. Upon leaving the critical region, the thread transfers the ownership of r back to the resource invariant $I(r)$. The ownership of the resource r later can be transferred to another thread entering the critical region C.

Though a powerful rule, the conditional critical region rule (Equation 2.2) is too restrictive in the sense that it does not allow concurrent reads of threads. Bornat et al. [15] incorporated fractional permissions [18] into CSL to overcome the restriction and allow more expressive sharing among threads, as elaborated in the following.

2.1.3 Fractional and Counting Permissions

Permissions are fundamental to specification and verification of concurrent programs. In concurrent separation logic, the basic heap node $x \mapsto E$, pronounced x *points to* E, asserts that it consists of a single cell with integer address x and integer content E. Heaps are connected together to form larger heaps by using the *separation connective* $*$. In order to reason about race-free sharing of resources among concurrent threads, heaps are enhanced with *permissions* π [15, 18]. A heap node $x \overset{\pi}{\mapsto} E$ indicates a permission to access the content E at the address x. A permission can be *partial* or *full* indicating read or write permission respectively. A permission (either full or

partial) can be split into multiple partial permissions which can be shared among threads. Partial permissions can also be gathered back into a single full permission for accounting. Two most popular permission systems are *fractional permissions* [18] and *counting permissions* [15].

In fractional permission system, permission is represented by a fractional number f. $f{=}1$ indicates a full permission while $0{<}f{<}1$ indicates a partial permission for read accesses. Given any fractional permission f where $0{<}f{\leq}1$, it is always possible to split f into two fractions f_1 and f_2 where $f_1{+}f_2{=}f$ and $f_1, f_2{>}0$, as follows:

$$x \overset{f}{\mapsto} E \wedge f{=}f_1{+}f_2 \wedge f_1{>}0 \wedge f_2{>}0 \implies x \overset{f_1}{\mapsto} E * x \overset{f_2}{\mapsto} E$$

This allows permissions to be split among concurrent threads. Threads having $0{<}f{<}1$ can safely read a shared location, while a thread having $f{=}1$ has exclusive access (either read or write) of the shared location. Permissions can also be combined to form an exclusive access, as follows:

$$x \overset{f_1}{\mapsto} E * x \overset{f_2}{\mapsto} E \implies x \overset{f_1+f_2}{\longmapsto} E$$

Similarly, in counting permission system, a total permission is written $x \overset{0}{\mapsto} E$ while a read permission is written $x \overset{-1}{\longmapsto} E$. Given a central *permission authority* holding a source permission n, it is always possible to split off into a new source permission $n{+}1$ (held by the central authority) and a read permission -1 for sharing:

$$x \overset{n}{\mapsto} E \wedge n{\geq}0 \iff x \overset{n-1}{\longmapsto} E * x \overset{-1}{\longmapsto} E$$

Fractional and counting permissions hence provide a means for permission accounting in concurrent separation logics, enable reasoning about race-free sharing of resources among concurrent threads. Recently, various permission systems such as binary tree share model [34], Plaid's permission system [11], and borrowing permissions [101] have been proposed. In a nutshell, they are akin to fractional and counting permissions.

2.1.4 Other Variants of Concurrent Separation Logic

Recent works have further improved concurrent separation logic (CSL). Brookes [20] showed that CSL is sound. The side conditions of parallel composition and conditional critical region rules can be removed if we treat stack variables as resource [16, 112]. Brookes showed that CSL with permissions and "variables as resource" is sound [19]. Additionally, there are many attempts to handle dynamic allocation of locks [51], dynamic creation of threads [51, 61], re-entrant locks [45, 52], and static barriers [62].

Although powerful, CSL and its variants have several limitations. First of all, it is only suitable for reasoning about well-synchronized concurrency. In well-synchronized programs, mutual exclusion is ensured in the critical regions. Therefore, it is unclear how to use CSL to reason about programs with ad hoc synchronizations [131]. In these programs, instead of using synchronization primitives, programmers use variables to synchronize in an ad hoc way. Second, similar to Owicki-Gries logic, CSL logic uses invariants to encode shared states; therefore, it also suffers from the preciseness of invariants as well as from the excessive use of auxiliary variables. Additionally, although CSL and its variants [16, 51, 52, 62] can guarantee race-freedom, they often ignore other properties such as deadlock freedom and correct barrier synchronization.

In summary, Owicki-Gries logic, CSL, and its variants focus on the assumption that threads are independent and hence they allow for local reasoning where threads can be verified independently from each other. Although the above logics are well-suited for verifying partial correctness and data-race freedom of shared-memory programs, they pay little attention to verification of other synchronization properties such as deadlock freedom and correct barrier synchronization.

2.2 Reasoning about Interference among Threads

In contrast to Owicki-Gries logic and CSL, which focus on independence among threads, Rely/Guarantee and its variants focus on specifying and verifying interference among threads.

2.2.1 Rely/Guarantee Reasoning

Rely/Guarantee reasoning (RG), proposed by Jones [69], is a well-established verification method for shared-memory concurrent programs. It is also known as Assume/Guarantee [39]. RG method uses binary relations of states (specifying state transitions) to describe interference among threads. A thread views all other threads in a program as its environment. The rely (or assume) condition specifies state transitions made by the environment; the guarantee condition specifies state transitions made by the current thread. A RG specification of a thread is formalized as follows:

$$R, G \ \vdash \{p\}\ C\ \{q\}$$

The specification states that given an execution of a thread C begins in a state satisfying the pre-condition p and an environment whose behaviors satisfy the rely condition R, then if any state transitions performed by the thread satisfy the guarantee condition G and the execution terminates, it will terminate in a state satisfying the post-condition q. Non-interference is guaranteed as long as the guarantee condition of each thread satisfies the rely conditions of all other threads, as described in the following rule for parallel composition $C_1||C_2$:

$$\frac{R \vee G_2, G_1 \ \vdash \{p\}\ C_1\ \{q_1\} \qquad R \vee G_1, G_2 \ \vdash \{p\}\ C_2\ \{q_2\}}{R, G_1 \vee G_2 \ \vdash \{p\}\ C_1 \ ||\ C_2\ \{q_1 \wedge q_2\}}$$

The rely and guarantee conditions of two threads ensure non-interference because they are compatible (the guarantee condition of thread C_1 satisfies the rely condition of thread C_2 and vice versa: $G_1 \Rightarrow R \vee G_1$ and $G_2 \Rightarrow R \vee G_2$). At the beginning, two threads begin in an initialized state satisfying the pre-condition p; at the end, if both threads terminate, both post-conditions hold. The total guarantee condition is $G_1 \vee G_2$ because the state transition belongs to either threads. RG method, therefore, is compositional in the sense that a thread is verified based on its own specification without knowing the code of other threads.

In contrast to CSL (Section 2.1.2) which is suitable for well-synchronized programs, RG reasoning is more general because it does not require specific language constructs for synchronization, which can be expressed in terms of rely and guarantee conditions. Therefore, it is capable of verifying programs using ad hoc synchronizations. However, RG is more complex because for each individual transition, we need to check that the state transition satisfies the guarantee condition. Additionally, RG is less memory-modular because it considers the entire memory as shared resources; therefore, it is usually hard to define global invariants for all these shared resources.

2.2.2 Other Variants

Due to the aforementioned limitations of Rely/Guarantee reasoning, Jones wanted a more compositional approach to verifying concurrent programs [70]. In response to Jones, RGSep [126], SAGL [39], LRG [38] and Deny/Guarantee reasoning [35], Concurrent Abstract Predicates [32], and Views [31], aim to achieve memory-modularity of separation logic without sacrificing RG's expressiveness. These approaches could achieve good modularity but are still limited to reasoning about partial correctness and data-race freedom, and mostly neglected the verification of synchronization properties such as deadlock freedom.

2.3 Automatic Verification Systems

In this section, we discuss state-of-the-art automated program verifiers which are based on the above-mentioned logics. While program logics attempt to reason locally and modularly, automatic verifiers are more concentrated on automation and expressiveness. Automation is a desirable feature to reduce human efforts, i.e. annotations. Expressiveness describes abilities of a verifier to capture various constructs used in real-world programs (such as concurrency and synchronization constructs) and to ensure properties of programs (e.g. data-race freedom and deadlock freedom).

2.3.1 Smalfoot

Smallfoot [9] is among the first verification tools based on concurrent separation logic (CSL). It has a symbolic execution engine [10] designed for a fixed set of shape predicates, including singly-, doubly-, and xor-linked lists and trees which are hard-wired into the system. It uses a complete decision procedure based on a collection of axioms (a.k.a lemmas) which are also hardwired into the system. Smallfoot uses CSL's parallel composition to enable concurrency and uses conditional critical regions (CCRs) for mutual exclusion among threads. Extending Smallfoot, Vafeiadis developed SmallfootRG [25] to support Rely/Guarantee reasoning based on RGSep.

Smallfoot is a very powerful verifier and requires less annotations; however, it can only operate on a fixed set of predicates. It does not support user-defined predicates which are essential to express users's intentions. Concurrency in Smallfoot is at the simplest form which is not popularly used in real world. Smallfoot does not support dynamical thread creation (e.g. via fork/join) as well as other synchronization constructs such as locks and barriers. In Smallfoot, every access to shared resources has to be done in critical regions, it limits concurrency in case of concurrent reads without any write (which can be handled using fractional permissions [18]). Although SmallfootRG can rely on the rely/guarantee conditions to allow concurrent reads, it is unclear how SmallfootRG can reason about dynamic creation of threads and resources. Additionally, by relying on separation logic, Smallfoot ensures data-race freedom in the presence of concurrent accesses to heap locations. For program variables, Smallfoot imposes side-conditions to prevent conflicting accesses to variables. However, these conditions are subtle and hard for compilers to check because it involves examining the entire program [16, 114].

2.3.2 Chalice

Chalice [4, 89, 90] is a program verifier for multi-threaded object-oriented programs developed at Microsoft. Its methodology is centered around implicit dynamic frame [119] (a variant of separation logic) and fractional permissions to express sharing and

non-sharing of objects and concurrent reads. Threads are created dynamically using fork/join primitives. Accesses to shared objects are synchronized via acquisition and release of monitors. Among existing verification systems, Chalice is the only system that supports verification of deadlock freedom. Chalice prevents certain types of deadlocks by allowing users to annotate a locking order associated with each monitor. Unfortunately, there are other types of deadlocks that Chalice is not designed to handle (as we will elaborate more in Chapter 4). Additionally, similar to Smallfoot, Chalice does not support other synchronization constructs such as barriers.

2.3.3 Verifast

Verifast [67] is a verifier for C-like and Java programs. It is based on separation logic; it focuses on error detection capability and expressiveness rather than on automation. Verifast is able to report illegal memory accesses as well as data races. Verifast allows users to define deep data structures via shape predicates. Verifast also supports lemma functions to prove properties of data structures as well as provide alternative ways to traverse data structures. In order to verify concurrent programs, Verifast adopts programming language and logic of Gotsman et al. [51], a variant of concurrent separation logic which supports verification of dynamic creation of locks and threads. The main disadvantage of Verifast is its high overhead for annotations, reportedly in the order of 10 to 20 lines of annotation per line of code [65]. In addition, similar to Smallfoot, Verifast focuses on verifying partial correctness and data-race freedom, and almost ignores synchronization properties such as deadlock freedom and correct barrier synchronization.

2.4 Open Issues

In previous sections, we discussed recent advances in reasoning about shared-memory concurrent programs. Despite those recent advances, there remain challenging open issues. This section introduces briefly the issues that are important and which will be addressed in subsequent chapters of this thesis.

2.4.1 Reasoning about First-class Threads

Most of the above existing works often focus on the theoretical parallel composition. However, in practice, mainstream languages such as Java, C#, and C/C++ provide fork/join constructs [98] for dynamic creation and termination of threads. In these languages, threads are considered as first-class citizens in that threads can be treated like objects of any other type: they can be dynamically created, stored in data structures, shared among different threads, passed as parameters, and returned from procedures. Therefore, it is desirable for verification systems to support reasoning about first-class threads.

There exist approaches that can handle dynamically-allocated threads using fork/join, e.g. [51, 60, 65, 91]. Hobor [60] allows threads to be dynamically created using fork but does not support join. Gotsman et al. [51] use thread handles to represent threads, while Chalice [91] uses tokens, and Verifast [65] uses thread permissions. A fork operation returns a unique handle/token/permission (collectively referred to as thread token) and a join operation on a thread token causes the joining thread to wait for the completion of the thread corresponding to the token. However, these current works [51, 60, 65, 91] support reasoning about threads in a limited way: unique tokens (representing threads) are not allowed to be split and shared among different threads. As such, they are unable to verify more complicated programs, e.g. those where threads are shared and joined in different threads. In other words, existing works do not fully consider threads as first-class. Therefore, there is a need for more expressive reasoning about partial correctness of programs with first-class threads.

2.4.2 Reasoning about Synchronization Properties

Beside partial correctness, it is also important to be able to reason about other desirable properties of concurrent programs. In the context of shared-memory programs, the desirable properties include synchronization properties such as deadlock freedom of programs with locks, and correct synchronization of programs with barriers.

2.4.2.1 Verifying Deadlock Freedom

Deadlocks are defined as states in which each thread in a set of threads blocks waiting for another thread in the set to release a lock or complete its execution and neither ever do so [27]. Deadlocks are common defects in software systems. Specifically, in Sun's bug report database at `http://bugs.sun.com/`, there are approximately 6,500 bug reports out of 198,000 ($\sim$ 3%) containing the keyword "deadlock" [102]. Hence, it is desirable to be able to verify that programs are deadlock-free.

There exist current works that are able to reason about programs with non-recursive locks and dynamically-created threads [51, 61], recursive locks [45, 52], and low-level languages [45]. However, they focus on verifying partial correctness and ignore the presence of deadlocks. Haack et al. [52] use lockbags when verifying partial correctness of concurrent programs manipulating Java recursive locks. However, their approach in [52] as well as their subsequent work in VerCors project [12] do not ensure deadlock freedom. Verifast [65] also ignores deadlocks when verifying correctness of concurrent programs. Chalice [90, 91] is the only verifier that is able to verify some types of deadlock freedom. Unfortunately, it is still limited since it is not designed to verify deadlock freedom of programs with intricate interactions between thread operations (e.g. fork/join) and lock operations (e.g. acquire/release). The desire for a comprehensive framework for verifying deadlock freedom remains.

2.4.2.2 Verifying Barrier Synchronization

Beside locks, barriers are commonly used in practice [13, 107]. Threads synchronizing on a barrier proceed in *phases*. When a thread issues a barrier wait, it waits until a pre-defined number of threads (all threads or just a group of threads) have also issued a barrier wait; after that, all participating threads proceed to the next phase. In Pthreads [2], barriers are *static*, i.e. the number of participants is fixed. In .NET framework [43], barriers are *dynamic* as the number of participants can vary during a program's execution. The java.util.concurrent library [49] supports both static and dynamic barriers (i.e. CyclicBarrier and Phaser respectively). On the one hand,

incorrect synchronization on a barrier could lead to blocking states (a.k.a deadlocks). On the other hand, verifying correct synchronization of barriers can provide compilers and analysers with important phasing information for improving the precision of their analyses and optimizations such as reducing false sharing [68], may-happen-in-parallel analysis [93, 134], and data race detection [76].

Many works have been proposed to verify correct synchronization of barriers. However, most of them focus on SPMD programs [5, 68, 76, 77, 93, 133, 134]. Threads in SPMD programs execute the same code hence, the verification is more tractable. Threads in SMPD programs also assume that barriers are global and all threads need to participate in barrier operations. As such, existing techniques for SPMD programs cannot be directly applied to fork/join programs where threads are dynamically-created and non-lexically-scoped. In the context of fork/join programs, we are only aware of the work by Hobor and Gherghina [63]. In this work, they propose a specification logic focusing on verifying partial correctness of programs with static barriers. Nonetheless, they are unable to verify correct synchronization of programs with dynamic barriers. Given the importance of barriers in practice, it is tempting to be able to verify correct synchronization of programs using both static and dynamic barriers.

2.5 Summary

This chapter presented an overview of existing approaches and systems for specifying and verifying shared-memory concurrent programs. Challenging open issues were also discussed. This chapter is by no means a complete reference to all existing works, but it aims to motivate the fact that existing works mostly focus on partial correctness and data-race freedom of programs with simplistic concurrency constructs (such as the theoretical parallel composition and conditional critical regions), and often ignore the verification of synchronization properties such as deadlock freedom and barrier synchronization. As we believe verifying partial correctness and data-race freedom, and ensuring the synchronization properties are of equal importance, this thesis aims

to propose methodologies not only to verify partial correctness and data-race freedom of concurrent programs with realistic concurrency constructs such as fork/join, locks, and barriers, but also to ensure their synchronization properties. Our methodologies will be presented in the following chapters.

Chapter 3

Threads as Resource

Overview. Threads are considered as first-class in mainstream languages such as Java, C#, and C/C++ in that threads can be treated like objects of any other type: they can be dynamically created, stored in data structures, shared among different threads, passed as parameters, and returned from procedures. Hence, it is desirable for verification systems to support reasoning about first-class threads.

One of the most popular techniques for reasoning about concurrent programs is separation logic [105, 115]. Originally, separation logic was used to verify heap-manipulating sequential programs, with the ability to express non-aliasing in the heap [115]. Separation logic was extended to verify shared-memory concurrent programs, e.g. concurrent separation logic [105], where ownerships of heap objects are considered as *resource*, which can be shared and transferred among concurrent threads. Using fractional permissions [18], one can express full ownerships for exclusive write accesses and partial ownerships for concurrent read accesses. Ownerships of stack variables can also be considered as resource and treated in the same way as heap objects [16].

Separation logic was traditionally extended to verify concurrent programs with parallel composition [105]. Recent works also extended separation logic to handle dynamically-created threads [51, 60, 65, 91]. Hobor [60] allows threads to be dynamically created using fork but does not support join. Gotsman et al. [51] use thread

handles to represent threads, while CHALICE [91] uses tokens, and VERIFAST [65] uses thread permissions for the same purpose. A fork operation returns a unique handle/-token/permission (collectively referred to as thread token) and a join operation on a thread token causes the joining thread (joiner) to wait for the completion of the thread corresponding to the token (joinee). However, existing works [51, 60, 65, 91] support reasoning about threads in a limited way: unique tokens (representing threads) are not allowed to be split and shared among different threads. As such, existing works do not fully consider threads as first-class.

Reasoning about first-class threads is challenging because threads are dynamic and non-lexically-scoped in nature. A thread can be dynamically created in a procedure (or a thread), but shared and joined in other procedures (or threads). In this chapter, we propose an expressive treatment of first-class threads, called "*threads as resource*". Our approach enables threads' ownerships to be reasoned about in a similar way to other types of resource. A thread's ownership is created when it is forked, and destroyed when it is joined. In contrast to ownership of a normal heap object which specifies values of its fields, ownership of a thread carries resource that can be obtained by the joiner when the thread is joined. This is to cater for the intuition that when a joiner joins with a joinee, the joiner expects to obtain (in order to later read or write) certain resource transferred from the joinee. As threads in fork/join programs are typically non-lexically-scoped, we allow threads' ownerships to be soundly split, combined, and (possibly partially) transferred among procedures and threads.

Our approach elegantly solves at least three verification problems that were not properly supported. First, threads can now be passed as arguments, shared, and joined by different threads. This enables verification of intricate fork/join behaviors such as *multi-join pattern* where a thread is shared and joined in multiple threads. Using our approach, the ownership of the joinee (and its resource) can be split and transferred (or shared) among the multiple joiners, so that they can respectively join with the joinee and get their corresponding portions of the joinee's resource. Second, by treating threads in a similar way to heap objects, we can apply current advances in

separation logic for heap objects to threads. For example, by combining "threads as resource" with inductive predicates, we can naturally capture a programming idiom called threadpool where threads are stored in data structures. Lastly, we can formally reason about the "liveness" of a thread. We achieve this by adding a special predicate that explicitly indicates when a thread is dead (i.e. after it is joined). Our approach has been implemented in a tool, and experimental results showed reasonable verification performance. Lastly, the notion of "threads as resource" has recently inspired us to propose "*flow-aware resource predicate*", a variant of Concurrent Abstract Predicates (CAP) [33, 36, 121], for more expressive verification of not only first-class threads but also other concurrency mechanisms such as `countDownLatch` and copyless multicast communication.

The rest of this chapter is organized as follows. Section 3.1 motivates our idea of "threads as resource". Section 3.2 introduces our proposed approach. Section 3.3 presents our prototype implementation and experimental results. Section 3.4 discusses the use of flow-aware resource predicates. Section 3.5 summarizes related work. Section 3.6 concludes this chapter.

3.1 A Motivating Example

This section illustrates our treatment of "threads as resource" for reasoning about programs with first-class threads. Fig. 3-1 shows a C-like program posing challenges to existing verification systems. In the program, the main thread executing the procedure `main` (called `main` thread) forks a new thread `t1` executing the procedure `thread1` (line 22). `thread1` will swap the values of the cells `x` and `y`. `main` then forks another thread `t2` executing the procedure `thread2` with `t1` passed as one of its arguments (line 25). Afterward, `t2` will join with `t1` (line 12) and manipulate the cell `y`, while `main` will also join with `t1` (line 27) but manipulate the cell `x`. In separation logic, a heap node $x \mapsto cell(vx)$ represents the ownership of an object of type *cell* pointed to by x and having the field `val` of vx (called ownership of x for short).

```
1  data cell { int val; }
2
3  void  thread1(cell x, cell y)
4    requires  x ↦ cell(vx) * y ↦ cell(vy)
5    ensures  x ↦ cell(vy) * y ↦ cell(vx);
6  { int tmp = x.val;   x.val = y.val;   y.val = tmp; }
7
8  void  thread2(thrd t1, cell y)
9    requires  t1 ↦ thrd⟨y ↦ cell(vy)⟩
10   ensures  y ↦ cell(vy + 2) ∧ dead(t1);
11 { // {t1 ↦ thrd⟨y ↦ cell(vy)⟩}
12   join(t1);
13   // {y ↦ cell(vy) ∧ dead(t1)}
14   y.val = y.val+2;
15   // {y ↦ cell(vy + 2) ∧ dead(t1)}
16 }
17
18 void  main()
19   requires  emp  ensures  emp;
20 { cell x = new cell(1);   cell y = new cell(2);
21   // {x ↦ cell(1) * y ↦ cell(2)}
22   thrd t1 = fork(thread1,x,y);
23   // {t1 ↦ thrd⟨x ↦ cell(2) * y ↦ cell(1)⟩}
24   // {t1 ↦ thrd⟨x ↦ cell(2)⟩ * t1 ↦ thrd⟨y ↦ cell(1)⟩}
25   thrd t2 = fork(thread2,t1,y);
26   // {t1 ↦ thrd⟨x ↦ cell(2)⟩ * t2 ↦ thrd⟨y ↦ cell(3) ∧ dead(t1)⟩}
27   join(t1);
28   // {x ↦ cell(2) * t2 ↦ thrd⟨y ↦ cell(3) ∧ dead(t1)⟩ ∧ dead(t1)}
29   x.val = x.val+1;
30   // {x ↦ cell(3) * t2 ↦ thrd⟨y ↦ cell(3) ∧ dead(t1)⟩ ∧ dead(t1)}
31   join(t2);
32   // {x ↦ cell(3) * y ↦ cell(3) ∧ dead(t1) ∧ dead(t2)}
33   assert(x ↦ cell(3) * y ↦ cell(3)); /*valid*/
34   destroy(x); destroy(y);
35   // {emp ∧ dead(t1) ∧ dead(t2)}
36 }
```

Figure 3-1: A Motivating Example

The program is challenging to verify because (1) fork and join operations on `t1` are non-lexically scoped (i.e. `t1` is forked in `main` but joined in thread `t2`), and (2) `t1` is shared and joined in both `t2` and `main` (i.e. a multi-join). In this program, the ownerships of `x` and `y` are flexibly transferred across thread boundaries, between `main`, `t1` and `t2`, via fork/join calls. To the best of our knowledge, we are not aware of any existing approaches capable of verifying this program. We propose "*threads as resource*" to verify such programs soundly and modularly. The key points to handle this program are (1) considering `t1` as resource, and (2) allowing it to be split and transferred between `main` and `t2` via fork/join calls.

Our approach is based on the following observation: when a thread (joiner) joins with another thread (joinee), the joiner expects to receive (in order to later read or write) certain resource transferred from the joinee. In the example program, `main` joins with `t1` and expects the ownership of `x` transferred from `t1`, while `t2` joins with `t1` and expects the ownership of `y`. Hence, the verification of the program in Fig. 3-1 is achieved by introducing the thread ownership $v \mapsto \mathsf{thrd}\langle\Phi\rangle$ indicating that v points to a live thread (as resource) carrying certain resource Φ. A thread having the ownership $v \mapsto \mathsf{thrd}\langle\Phi\rangle$ can perform a `join(`v`)`, and yield the resource Φ and a pure predicate $dead(v)$ after joining. This special predicate $dead(v)$ explicitly indicates that thread v is no longer alive. In Fig. 3-1, when `t1` is forked (line 22), its pre-condition is consumed and exchanged for the thread's ownership $t1 \mapsto \mathsf{thrd}\langle x \mapsto cell(2) * y \mapsto cell(1)\rangle$ carrying the post-state of `thread1` (i.e. `t1`'s state after it has finished its execution). This is sound and modular as other threads can only observe the post-state of `thread1` when they join with `t1`. Our approach enables the thread's ownership to be split into $t1 \mapsto \mathsf{thrd}\langle x \mapsto cell(2)\rangle$ and $t1 \mapsto \mathsf{thrd}\langle y \mapsto cell(1)\rangle$ (from line 23 to line 24). This allows the latter to be transferred to `t2` while the former remains with `main`. Consequently, having the ownerships of `t1`, both `t2` and `main` can perform `join(t1)` and get the corresponding resource: `t2` obtains the ownership of `y` to write to it, while `main` obtains and writes to `x` (i.e. `t2` and `main` write-share the resource transferred from `t1`). Using our "threads as resource" approach, the program can be verified as

both data-race-free and partially correct.

Our treatment of "threads as resource" allows the ownership of a thread to be flexibly split and transferred. For example, in a program similar to Fig. 3-1, instead of writing to cells `x` and `y`, both `main` and t2 may want to concurrently read the value of the cells. Using fractional permissions [18], we could now split the ownership of `t1` from $t1 \mapsto \mathsf{thrd}\langle x \mapsto cell(2) * y \mapsto cell(1)\rangle$, into $t1 \mapsto \mathsf{thrd}\langle x \overset{0.6}{\longmapsto} cell(2) * y \overset{0.6}{\longmapsto} cell(1)\rangle$ and $t1 \mapsto \mathsf{thrd}\langle x \overset{0.4}{\longmapsto} cell(2) * y \overset{0.4}{\longmapsto} cell(1)\rangle$, and transfer them into the corresponding codes for `main` and `t2`. This allows `main` and `t2` to be able to read concurrently cells `x` and `y` after joining with the `t1` thread.

In summary, we propose to treat threads as resource, thus allowing threads' ownerships to be soundly split and transferred across procedure and thread boundaries. This supports first-class threads and enables modular reasoning of intricate concurrent programs with non-lexically-scoped fork/join and multi-join. We will give more details in the rest of this chapter.

3.2 Proposed Approach

3.2.1 Programming Language

We use the core programming language in Fig. 3-2 to convey our idea. A program consists of data declarations ($data\_decl^*$), global variable declarations ($global\_decl^*$), and procedure declarations ($proc\_decl^*$). Each procedure declaration is annotated with pairs of pre/post-conditions (Φ_{pr}/Φ_{po}). New objects of type C can be dynamically created and destroyed using **new** and **destroy**. A **fork** receives a procedure name pn and a list of parameters v^*, creates a new thread executing the procedure pn, and returns an object of `thrd` type representing the newly-created thread. **join**(v) waits for the thread that is pointed to by v to finish its execution. Note that a joinee could be joined in multiple joiners. At run-time, the joiners wait for the joinee to complete its execution. If a joiner waits for an already-completed (or dead) thread, it proceeds immediately without waiting (i.e. the join operation becomes no-op). We

P	::=	$data\_decl^*$ $global\_decl^*$ $proc\_decl^*$	Program
$data\_decl$	::=	**data** C { $field\_decl^*$ }	Data declaration
$field\_decl$	::=	$type\ f$;	Field declaration
$global\_decl$	::=	**global** $type\ v$	Global variable declaration
$proc\_decl$	::=	$type\ pn(param^*)$ $spec^*$ { s }	Procedure declaration
$spec$	::=	**requires** Φ_{pr} **ensures** Φ_{po};	Pre/Post-conditions
$param$	::=	$type\ v$	Parameter
$type$	::=	`int` | `bool` | `void` | `thrd` | C	Type
e	::=	$v \mid v.f \mid k \mid e_1{+}e_2 \mid e_1{=}e_2 \mid e_1{\neq}e_2$	Var/field/const/expr
s	::=	$v = \mathbf{new}\, C(v^*) \mid \mathbf{destroy}(v)$ $\mid\ v = \mathbf{fork}(pn, v^*) \mid \mathbf{join}(v)$ $\mid\ \mathbf{if}\ e\ \mathbf{then}\ s_1\ \mathbf{else}\ s_2$ $\mid\ s_1; s_2 \mid pn(v^*) \mid \ldots$	Statement

Figure 3-2: Core Programming Language with First-Class Threads

do not allow canceling a thread. A thread is dead after it is joined or when the entire program has finished its execution. The semantics of other program statements (such as procedure calls $pn(v^*)$, conditionals, loops, assignments) are standard as can be found in the mainstream languages.

3.2.2 Specification Language

Fig. 3-3 shows our specification language for concurrent programs manipulating "threads as resource". A classical separation logic formula Φ is in disjunctive normal form. Each disjunct in Φ consists of a heap formula κ and a pure formula π. Furthermore, Δ denotes a composite formula which could always be translated into the Φ form. A pure formula π includes standard equality/inequality, Presburger arithmetic, and a pure predicate $dead(v)$ indicating that the thread v has completed its execution. π could also be extended to include other constraints such as set constraints. A heap formula κ consists of multiple atomic heap formulae ι connected with each other via the separation connective $*$. An atomic heap formula $v \overset{\varepsilon}{\mapsto} C(v^*)$ (or heap node) represents the fact that the current thread has a certain fractional permission ε to

$$
\begin{array}{rrcl}
\text{Separation formula} & \Phi & ::= & \bigvee(\exists v^* \cdot \kappa \wedge \pi) \\
\text{Composite formula} & \Delta & ::= & \Phi \mid \Delta_1 \vee \Delta_2 \mid \Delta_1 \wedge \pi \mid \Delta_1 * \Delta_2 \mid \exists v \cdot \Delta \\
\text{Heap formula} & \kappa & ::= & \mathtt{emp} \mid \iota \mid \kappa_1 * \kappa_2 \\
\text{Atomic heap formula} & \iota & ::= & v \overset{\varepsilon}{\mapsto} C(v^*) \mid v \mapsto \mathsf{thrd}\langle\Phi\rangle \\
\text{Pure formula} & \pi & ::= & \alpha \mid \pi_1 \wedge \pi_2 \mid \pi_1 \vee \pi_2 \mid \neg\pi \\
 & & & \mid \exists v \cdot \pi \mid \forall v \cdot \pi \mid \mathit{dead}(v) \\
\text{Arithmetic formula} & \alpha & ::= & \alpha_1^t = \alpha_2^t \mid \alpha_1^t \neq \alpha_2^t \mid \alpha_1^t < \alpha_2^t \mid \alpha_1^t \leq \alpha_2^t \\
\text{Arithmetic term} & \alpha^t & ::= & k \mid v \mid k \times \alpha^t \mid \alpha_1^t + \alpha_2^t \mid -\alpha^t
\end{array}
$$

Fractional permission variable $\varepsilon \in (0,1]$ $\quad$ $v \in$ Variables

$k \in$ Integer or fractional constants $\quad$ $C \in$ Data names

Figure 3-3: Grammar for Core Specification Language

access an object of type C pointed to by v. v^* captures a list of variables representing the fields of the object v.

The atomic heap formula $v \mapsto \mathsf{thrd}\langle\Phi\rangle$ (or thread node) captures our idea of "threads as resource": v points to a thread carrying certain resource Φ, which is available after the thread is joined. By representing threads as heap resource, we allow them to be flexibly split and transferred in a similar way to other types of resource such as heap nodes. Note that thread nodes themselves are non-fractional, but their resources can already be flexibly split. Furthermore, no resource leakage from threads is possible since we explicitly track when each thread becomes dead.

Our approach allows for expressive reasoning about threads and their liveness. For example, a formula $t \mapsto \mathsf{thrd}\langle\Phi\rangle \bigvee \mathit{dead}(t)$ specifies the fact that the thread t could be either alive or dead. On the other hand, a formula with $t \mapsto \mathsf{thrd}\langle\Phi\rangle \wedge \mathit{dead}(t)$ indicates the fact that t is already dead and hence the resource Φ can be safely released.

3.2.3 Forward Verification Rules

Our verification system is built on top of entailment checking:

$$\Delta_A \vdash \Delta_C \rightsquigarrow \Delta_R$$

This entailment checks if antecedent Δ_A is precise enough to imply consequent Δ_C, and computes the residue Δ_R for the next program state. For example:

$$x \overset{0.6}{\longmapsto} cell(1) * y \overset{0.6}{\longmapsto} cell(2) \vdash x \overset{0.6}{\longmapsto} cell(1) \rightsquigarrow y \overset{0.6}{\longmapsto} cell(2)$$

Fig. 3-4 presents our forward verification rules. Here we only focus on three key constructs affecting threads' resource: procedure call, fork, and join. Forward verification is formalized using Hoare's triple for partial correctness: $\{\Phi_{pr}\}P\{\Phi_{po}\}$. Given a program P starting in a state satisfying the pre-condition Φ_{pr}, if the program terminates, it will do so in a state satisfying the post-condition Φ_{po}. For simplicity, in this thesis, we describe the verification rules with one pair of pre/post condition. Multiple pre/post specifications can be handled in the same way as [26]. We also omit the treatment of pass-by-reference parameters whose side-effects can be captured by applying permissions to program variables (see Appendix A for such a mechanism).

In order to perform a procedure call (**CALL**), the caller should be in a state Δ that can entail the pre-condition Φ_{pr} of the callee (i.e the procedure pn). $spec(pn)$ denotes the specification of the procedure pn. For conciseness, we omit the substitutions that link actual and formal parameters of the procedure prior to the entailment. After the

$$\frac{spec(pn) := \texttt{pn}(w^*)\ \textbf{requires}\ \Phi_{pr}\ \textbf{ensures}\ \Phi_{po};\ \{\ s\ \} \quad \Delta \vdash \Phi_{pr} \rightsquigarrow \Delta_1 \quad \Delta_2 \overset{\texttt{def}}{=} \Delta_1 * \Phi_{po}}{\{\Delta\}\ pn(w^*)\ \{\Delta_2\}} \quad \textbf{CALL}$$

$$\frac{spec(pn) := \texttt{pn}(w^*)\ \textbf{requires}\ \Phi_{pr}\ \textbf{ensures}\ \Phi_{po};\ \{\ s\ \} \quad \Delta \vdash \Phi_{pr} \rightsquigarrow \Delta_1 \quad \Delta_2 \overset{\texttt{def}}{=} \Delta_1 * v \mapsto \mathsf{thrd}\langle\Phi_{po}\rangle}{\{\Delta\}\ v := \textbf{fork}(pn,w^*)\ \{\Delta_2\}} \quad \textbf{FORK}$$

$$\frac{\Delta_2 \overset{\texttt{def}}{=} \Delta * \Phi_{po} \wedge dead(v)}{\{\Delta * v \mapsto \mathsf{thrd}\langle\Phi_{po}\rangle\}\ \textbf{join}(v)\ \{\Delta_2\}} \quad \textbf{JOIN–1}$$

$$\{\Delta \wedge dead(v)\}\ \textbf{join}(v)\ \{\Delta \wedge dead(v)\} \quad \textbf{JOIN–2}$$

Figure 3-4: Selected Verification Rules

entailment, the caller subsumes the post-condition Φ_{po} of the callee with the residue Δ_1 to form a new state Δ_2. Ownerships are transferred across procedure boundaries, from the caller to the callee via the entailment of the pre-condition and from the callee to the caller via the spatial conjunction on the post-condition.

Similarly, when performing a fork (**FORK**), the forker should be in a state Δ that can entail the pre-condition Φ_{pr} of the forkee (i.e the newly-created thread executing the procedure pn). Afterward, a new thread node $v \mapsto \mathsf{thrd}\langle\Phi_{po}\rangle$ carrying the post-condition Φ_{po} of the forkee is created. The thread node is then combined with the residue Δ_1 to form a new state Δ_2. The thread node is considered as resource in Δ_2; hence, it can be flexibly split and transferred in subsequent parts of the program. The **FORK** rule is sound since other threads can only observe the post-state of the forkee when joining with it. It also ensures modularity as the forker only knows the pre/post-conditions of the forkee.

When joining a thread (**JOIN–1**), the joiner simply exchanges the thread node, which carries a resource Φ_{po}, with the resource itself. Each joinee could be joined by multiple joiners. Our verification rules are based on the observation that when a joiner joins with a joinee, the joiner is expecting to receive certain resource transferred to it from the joinee. Hence, each joiner will receive the current resource carried by the thread node. After a thread has been joined, it becomes dead (indicated by the pure *dead* predicate). Joining a dead thread is equivalent to a no-op (**JOIN–2**).

Using our verification rules, a **CALL** can be modeled as a **FORK** immediately followed by a **JOIN**. As threads are considered as resource, fork and join operations can be in different lexical scopes and thread nodes can be transferred across procedure and thread boundaries. Furthermore, if there is a recursive fork call in a procedure (also called nested fork) such as the parallel Fibonacci program[1], the verification proceeds normally: a new thread node corresponding to the newly-created thread executing the procedure is created. Therefore, in our system, a nested fork is handled in the same way as a normal fork.

[1]Available on `http://loris-7.ddns.comp.nus.edu.sg/~project/threadhip/`

$$\frac{\Phi_1 \Longleftrightarrow \Phi_1^{'} \qquad \Phi_2 \Longleftrightarrow \Phi_2^{'}}{\Phi_1 \bigvee \Phi_2 \Longleftrightarrow \Phi_1^{'} \bigvee \Phi_2^{'}} \quad \textbf{R–DISJ}$$

$$\frac{\kappa \Longleftrightarrow \kappa^{'}}{\kappa \wedge \pi \Longleftrightarrow \kappa^{'} \wedge \pi} \quad \textbf{R–CONJ}$$

$$\frac{\kappa_1 \Longleftrightarrow \kappa_1^{'} \qquad \kappa_2 \Longleftrightarrow \kappa_2^{'}}{\kappa_1 * \kappa_2 \Longleftrightarrow \kappa_1^{'} * \kappa_2^{'}} \quad \textbf{R–SCONJ}$$

$$\kappa_1 * \kappa_2 \Longleftrightarrow \kappa_2 * \kappa_1 \quad \textbf{R–COM}$$

$$\kappa * \mathsf{emp} \Longleftrightarrow \kappa \quad \textbf{R–EMP}$$

$$v \xmapsto{\varepsilon_1+\varepsilon_2} C(v^*) \Longleftrightarrow v \xmapsto{\varepsilon_1} C(v^*) * v \xmapsto{\varepsilon_2} C(v^*) \quad \textbf{R–FRAC}$$

$$v \mapsto \mathsf{thrd}\langle \Phi_1 * \Phi_2 \rangle \Longleftrightarrow v \mapsto \mathsf{thrd}\langle \Phi_1 \rangle * v \mapsto \mathsf{thrd}\langle \Phi_2 \rangle \quad \textbf{R–THRD1}$$

$$v \mapsto \mathsf{thrd}\langle \Phi \rangle \wedge \mathit{dead}(v) \Longrightarrow \Phi \quad \textbf{R–THRD2}$$

Figure 3-5: Sub-structural Rules

3.2.4 Manipulating "Threads as Resource"

The notion of "threads as resource" plays a critical role in our approach as it enables threads to be treated in a similar way to other objects: a thread node can be created, stored, split, and transferred (or shared) among multiple threads, allowing them to join and to receive suitable resource after joining.

Our sub-structural rules for manipulating resource are presented in Fig. 3-5. The rules rearrange resource in a separation logic formula into equivalent forms. We denote resource equivalence as $\Longleftrightarrow$. By resource equivalence, we mean that the total resource on the left and the right sides of $\Longleftrightarrow$ are the same. Our approach allows resource to be split, combined, and transferred across procedures and threads, while it guarantees that the total resource remains unchanged. The rules **R–DISJ**, **R–CONJ**, **R–SCONJ**, **R–COM**, and **R–EMP** are straightforward. With fractional permissions ε, heap nodes can be split and combined in a standard way (**R–FRAC**). The left-to-right direction indicates permission splitting while the right-to-left indicates permission combining.

We also allow thread nodes to be split and combined (**R–THRD1**). Splitting a thread node (left-to-right) will split the resource carried by the node while combining thread nodes (right-to-left) will combine the resource of the constituent nodes. Finally, when a thread is dead, its carried resource can be safely released (**R–THRD2**).

Soundness. Our "threads as resource" approach allows for sound resource transfer among threads. Furthermore, the soundness of a permission-based resource logic must ensure that the total number of permissions on each heap object never exceeds the full permission (i.e. 1 in the fractional permission system [18]). At any time, at most one thread can write to a heap object, and when a thread has a read permission to a heap object, all other threads similarly hold read permissions as well. This ensures that verified programs are data-race free. To guarantee soundness, we show that our approach neither invents new resource nor destroys existing resource. Moreover, it guarantees that the total resource of the program is not changed by our verification rules. We now state the main soundness lemma, details are given in Appendix B.

Lemma 1 (Soundness of Threads as Resource)**.** *Given a program with a set of procedures P^i together with their corresponding pre/post-conditions (Φ^i_{pr}/Φ^i_{po}), if our verifier derives a proof for every procedure P^i, i.e. $\{\Phi^i_{pr}\}P^i\{\Phi^i_{po}\}$ is valid, then the program is race-free and partially correct.*

3.2.5 Applications

Verifying the Multi-join Pattern

A program with multi-join pattern allows a thread (joinee) to be shared and joined in multiple threads (joiners). During the program's execution, the joiners wait for the joinee to finish its execution. If joiners wait for an already-completed joinee, they proceed immediately without waiting. By joining with the joinee, the joiners expect to receive certain resource transferred from the joinee. The program in Fig. 3-1 is an example of such a multi-join pattern. As we have shown in previous sections, our approach handles the multi-join pattern naturally. Our approach allows the

```
1  data node { int val; node next; }
2  data list { node head; }
3  data count { int val; }
4  self ↦ ll(n) ≝ self=null ∧ n=0
5   ⋁ ∃q · self ↦ node(_, q) * q ↦ ll(n−1)
6   inv n≥0;
8  void  countList(list l)
9    requires  l ↦ list(h) * h ↦ ll(n) ∧ n≥0
10   ensures  l ↦ list(h) * h ↦ ll(n) ∧ res=n;
11 { ...}
12 list  createList(int n)
13   requires  n≥0
14   ensures  res ↦ list(h) * h ↦ ll(n);
15 { ...}
16 list  destroyList(list l)
17   requires  l ↦ list(h) * h ↦ ll(n)
18   ensures  emp;
19 { ...}
20 void  mapper(list l, list o, list e)
21   requires l ↦ list(h) * h ↦ ll(n) * o ↦ list(null) * e ↦ list(null)
22   ensures o ↦ list(oh) * oh ↦ ll(n1) * e ↦ list(eh) * eh ↦ ll(n2) ∧ n=n1+n2;
23 { ...}
24 void  reducer(thrd m, list l, count c)
25   requires  m ↦ thrd⟨l ↦ list(h) * h ↦ ll(n)∧n≥0⟩ * c ↦ count(_)
26   ensures  l ↦ list(h) * h ↦ ll(n) * c ↦ count(n) ∧ dead(m);
27 { join(m); /*multi-joined by the two reducers*/
28   c.val = countList(l); }
29 void  main()
30   requires emp ensures emp;
31 { int n = 10000;  list l = createList(n);
32   list ol = new list(null); list el = new list(null);
33   count c1 = new count(0); count c2 = new count(0);
34   /*fork mapper/reducer threads*/
35   thrd m = fork(mapper,l,ol,el);
36   thrd r1 = fork(reducer,m,ol,c1);
37   thrd r2 = fork(reducer,m,el,c2);
38   /*wait for them to finish*/
39   join(r1); join(r2);
40   assert(c1.val + c2.val = n); /*valid*/
41   destroyList(ol); destroyList(el);
42   destroy(c1); destroy(c2);
43 }
```

Figure 3-6: Map/Reduce using Multi-join

ownership of the joinee to be split, shared, and joined by multiple joiners, where each joiner obtains their corresponding part of the joinee's resource upon join.

We now illustrate another example of multi-join concurrency pattern in Fig. 3-6, based on the map/reduce paradigm. In this program, the `main` thread concurrently forks three threads: a mapper `m` to produce two lists, and two reducers `r1` and `r2` to process a list each. Both the reducers each take `m` as a parameter and joins it at an appropriate place to recover their respective lists from `m`. The main thread subsequently joins up the two reducers before completing its execution. This multi-join program is challenging to verify because (i) fork and join operations on the mapper `m` are non-lexically scoped (i.e. `m` is forked in `main` but joined in threads `r1` and `r2`), and (2) part of the computed resources from `m` is made available to `r1`, while another part is made available to `r2`. In this program, the ownerships of two lists produced by the mapper must be flexibly transferred across thread boundaries, via fork/join calls. The key points to handle this program are (1) considering the executing thread of `m` as resource, and (2) allowing it to be split and transferred between `main`, `r1` and `r2` via fork/join calls. Using our approach, the program can be verified as both data-race-free and partially correct.

Inductive Predicates and Threads as Resource

Modeling threads as resource open opportunities for applying current advances in separation logic, which were originally designed for heap objects, to threads. In this section, we describe how "threads as resource" together with inductive predicates [48, 103] can be used to naturally capture a simple threadpool, where threads are stored in data structures.

An example program is presented in Fig. 3-7. The program receives an input `n`, and then invokes `forkThreads` to create `n` concurrent threads executing the procedure `thread`. For simplicity, we assume each thread will have a read permission of the cell `x` in the pre-condition and will return the read permission in the post-condition. The program will wait for all threads to finish their execution by invoking `joinThreads`.

```
1  data cell { int val; }
2  data item { thrd t; item next; }
3
4  int input() requires emp ensures res>0;
5
6  void thread(cell x,int M)
7    requires x ⟼^{1/M} cell(_) ∧ M>0 ensures x ⟼^{1/M} cell(_);
8
9  item forkHelper(cell x, int n, int M)
10   case { n = 0 → requires emp ensures emp ∧ res = null;
11        n > 0 → requires x ⟼^{n/M} cell(_) ∧ M≥n
12                ensures res ↦ pool(x, n, M); }
13 { if (n==0){ return null;} else {
14     thrd t = fork(thread,x,M);
15     item p = forkHelper(x,n-1,M);
16     item i = new item(t,p);
17     return i; }
18
19 item forkThreads(cell x, int n)
20   requires x ↦ cell(_) ∧ n>0
21   ensures res ↦ pool(x, n, n);
22 { return forkHelper(x,n,n); }
23
24 void joinHelper(item tp, cell x, int n, int M)
25   requires tp ↦ pool(x, n, M) ∧ M≥n ∧ n>=0
26   ensures x ⟼^{n/M} cell(_) ∧ n>0 ⋁ emp ∧ n = 0;
27 { if (tp==null){ return;} else {
28     joinHelper(tp.next,x,n-1,M);
29     join(tp.t); destroy(tp); }
30
31 void joinThreads(item tp, cell x, int n)
32   requires tp ↦ pool(x, n, n) ∧ n>0;
33   ensures x ↦ cell(_);
34 { return joinHelper(tp,x,n,n); }
35
36 void main() requires emp ensures emp;
37 { cell x = new cell(1); int n = input();
38   item tp = forkThreads(x,n);
39   joinThreads(tp,x,n);
40   destroy(x); }
```

Figure 3-7: Verification of a Program with Threads using Inductive Predicates

At the end, as threads already finished, it is safe to destroy the cell `x`. In this program, each item in the threadpool is a data structure of type `item`. Each `item` will store a thread in its field `t` and a pointer `next` to the next item in the pool. The `forkThreads` returns the first item in the pool, while the `joinThreads` receives the item and joins with all threads in the pool. In the program's specifications, "*res*" is used to denote the returned result of a procedure and "_" represents an unknown value.

The key idea to verify this program is to use an inductively defined predicate, called `pool` to abstract the threadpool. As threads are modeled as resource, they can be naturally captured inside the predicate in the same way as other heap resource, as follows:

$$self \mapsto pool(x, n, M) \stackrel{\texttt{def}}{=} self{=}null \wedge n{=}0 \wedge M{>}0$$
$$\bigvee \exists t, p \cdot self \mapsto item(t, p) * t \mapsto \mathsf{thrd}\langle x \stackrel{1/M}{\longmapsto} cell(\_)\rangle * p \mapsto pool(x, n{-}1, M)$$
$$\textbf{inv}\ n{\geq}0 \wedge M{>}0;$$

The above predicate definition asserts that a `pool` can be empty (the base case $self{=}null$) or consists of a head item (specified by $self \mapsto item(t, p)$), a thread node ($t \mapsto \mathsf{thrd}\langle x \stackrel{1/M}{\longmapsto} cell(\_)\rangle$) and a tail data structure which is also a `pool`. The invariant $n{\geq}0 \wedge M{>}0$ must hold for all instances of the predicate. Using the above definition and case analysis [48], the program can be verified as partially correct and data-race-free. Although we use linked lists here, our approach easily adapts to other data structures, such as arrays.

Thread Liveness and Resource Leakage

Using our approach, threads' liveness can be precisely tracked. For example, we could modify the program in Fig. 3-7 to additionally keep track of already-completed (or dead) threads. In the procedure `joinHelper`, after a thread is joined, instead of destroying the corresponding item (line 29), we could capture all items and their dead threads in a *deadpool* [2], inductively defined as follows:

[2]We refer interested readers to `deadpool` program in our project webpage for more details.

$$self \mapsto deadpool(n) \stackrel{\text{def}}{=} self{=}null \wedge n{=}0$$
$$\bigvee \exists t, p \cdot self \mapsto item(t,p) * p \mapsto deadpool(n{-}1) \wedge dead(t)$$
$$\textbf{inv}\ n{\geq}0;$$

Our approach is also able to keep track of threads' resource in a precise manner. This is important for avoiding leakages of thread resource. As an example, consider the use of a resource split, prior to a join operation.

$$// \{t \mapsto \mathsf{thrd}\langle \Phi_1 * \Phi_2 \rangle\}$$
$$// \{t \mapsto \mathsf{thrd}\langle \Phi_1 \rangle * t \mapsto \mathsf{thrd}\langle \Phi_2 \rangle\}$$
```
join(t);
```
$$// \{\Phi_1 * t \mapsto \mathsf{thrd}\langle \Phi_2 \rangle \wedge dead(t)\}$$

This split causes the join operation to release only resource Φ_1, whilst Φ_2 remains trapped as resource inside a thread node. This results in a resource leakage if the scenario is not properly considered. However, our verification system handles such scenarios by releasing the trapped resource using the **R–THRD2** rule in Fig. 3-5, thus ours avoids the leakages of thread resource.

3.3 Experiments

We demonstrate the feasibility of our "threads as resource" approach by implementing it into a tool for separation logic reasoning. We use our tool to verify partial correctness and data-race freedom of concurrent programs with first-class threads against user-given specifications. The verification is performed modularly for each method, and loops are transformed to recursive methods. Proof obligations generated by our tool will be discharged by external provers such as Redlog [37], Z3 [99], Omega [78], and Mona [79].

The expressiveness of "threads as resource" is beyond that of other verification systems for fork/join programs. However, due to the lack of commonly accepted benchmarks in the literature, we cannot easily compare our tool with other systems.

Table 3.1: Experimental Results. The second column indicates types of a program, i.e. whether it uses fork/join (F), locks (L), non-lexical fork/join (N), multi-join (M), and inductive predicates (P); verification times are average of the 10 runs (in seconds);

Program	**Types**	**Verification Time (s)**
`fibonacci [85]`	F	0.08
`parallel-mergesort [85]`	F	1.24
`oracle [60]`	F/L	1.65
`owicki-gries [65]`	F/L	1.24
`multi-join1`	F/N/M	0.08
`multi-join2`	F/N/M	0.22
`mapreduce`	F/N/M	0.52
`threadpool`	F/N/P	0.20
`deadpool`	F/N/P	0.26
`multicast [51]`	F/L/N/P	1.06

In order to give readers an idea of the applicability of our approach, we did an experiment on a benchmark consisting of small but intricate programs of various types inspired by the literature. [3] Besides the theoretical contributions, the empirical questions we investigate are (1) whether "threads as resource" is capable of verifying more challenging programs, and (2) how well our tool performs. All experiments were conducted on a machine with Ubuntu 14.04, 3.20GHz Intel Core i7-960 processor, and 12GB memory.

The experimental results are presented in Table 3.1. The programs are classified based on whether they use fork/join (F), locks (L), non-lexical fork/join (N), multi-join (M), and inductive predicates (P). Table 3.1 shows that our tool is able to verify programs of various types. For these programs, the verification time is reasonable (less than three seconds). We believe that existing verifiers for verifying concurrent programs can easily integrate our "threads as resource" approach into their systems, and benefit from its greater expressiveness and reasonable verification performance.

[3] Our tool and all experimental programs are available for both online use and download at `http://loris-7.ddns.comp.nus.edu.sg/~project/threadhip/`.

3.4 Flow-Aware Resource Predicates

The notion of "threads as resource" has recently inspired us to propose "*flow-aware resource predicate*", a variant of Concurrent Abstract Predicates (CAP) [33, 36, 121]. Although our proposal for flow-aware resource predicate was originally motivated to handle first-class threads, the proposal is general and can handle other popular concurrency mechanisms such as `countDownLatch` and copyless multicast communication. In the scope of this thesis, we discuss how flow-aware predicates can be used to succinctly model first-class threads. More details on how they are applied to other concurrency mechanisms can be found in our companion technical report [84].

Recall that a newly-spawn thread's pre-condition is consumed at its fork point while its post-condition is carried by thread node(s) and is released at join point(s). In other words, the pre-condition flows into the newly-spawn thread's abstraction at its fork point, and the post-condition flows out of joinee's abstraction at its join point. Our "*flow-aware resource predicate*" is proposed to explicitly track resources that flow into and out of its shared abstraction. Resources of such a predicate can be more flexibly split and transferred across procedure and thread boundaries.

In contrast to `fork` and `join` operations discussed in previous sections which adopt the C-style fork/join concurrency, in this section, we adopt the Java-style concurrency by additionally introducing a `create_thread` operation to create (but not yet execute) a thread. The newly-created thread will then be forked using `fork`. The new approach is more general since the old `fork` operation can be encoded using the `create_thread` operation immediately followed by the new `fork` operation.

In our framework for flow-aware resource predicate, each thread is first created from a method $\texttt{f}(\texttt{v}^*)$ that requires precondition `P` and ensures postcondition `Q`, as specified below:

```
thrd create_thread(f) with P, Q
  requires emp
  ensures Thrd[w*](res, ⊖P, ⊕Q, v*);
```

For simplicity, we require pre/post to be explicitly declared, though this could be inferred from the specification of `f` itself. This method explicitly builds a resource predicate instance $\texttt{Thrd}[\texttt{w}^*](\texttt{t}, \ominus\texttt{P}, \oplus\texttt{Q}, \texttt{v}^*)$ where $\texttt{w}^*$ denotes logically bound variables that link precondition `P` with postcondition `Q`. [4] We use the flow annotation $\ominus$ to denote an inflow (from the current thread) into the resource predicate, while annotation $\oplus$ will denote an outflow from the resource predicate (into the current thread). Resource predicates with both inflow and outflow resources are effectively predicate transformers, used to model methods. Once such a thread has been created, the `fork` operation can be used to start the thread, as denoted by the specification:

```
void fork(thrd t,v*)
  requires Thrd[w*](t, ⊖P, ⊕Q, v*) * P
  ensures Thrd2(t, ⊕Q);
```

which consumes the thread's precondition, and yields another resource predicate $\texttt{Thrd2}(\texttt{t}, \oplus\texttt{Q})$. This new predicate denotes a thread under execution, whose postcondition `Q` is only made available after its thread has finished execution (or joined). The specification of the join operation is itself captured below, which would mark its thread as having completed execution, through a `dead(t)` predicate.

```
void join(thrd t)
      requires Thrd2(t, ⊕Q)
      ensures Q * dead(t);
      requires dead(t)
      ensures dead(t);
```

If the thread is already dead at the point of joining, its state remains unchanged, as captured by the second pair of pre/post specification. Prior approaches, e.g. [36], that use concurrent abstract predicate for reasoning have only considered the property of data-race freedom, but have not considered *resource-preservation* and *deadlock-freedom*. We present our solution to these problems, initially in the context of modelling for first-class threads.

[4] In this section, we slightly abuse the notation $\texttt{t} \mapsto \texttt{pred}\langle\ldots\rangle$ by instead using the predicate instance `pred(t,...)` where `t` is the root pointer of the predicate `pred`.

Ensuring Resource-Preservation

We intend to track all concurrency resources accurately and flexibly. We shall do it in three ways with classical separation logic.

Firstly, the flow annotations are important for ensuring the soundness of our resource predicates, since it can help ensure that all resources are being tracked precisely, by the resource-preserving concurrency primitives. With this property, we can easily show that every concurrency primitive always tracks its resources in a precise manner. In the case of threads, both the `fork` and `join` primitives are *resource-neutral* since the number of incoming and outgoing resources are perfectly balanced. This classification is not possible if flow-annotations are not suitably marked for our resource predicates. However, the `create_thread` primitive has a net resource of $\{\ominus\mathtt{P}, \oplus\mathtt{Q}\}$ since the thread's execution acts as a *predicate transformer*. Note that the concept of resource preservation is not contradicted by the presence of predicate transformer. Resource preservation requires us to account for every resource created, destroyed and transferred through communication channels (or global memory). We can do so in our specification logic by disallowing ambiguous disjunction with different net resources. Two formulas (from a disjunction) are ambiguous if their conjunction is satisfiable. (This concept is related to the notion of precise predicates.)

Secondly, we permit that incoming and outgoing resources to be *flexibly split*, where possible. This will allow us to support more complex concurrency patterns. For example, to support multi-join pattern where a thread may be joined at multiple locations; we shall achieve it using a split lemma that is resource-preserving:

$$\mathtt{Thrd2(t, \oplus(Q_1 * Q_2))} \longrightarrow \mathtt{Thrd2(t, \oplus Q_1)} \ * \ \mathtt{Thrd2(t, \oplus Q_2)}$$

Lemmas for splitting resources have been proposed before for concurrency reasoning. For example, [36] uses $\mathtt{P} \mathrel{-\!\!*} (\mathtt{Q} * \mathtt{R})$ to denote the ability for resource `P` to be split into two resources `Q` and `R`. We clarify the importance of flow-aware resource splitting, and provide support to enable splitting in scenarios that permit aggregated commands. A list of commands $\mathtt{e}_1, \ldots, \mathtt{e}_n$ can be aggregated into a single command `e` if the former $\mathtt{e}_1, \ldots, \mathtt{e}_n$ can be replaced by the latter `e`, without any change in its

observable meaning. For example, two joins on the same thread in a concurrent program can always be replaced by a single join at an earlier forking point. Due to this aggregation property, we allow split to occur for $\texttt{Thrd2}(\texttt{t}, \oplus(\texttt{Q}_1 \texttt{ * } \texttt{Q}_2))$ whose resources are needed for pre-conditions of multi-join operations.

Lastly, with the help of classical separation logic, we shall further classify our resources into the following categories:

Definition 1 (Resource-Loaded)**.** *A* resource-loaded *predicate is a predicate which definitely captures some resource. Each of its occurrences must be tracked precisely to avoid resource leakage. Two examples of* resource-loaded *predicates are* $\texttt{Thrd2}(\texttt{t}, \oplus\texttt{Q})$ *and* $\texttt{Thrd}[\texttt{w}^*](\texttt{t}, \ominus\texttt{P}, \oplus\texttt{Q}, \texttt{v}^*)$.

Definition 2 (Resource-Less)**.** *A* resource-less *predicate is a predicate which no longer captures any resources. An example of this predicate is* `dead(t)` *which denotes a thread that has expired (finished execution). Such a predicate is also a pure predicate.*

This classification can help ensure that each *resource-loaded* predicate is never lost by our classical reasoning system. In contrast, *resource-less* predicates, like `dead(t)`, can be lost as they are classified as pure predicates, with the following idempotent lemma:

$$\texttt{dead(t)} \longleftrightarrow \texttt{dead(t)} \texttt{ * dead(t)}$$

For each dead thread, we allow its *resource-loaded* predicate with only outgoing resource to be automatically released, thus:

$$\texttt{Thrd2}(\texttt{t}, \oplus\texttt{Q}) \texttt{ * dead(t)} \longrightarrow \texttt{dead(t)} \texttt{ * Q}$$

This is to help with resource-preservation when a resource is known to have terminated. We ensure that each lemma used is resource-preserving, and the specification of each method used be resource-precise with no ambiguous disjunction.

Ensuring Deadlock Freedom

We now consider how (intra-resource) deadlock errors (from a single resource) are detected in our approach through the use of synchronization lemmas. In particular,

each *resource-loaded* predicate with an incoming resource cannot co-exist with its *resource-less* predicate state, as highlighted below, for each thread.

$$\texttt{Thrd}[\texttt{w}^*](\texttt{t}, \ominus\texttt{P}, \oplus\texttt{Q}, \texttt{v}^*) \ * \ \texttt{dead(t)} \longrightarrow \texttt{ERROR}$$

Note that `ERROR` is distinct from `false`, since each of its occurrences would be flagged by our verification, while `false` simply denotes unreachability. Furthermore, we refer to these as synchronization lemmas, since these errors are meant to capture single-resource deadlock scenarios. In this example, it is a deadlock caused by a `join` operation being invoked before the `fork` operation. It is wrong to terminate a thread before it has even started. Nevertheless, this synchronization scenario never arises in our current modelling for threads, since the resource predicates are being created in the following orders (based on usage protocol specified via its pre/post): $\texttt{Thrd}[\texttt{w}^*](\texttt{t}, \ominus\texttt{P}, \oplus\texttt{Q}, \texttt{v}^*) \rightarrow \texttt{Thrd2}(\texttt{t}, \oplus\texttt{Q}) \rightarrow \texttt{dead(t)}$, where the third `dead(t)` could not have co-existed with the first uninitiated $\texttt{Thrd}[\texttt{w}^*](\texttt{t}, \ominus\texttt{P}, \oplus\texttt{Q}, \texttt{v}^*)$ for any thread `t`. Any such deadlock errors would have been detected as a pre-condition failure for `join` which requires a `fork` to occur before hand.

Inter-resource deadlocks may also have occurred, as illustrated by our next example on two threads:

$$\texttt{t1 = create\_thread(f1)} \ \textit{with} \ \texttt{P}_1, \texttt{Q}_1;$$
$$\texttt{t2 = create\_thread(f2)} \ \textit{with} \ \texttt{P}_2, \texttt{Q}_2;$$
$$\texttt{// \{ Thrd(t1, } \ominus\texttt{P1, } \oplus\texttt{Q1) * Thrd(t2, } \ominus\texttt{P2, } \oplus\texttt{Q2) \}}$$

$$\left(\begin{array}{l} \texttt{// \{ Thrd(t2, } \ominus\texttt{P2, } \oplus\texttt{Q2) \}} \\ \texttt{join(t2); /*pre-cond fails*/} \\ \texttt{fork(t1);} \end{array} \ \middle\| \ \begin{array}{l} \texttt{// \{ Thrd(t1, } \ominus\texttt{P1, } \oplus\texttt{Q1) \}} \\ \texttt{join(t1); /*pre-cond fails*/} \\ \texttt{fork(t2);} \end{array} \right);$$

Here, waiting could only have occurred for `join` commands. Inter-thread deadlock would have occurred in this scenario, since each of the two `join` commands is waiting for the other thread to initiate the `fork` operation. However, due to strict order of the three resource predicates, such a scenario would have been detected as a pre-condition failure for the `join` commands. Hence, synchronization lemma is not

required for preventing deadlocks amongst multiple threads due to its usage protocol. Once verified, we guarantee our thread resources to be deadlock-free.

In this section, we have shown how flow-aware resource predicates can be used to reason about first-class threads. Our flow-aware resource predicate is general and can handle other popular concurrency mechanisms such as `countDownLatch` and copyless multicast communication. More details can be found in our technical report [84].

3.5 Discussion

This section discusses related works on reasoning about shared-memory concurrent programs. Our approach currently supports only partial correctness. Proving (non-)termination is an orthogonal issue and could be separately supported.

Traditional works on concurrency verification such as Owicki-Gries [111] and Rely/Guarantee reasoning [69] often focused on simple parallel composition, rather than fork/join. Fork/join concurrency is more general than the parallel composition for two main reasons. First, fork/join supports dynamic thread creation and termination. Second, while threads in a parallel composition are lexically scoped, threads in fork/join programs can be non-lexically scoped. Therefore, fork/join programs are more challenging for verification. Even recent approaches such as CSL [105], RGSep [126], LRG [38], and Views [31], omit fork/join concurrency from their languages. Our "threads as resource" is complementary to the above approaches and could be integrated into them.

There also exist approaches that can handle fork/join operations. Both Hobor [60] and Feng and Shao [40] support fork and omit join with the claim that thread join can be implemented using synchronization. However, without join, the former allows threads to leak resource upon termination while the latter requires global specifications of inter-thread interference. Approaches that can handle both fork and join often use tokens [51, 65, 91] to represent the post-states of forked threads. However, they offer limited support for first-class threads: the tokens are not allowed to be

split and shared among concurrent threads. As such, they are not expressive enough to verify programs with more intricate fork/join behaviors such as the multi-join pattern where threads are shared and joined in multiple threads. Existing works could encode the multi-join pattern by using synchronization primitives such as channels or locks. However, the encoding requires additional support for the primitives and could complicate reasoning (i.e. we have to reason about channels or locks instead of just focusing on threads). Our approach is more elegant and natural. Inspired by the key notation of resource in separation logic [16, 105], we propose to model threads as resource, thus allow ownerships of threads to be flexibly split and distributed among multiple joiners. This enables verification of the multi-join pattern. In addition, unlike ours, none of related works that we are aware of support explicit reasoning about thread liveness. To the best of our knowledge, only Haack et al. [53, 54] can reason about some multi-join scenarios. In their approach, a thread token can be associated with a fraction and this allows multiple joiners to join with the same joinee in order to read-share the joinee's resource. However, this simple multiplicative treatment of thread tokens is not expressive enough as it is unable to verify programs that require the joiners to write-share the resource of the joinee (e.g. the program in Fig 3-1). In order to cater to a more flexible treatment of joinees and their resource, modeling threads as resource is essential.

Our flow-aware resource predicates are variants of Concurrent Abstract Predicates (CAP) [33, 36, 121]. The basic idea behind CAP [33] was to provide an abstraction of possible interferences from concurrently running threads, by partitioning the state into regions with protocols governing how the state in each region is allowed to evolve. Dodds et al. [36] introduced a higher-order variant of CAP to give a generic specification for a library for deterministic parallelism, making explicit use of nested region assertions and higher-order protocols. Despite being powerful, their specifications may render the reasoning to be unsound in certain corner cases. More recently, Svendsen et al. [121] presented a new logic, Higher Order Concurrent Abstract Predicates (HOCAP), allowing clients to refine the generic specifications of concurrent

data structures. HOCAP was developed based on Jacobs and Piessens' idea of parameterizing specifications of concurrent methods with ghost code, to be executed in synchrnonization points [65]. Our flow-aware resource predicates explicitly track resources that flow into and out of their abstractions and allow resources to be flexibly split and transferred across procedure and thread boundaries. Our proposed specification and verification mechanism is rather general as it not only supports first-class threads, but is also capable of handling other popular concurrency patterns such as `countDownLatch` and copyless message passing [91, 127].

3.6 Summary

We proposed to model first-class threads as resource to enable expressive treatment of threads' ownerships. Our approach allows resources of threads to be flexibly split, combined, and transferred across procedure boundaries. This enables verification of multi-join pattern, where multiple joiners can share and join the same joinee in order to manipulate (read or write) the resource of the joinee after join. In addition, we demonstrated how threads as resource is combined with inductive predicates to capture the commonly-used threadpool idiom. Using a special *dead* predicate, we showed that thread liveness can be precisely tracked. We have implemented our approach in a tool, and experimental results showed reasonable verification time. We also discussed our newly-proposed "*flow-aware resource predicate*", which was inspired by "threads as resource", for verifying various concurrency mechanisms, including and beyond first-class threads.

This chapter provides an infrastructure for addressing other open issues in reasoning about programs with first-class threads via fork/join. Specifically, in the next chapter, we present an expressive verification framework for ensuring deadlock freedom of shared-memory programs with fork/join concurrency and (mutex) locks.

Chapter 4

Verification of Deadlock Freedom

Overview. Understanding and reasoning about the correctness of concurrent programs is rather complicated due to non-deterministic interleavings of concurrent threads [88]. These interleavings may result in *deadlocks* [27], i.e. states in which each thread in a set blocks waiting for another thread in the set to release a lock or complete its execution. Deadlocks are common defects in software systems. Specifically, in Sun's bug report database at `http://bugs.sun.com/`, there are approximately 6,500 bug reports out of 198,000 ($\sim 3\%$) containing the keyword "deadlock" [102]. In this chapter, we propose an expressive framework for reasoning about the correctness of concurrent programs with a focus on eliminating deadlocks.

Existing verification systems [51, 61, 90, 91] often use *abstract predicates* to represent states of locks. For example, Gotsman et al. [51] use abstract predicate *Locked(x)* to specify that the lock x is owned by the current thread. Hobor et al. [61] use the predicate *hold x R* and Chalice [90, 91] uses *holds(x)* for the same purpose. Intuitively, a lock is owned by a thread if it is in the set of locks already acquired by the thread, i.e. the thread's lockset. Interestingly, although using predicates, previous works [51, 61, 90, 91] formulate their soundness proof using the notion of lockset. Additionally, Haack et al. [52] show that lockset (or rather lockbag) is necessary to reason about Java recursive locks. In retrospect, one can say that lockset has proven to be an important abstraction for verifying concurrent programs that manipulate

locks.[1] In this chapter, we advocate the use of *precise locksets* for explicitly reasoning about the presence or absence of locks, empowering a more expressive framework for verifying deadlock freedom even in the presence of interactions between thread operations (e.g. fork/join) and lock operations (e.g. acquire/release). Due to the dynamic nature of threads, sound reasoning of the interactions between thread and lock operations is non-trivial.

```
1  int running;
2  pthread_t thread;
3  mutex_t mutex;
4
5  void* timer(){
6    int state;
7    do{
8      mutex_lock(&mutex);
9      state=running;
10     mutex_unlock(&mutex);
11     .../*timing*/
12   }while(state);
13 }
```

```
14 void main(){
15   running = 0;/*init timer*/
16   mutex_lock(&mutex);
17   running = 1;/*start timer*/
18   pthread_create(&thread,&timer);
19   mutex_unlock(&mutex);
20   /*begin timed computation*/
21   ...
22   /*end computation*/
23   mutex_lock(&mutex);
24   running = 0;/*stop timer*/
25   mutex_unlock(&mutex);
26   pthread_join(thread);
27 }
```

Figure 4-1: A Program with Interactions between Thread and Lock Operations

Fig. 4-1 outlines a simplified[2] C implementation of a timer used in NetBSD operating system's report database [1]. Though rather intricate due to the interactions between lock and thread operations, the program is deadlock-free because the two threads never wait for each other. However, if the programmer does not release the lock before joining (e.g. line 25 is missing or line 25 and 26 are swapped), the interactions will cause a deadlock when the main thread blocks waiting to join the child thread and the child thread also blocks waiting to acquire the mutex being held by the

[1] See Section 4.1.1 for detailed comparison between abstract predicates and locksets.

[2] In the original implementation, there is a conditional variable associated with the mutex to more efficiently signal the `timer` thread to start and stop timing. As verifying conditional variables is an orthogonal issue, we have omitted them for simplicity.

main thread. For larger programs with many (possibly non-deterministic) execution branches, these interactions are not easy to follow [88]. With concurrent programs becoming mainstream in this multi-core era, we will increasingly require a more comprehensive solution for constructing and verifying these intricate interactions.

In this chapter, we propose an expressive verification framework to guarantee deadlock freedom in the presence of such interactions. Our framework has the following innovations:

- *Delayed lockset checking* to help reason about the interactions between thread and lock operations. Unlike the traditional verification approaches [51, 52, 61, 65, 90] that check pre-conditions of procedures entirely at fork points, this technique allows lockset constraints in the pre-conditions to be delayed and checked at join points instead. This prevents deadlocks due to the interactions and also permits more programs to be declared as deadlock-free.

- *Precise lockset reasoning*, as opposed to ones based on *abstract predicates* or *approximated locksets*, to ensure that deadlock-free pre-conditions on lock acquisition and release can be guaranteed. Any uncertainty from static program analysis is simply captured through the use of explicit disjunction.

- *Combining lockset with the concept of locklevels*, which has been used popularly in the literature [17, 90, 120], to form an expressive framework for ensuring deadlock freedom, covering various scenarios such as double lock acquisition, interactions between thread and lock operations, and unordered locking.

- A prototype specification and verification system, called ParaHIP, to show that the proposed framework has been successfully integrated with separation logic [115] for reasoning about concurrent programs.

The rest of this chapter is organized as follows. Section 4.1 gives concrete examples that motivate our delayed lockset checking technique and show how precise lockset reasoning can be systematically supported. Section 4.2 presents our framework in

details. Section 4.3 discusses the implementation and experimental results of our prototype tool. Section 4.4 summarizes related work. Section 4.5 concludes our work in this chapter.

4.1 Motivation and Proposed Approach

4.1.1 Lockset as an Abstraction

While most previous works [51, 61, 90] use *abstract predicates* for reasoning about concurrent programs manipulating locks, we advocate the use of explicit *locksets*. Though monadic predicates are logically equivalent to sets, they are not always realised as such for several reasons. Firstly, while *Locked(x)* only captures the concrete presence of the lock x in the current thread's lockset, the notion of lockset can capture a symbolic set of locks. Secondly, the application of frame rule [115] makes it more difficult to reason about the absence of a given predicate. Thus it is harder to reason about absence of locks using predicates, e.g. to avoid deadlocks due to double acquisition. Lastly, we often avoid the use of negation for predicates, such as $\neg$*Locked(x)*, since such operator may be difficult to implement. In contrast, with lockset, if a callee is going to acquire a non-recursive lock x, it is simpler to check that a given lock is not in the current thread's lockset (denoted by **LS**), by using $x \notin \mathbf{LS}$ in the pre-condition of the callee. Nevertheless, such check can be considered as sound only if the given lockset is *precise* and not an *approximation*, as explained next.

4.1.2 Precise Lockset Reasoning

In our verification framework, **LS** is a thread-local ghost variable[3] capturing the set of locks held by a thread. Lockset is a verification concept rather than a programming language concept. Using lockset, verification rules for *acquire* and *release* operations on non-recursive (mutex) locks[4] can be defined as follows:

[3]Ghost variables are variables used for verification purpose. They do not affect program correctness.

[4]Cannot be acquired more than once; also called non-reentrant locks

```
acquire(lock x)
  requires x∉LS
  ensures LS′=LS∪{x};

release(lock x)
  requires x∈LS
  ensures LS′=LS−{x};
```

Note that we use *primed notation* to denote updates to variables. The primed version $\mathbf{LS}'$ of the variable **LS** denotes its latest value; the unprimed version **LS** denotes its old value at the start of the respective procedure call. Using lockset, it is straightforward to prevent the deadlock due to acquiring a non-recursive lock twice in the `thread` code of Fig. 4-2. In this sequential setting, our verification reports an error because the pre-condition of the callee `func` ($l1 \notin \mathbf{LS}$) cannot be satisfied by the current lockset of the caller ($\mathbf{LS}'=\{l1\}$). Additionally, the `release` rule excludes the possibility of releasing a lock more than once.

```
void thread()
  requires LS={}  ensures LS′={};
{
  lock l1 = new lock();
  //{ LS′={} }
  acquire(l1);
  //{ LS′={l1} }
  func(l1); /*Error*/
  release(l1);
}
```

```
void func(lock l1)
  requires l1∉LS  ensures LS′=LS;
{
  //{ l1∉LS ∧ LS′=LS }
  acquire(l1);
  //{ l1∉LS ∧ LS′=LS∪{l1} }
  release(l1);
  //{ l1∉LS ∧ LS′=LS∪{l1}−{l1} }
  //{ l1∉LS ∧ LS′=LS }
}
```

Figure 4-2: Deadlock due to Double Acquisition of a Non-recursive Lock

In each given program, there can be many locking scenarios across different execution branches. Each branch could potentially have a different lockset. The following code fragment shows a simple example where locksets at two branches are $\mathbf{LS}'=\{x\}$ and $\mathbf{LS}'=\{\}$, which are clearly different:

```
//{ LS′={} }
if (b) { acquire(x);//{ LS′={x} } } else { //{ LS′={} } }
```

For static analysis, we often perform some approximation. For example, one may *over-approximate* on the lockset, by using $\mathbf{LS}'=\{x\}$ as the post-state of the above code fragment. However, this approach would fail to detect the definite presence of the lock `x` for safe release. Another approach is to *under-approximate* on the lockset by using $\mathbf{LS}'=\{\}$, but this approach fails to detect the definite absence of the lock for safe acquisition. Thus, one plausible solution is to combine the two approximations by capturing both may-hold and must-hold locksets, simultaneously. However, this approach would be more complex due to the use of two locksets. In this chapter, we propose a simpler solution that would mandate the use of *precise locksets* in our verification/analysis. For approximation, we propose to use *disjunctive formulae* to capture uncertainty and also allow program states, other than lockset, to be over-approximated. In the above example, we can ensure precise lockset by using either $b \wedge \mathbf{LS}'=\{x\} \vee \neg b \wedge \mathbf{LS}'=\{\}$ or even $\mathbf{LS}'=\{x\} \vee \mathbf{LS}'=\{\}$ as its post-state, but never $\mathbf{LS}'=\{x\}$, since we always ensure that each lockset is precisely captured and never approximated. This principle allows us to support precise reasoning on locksets for verifying deadlock freedom.

4.1.3 Delayed Lockset Checking

Fig. 4-3 shows two programs that are challenging for existing verification systems, because they express rich interactions between fork/join concurrency and lock operations. The traditional way of verification [51, 52, 61, 65] cannot sufficiently handle these scenarios because it performs the check for the pre-condition of the forkee *only* at the *fork point*. This could incorrectly verify the program in Fig. 4-3(a) as deadlock-free and reject the deadlock-free program in Fig. 4-3(b). The well-known technique [17, 90, 120] which requires threads to acquire multiple locks in a specific order to avoid deadlocks could not directly handle complications due to fork/join concurrency. In this work, we propose *delayed lockset checking* technique that is capable of preventing deadlock scenarios (such as that presented in Fig. 4-3(a)) and proving more programs (such as that described in Fig. 4-3(b)) to be deadlock-free.

```
void func(lock l1)
  requires l1∉LS  ensures LS′=LS;
{ acquire(l1); release(l1); }

void main()
  requires LS={}  ensures LS′={};
{
 lock l1 = new lock();
 //{ LS′={} }
 int id = fork(func,l1);/*DELAY*/
 //{ LS′={} }
 acquire(l1);
 //{ LS′={l1} }
 /*Potentially deadlocked when join*/
 join(id); /*CHECK, error*/
 release(l1);
}
```

(a) Potentially deadlocked

```
void func(lock l1)
  requires l1∉LS  ensures LS′=LS;
{ acquire(l1); release(l1); }

void main()
  requires LS={}  ensures LS′={};
{lock l1 = new lock();
 //{ LS′={} }
 acquire(l1);
 //{ LS′={l1} }
 int id = fork(func,l1);/*DELAY*/
 //{ LS′={l1} }
 release(l1);
 //{ LS′={} }
 join(id);/*CHECK, ok*/
 //{ LS′={} }
}
```

(b) Deadlock-free

Figure 4-3: Examples of Programs Exposing Interactions between Thread and Lock Operations

This technique is based on the following observation. At a *fork point*, a verifier is unaware of future operations performed by a main (or parent) thread; the only information it knows of is future locking operations executed by a child thread thanks to the use of *lockset*. For example, a constraint $l1 \notin \mathbf{LS}$ in the pre-condition of a child thread implies that the child thread is going to acquire the lock `l1`. Therefore, in order to ensure that the child thread will finally be able to acquire the lock (and thus avoid deadlocks), the main thread should not be holding the lock while waiting for the child thread at its *join point*. In other words, *when forking a child thread, lockset constraints in its pre-condition are not checked at the fork point but are delayed to be checked at its join point instead.*

The deadlock in Fig. 4-3(a) can be prevented by deferring the lockset constraint $l1 \notin \mathbf{LS}$ of the child thread to its join point. At the join point, the constraint is checked and the verification reports an error because the constraint is unsatisfiable

($\mathbf{LS}'=\{l1\}$ at the join point). Similarly, the program in Fig. 4-3(b) is ensured as being deadlock free because the lockset constraint $l1 \notin \mathbf{LS}$ is delayed from the fork point and is satisfiable at the join point ($\mathbf{LS}'=\{\}$). Note that, although main and child threads have different locksets, a constraint $l1 \notin \mathbf{LS}$ in pre-conditions of a child thread indicates its intention to acquire the lock $l1$, hence this constraint can be soundly checked against the lockset of the main thread to prevent deadlocks. Besides, it is unsound to check lockset constraints at any satisfiable points in the middle of the fork point and the join point. For example, in a scenario similar to Fig. 4-3(b), after forking a child thread, the main thread releases the lock. At this point, the lockset constraint is satisfiable. However, the main thread could later acquire the lock again and wait for the child thread to join. This scenario still suffers a potential deadlock. As a result, it is only sound to check delayed lockset constraints at just the join points.

In summary, the main benefit of our *delayed lockset checking* technique is to facilitate more expressive deadlock verification in the presence of interactions between parent/child threads and lock operations.

4.1.4 Combining Lockset and Locklevel

Another type of deadlocks occurs when threads attempt to acquire the same set of locks in different orders (unordered locking). An example of such a scenario is shown in Fig. 4-4. Locklevel is well-known for preventing deadlocks due to unordered locking [17, 90, 120]. For example, in Chalice, each lock in a program is associated with a ghost field *mu* representing the lock's level, e.g. $l1.mu$ denotes the locklevel of lock `l1`. With it, deadlocks can be prevented indirectly by ensuring that locks are acquired in a strictly increasing order of locklevels. To check that locks are acquired in the specified order, a ghost variable **waitlevel** is used to capture the maximum level currently acquired by a thread, i.e. **waitlevel** is the maximum level among locklevels of all locks in current thread's lockset **LS**. A thread can acquire a lock only if its current waitlevel $\mathbf{waitlevel}'$ is lower than the lock's level. Using locklevels, the deadlock in Fig. 4-4 can be prevented. The verification system reports an error when

the child thread attempts to acquire lock `l1` whose locklevel is lower than the current waitlevel of the child thread.

In the pre-condition of the `func` procedure (Fig. 4-4), we use the specification $[\omega\#\psi]$ to capture the fact that the waitlevel constraint ω and the lockset constraint ψ are *mutually exclusive*, i.e. the former is checked in accordance with sequential settings (at the points of normal procedure calls or fork operations), while the latter is a check needed to be delayed in concurrent settings (at the points of fork operations) and will be check at join points instead. This provides a *single* mechanism for procedure declarations so that each procedure could be either forked as a child thread or invoked as a normal procedure call.

In summary, precise lockset, delayed lockset checking, and locklevel are complementary and combining them is essential to form an expressive framework for verifying

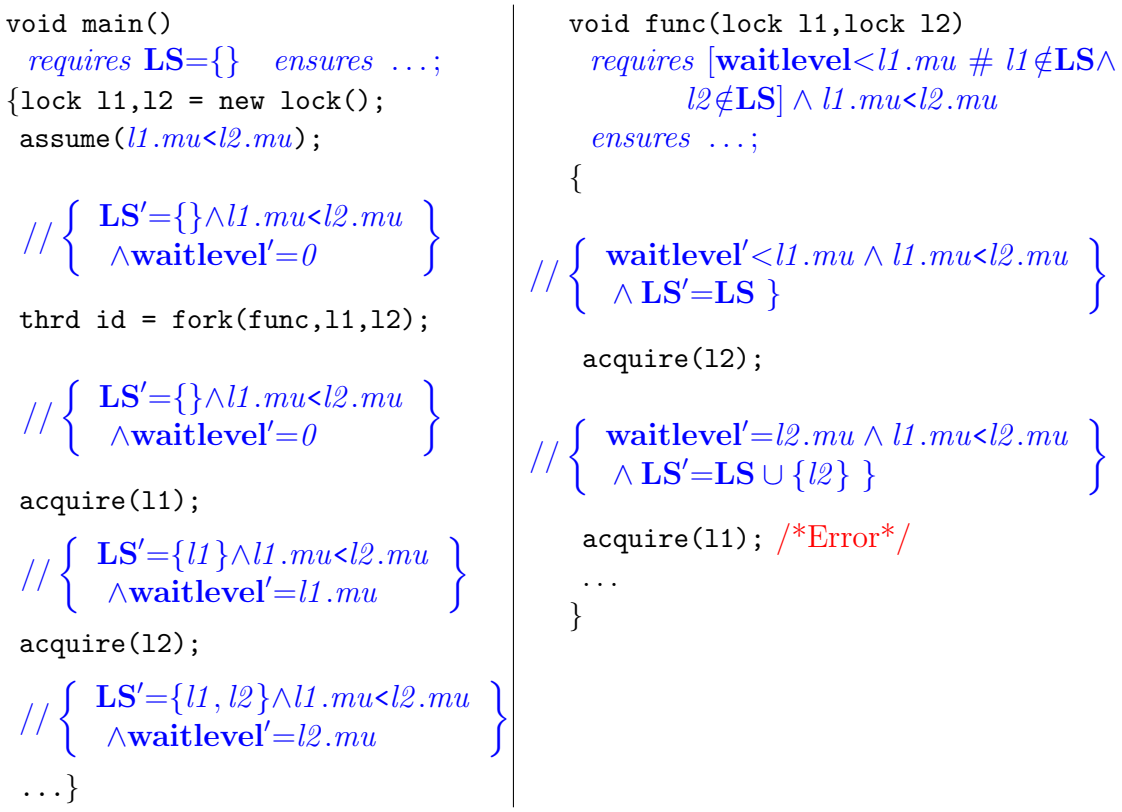

Figure 4-4: A Potential Deadlock due to Unordered Locking

various deadlock scenarios such as double acquisition, interactions between fork/join and acquire/release, and unordered locking.

4.2 Formalism

In this section, we present a specification logic that can be used to verify deadlock freedom. We show how our approach, based on precise lockset abstraction, can be integrated with the locklevel idea from CHALICE [90]. We also present a specification formalism to unify constraints on lockset, locklevel and waitlevel into a single specification and to allow each procedure to be used internally or as the entry point of a newly-forked thread.

4.2.1 Programming Language

t	::=	$\ldots \mid$ `lock`	Type
s	::=	`lock` $v =$ **new** `lock`(v) $\mid$ **acquire**$(v) \mid$ **release**(v) $\mid \ldots$	Statement

Figure 4-5: Programming Constructs for (Mutex) Locks

We enhance the programming language described in previous chapter (Section 3.2.1) with constructs for non-recursive (mutex) locks. `lock` $v =$ **new** `lock`(v) creates a new lock with a ghost argument representing its locklevel. **acquire**(v) and **release**(v) attempt to acquire and respectively release the lock v.

Ensuring Ownership Semantics. Locks in programming languages such as Pthreads provide the notion of ownership (see §10.1.2 of [21]) whereby each lock has to be released only by the thread which acquired (or owned) it. Conforming to this semantics is important to avoid undefined behaviors which could potentially cause unexpected errors [21]. To ensure this semantics, when verifying a forked procedure, our verifier additionally checks if locksets in pre/post-conditions of the forked procedure

are empty. An empty lockset in pre-condition of a forked procedure ensures that a child thread does not inherit any locks from its main thread when being forked and hence prevents the child thread from releasing a lock acquired by the main thread. An empty lockset in post-condition of a forked procedure prevents deadlocks in case other threads try to acquire or release a lock held by a terminated child thread. Note that this requirement on empty locksets is not applicable to normal procedure calls.

4.2.2 Integrating Specification with Locklevels

In our specification logic, a lockset variable **LS** captures a set of locks held by the current thread. Like CHALICE [90], each lock in a program has an immutable ghost field *mu* representing the lock's level. Locklevels are implemented as natural numbers and operator $op \in \{=, <, >\}$ is used over locklevels. The lowest (bottom) locklevel is denoted as 0. A **waitlevel** variable can be derived from the lockset and locklevels. As a reminder, waitlevel is the maximum level among locklevels of all locks in current thread's lockset **LS**. Levels of locks in a program are strictly positive while a bottom locklevel denotes the waitlevel in case of empty lockset. Using lockset as an abstraction, constraints on **waitlevel** can be expressed in terms of constraints on lockset and locklevels as follows:

$\texttt{maxLL}(S) \stackrel{\texttt{def}}{=} \texttt{if } S = \{\} \texttt{ then } 0 \texttt{ else max}\{v.mu \mid v \in S\}$

$\textbf{waitlevel } op\ x \stackrel{\texttt{def}}{=} \texttt{maxLL}(\textbf{LS})\ op\ x$

$\textbf{waitlevel}'\ op\ x \stackrel{\texttt{def}}{=} \texttt{maxLL}(\textbf{LS}')\ op\ x$

where $op \in \{=, <, >\}$, and $\texttt{maxLL}(S)$ returns the maximum locklevel of the locks in the set S and returns the bottom locklevel (i.e. 0) if S is empty. In our implementation, the constraints on **waitlevel** are translated into the following boolean expressions, which are discharged by Mona prover [79].

$\textbf{waitlevel}{<}x \stackrel{\texttt{def}}{=} (\textbf{LS}{=}\{\} \Rightarrow 0{<}x) \wedge (\textbf{LS}{\neq}\{\} \Rightarrow \forall v{\in}\textbf{LS} \cdot v.mu{<}x)$

$\textbf{waitlevel}{>}x \stackrel{\texttt{def}}{=} (\textbf{LS}{=}\{\} \Rightarrow 0{>}x) \wedge (\textbf{LS}{\neq}\{\} \Rightarrow \exists v{\in}\textbf{LS} \cdot v.mu{>}x)$

$\textbf{waitlevel}{=}x \stackrel{\texttt{def}}{=} (\textbf{LS}{=}\{\} \Rightarrow 0{=}x) \wedge$
$(\textbf{LS}{\neq}\{\} \Rightarrow \forall v{\in}\textbf{LS} \cdot v.mu{\leq}x \wedge \exists u{\in}\textbf{LS} \cdot u.mu{=}x)$

The following procedure illustrates the use of the above constraints on waitlevel:

```
void acquire_both(lock l1,lock l2)
  requires [waitlevel<l1.mu # l1∉LS∧l2∉LS] ∧ l1.mu<l2.mu
  ensures waitlevel'=l2.mu ∧ LS'=LS ∪ {l1, l2};
{ acquire(l1);
  assert( waitlevel'<l2.mu ); /*valid*/
  acquire(l2);
  assert( waitlevel'>l1.mu ); /*valid*/
}
```

4.2.3 Specification Language

Logic formula	Φ	::=	$\bigvee(\exists v^* \cdot \kappa \wedge \ell \wedge \pi)$
Heap formula	κ	::=	$\mathsf{emp} \mid \iota \mid \kappa_1 * \kappa_2$
Atomic heap formula	ι	::=	$v \mapsto \mathsf{thrd}\langle\gamma \rightarrow \Phi\rangle \mid \ldots$
Lock formula	ℓ	::=	$[\bigwedge\omega \# \bigwedge\psi]$
Delayed formula	γ	::=	$\bigvee(\bigwedge\psi \wedge \pi)$
Waitlevel formula	ω	::=	$\mathbf{waitlevel}{=}\alpha^t \mid \mathbf{waitlevel}{<}\alpha^t \mid \mathbf{waitlevel}{>}\alpha^t$
Lockset formula	ψ	::=	$v \in \mathbf{LS} \mid v \notin \mathbf{LS}$
Pure formula	π	::=	$\alpha \mid \beta \mid \pi_1 \wedge \pi_2 \mid \pi_1 \vee \pi_2 \mid \neg\pi \mid \exists v \cdot \pi \mid \forall v \cdot \pi \mid \mathbf{true}$
Set term	β^t	::=	$\mathbf{LS} \mid \{\} \mid \{v\} \mid \beta_1^t \cup \beta_2^t \mid \beta_1^t \cap \beta_2^t \mid \beta_1^t - \beta_2^t$
Set formula	β	::=	$\beta_1^t \sqsubset \beta_2^t \mid \beta_1^t = \beta_2^t$
Arithmetic term	α^t	::=	$k \mid v \mid v.mu \mid k \times \alpha^t \mid \alpha_1^t + \alpha_2^t \mid -\alpha^t$
Arithmetic formula	α	::=	$\alpha_1^t = \alpha_2^t \mid \alpha_1^t \neq \alpha_2^t \mid \alpha_1^t < \alpha_2^t \mid \alpha_1^t \leq \alpha_2^t$

$v, w \in$ Variables $\qquad k \in$ Integer constants

Figure 4-6: Grammar for Specification Language with **LS** and **waitlevel**

Fig. 4-6 shows our specification logic. In the specification, Φ is a logic formula in disjunctive normal form. Each disjunct in Φ consists of a heap formula κ for "threads as resource", a lock formula ℓ, and a pure formula π. Each thread node in κ captures a child thread. A lock formula ℓ consists of waitlevel formulae ω, and lockset formulae ψ. ω and ψ are self-explanatory.

A lock formula $[\bigwedge\omega \ \# \ \bigwedge\psi]$ presents our mechanism for each procedure's dual use, namely for both sequential and concurrent execution. The formula captures both waitlevel formula $\bigwedge\omega$ and lockset formula $\bigwedge\psi$ that are *mutually exclusive*. The former is checked for sequential procedural calls, while the latter must be delayed and checked at join points of forked threads. We provide both specifications in a unified format to cater to the differences in semantics for both sequential and concurrent computations. In sequential settings, e.g. when invoking a normal procedure call, the pre-condition of a procedure is an assertion that has to be fulfilled by the caller. If one or more constraints about lockset and waitlevel in the pre-condition are not met, verification fails. In concurrent settings and due to the ownership semantics of locks, each new child thread does not inherit any locks from its parent thread. Hence, it has empty lockset and bottom waitlevel. Thus, constraints on waitlevel need not be checked here. Nevertheless, the constraints on lockset indicate the intention of the child thread and must be "delayed for checking" at its join point instead.

A thread node represents the final state of a child thread. It consists of a delayed formula γ (for delayed lockset checking), and a logic formula Φ capturing the thread's post-state (i.e. its effects after finishing its execution). The thread node $v \mapsto \mathsf{thrd}\langle\gamma \rightarrow \Phi\rangle$ denotes the fact that when the thread is joined and its delayed formula γ is satisfied, then its effects Φ will be visible to the calling thread. The sub-structural rules for handling "threads as resource" in the presence of delayed lockset constraints are presented in Fig. 4-7. The rules manipulate post-states Φ of threads and leave delayed lockset constraints γ intact.

$$v \mapsto \mathsf{thrd}\langle\gamma \rightarrow \Phi_1 * \Phi_2\rangle \iff v \mapsto \mathsf{thrd}\langle\gamma \rightarrow \Phi_1\rangle * v \mapsto \mathsf{thrd}\langle\gamma \rightarrow \Phi_2\rangle \qquad \textbf{R–THRD3}$$

$$v \mapsto \mathsf{thrd}\langle\gamma \rightarrow \Phi\rangle \wedge \mathit{dead}(v) \implies \Phi \qquad \textbf{R–THRD4}$$

Figure 4-7: Added Sub-structural Rules for Delayed Lockset Checking

The formula γ illustrates our support for *delayed lockset checking*. Each disjunct in γ consists of delayed lockset constraints $\bigwedge\psi$ and a pure formula π to more precisely capture additional constraints for the corresponding delayed lockset constraints to hold. At each join point, only disjuncts whose pure formula is satisfied are candidates for delayed lockset checking.

Lastly, a pure formula π consists of standard equality/inequality, Presburger arithmetic, and set constraints. Additionally, it is straightforward to enhance our specification logic to ensure data-race freedom. However, for simplicity of presentation, this chapter focuses on just the framework for deadlock freedom and ignores all issues pertaining to data-races.

For illustration, consider the following logic formula:

$$v \mapsto \mathsf{thrd}\langle(l1 \notin \mathbf{LS}' \wedge b \wedge l1 \neq \mathtt{null}) \vee (l2 \notin \mathbf{LS}' \wedge \neg b \wedge l2 \neq \mathtt{null}) \rightarrow \mathtt{emp}\rangle \wedge$$
$$l1 \neq \mathtt{null} \wedge l1.mu > 0 \wedge l2 \neq \mathtt{null} \wedge l2.mu > 0 \wedge \mathbf{LS}' = \{l2\} \wedge b$$

The formula represents a program state where there are two concurrent threads: a main thread currently holding the lock $l2$ (i.e. $\mathbf{LS}'=\{l2\}$) and a child thread captured by a thread node. The child thread has a disjunctive delayed formula which precisely captures two locking scenarios: the child thread either acquires the lock $l1$ if the boolean condition on variable b holds or acquires the lock $l2$ if the condition does not hold. Suppose that the main thread is going to join the child thread. The main thread, knowing that b holds, can exclude the deadlock scenario that the child thread potentially attempts to acquire the lock $l2$. Hence it is deadlock-free to join the child thread. Note that due to our assumption on data-race freedom, the boolean condition on variable b is consistent in both threads.

4.2.4 Verification Rules

Proof rules for forward verification are presented in Fig. 4-8. They are formalized using Hoare's triples of the form $\{\Phi_{pr}\}P\{\Phi_{po}\}$: given a program P beginning in a state satisfying the pre-condition Φ_{pr}, if it terminates, it will do so in a state satisfying the

post-condition Φ_{po}. In the figure, we only focus on key statements that are related to concurrency and lockset: procedure call, fork, join, conditional, and lock operations. In our framework, each program state Δ could consist of thread nodes that capture the final states of child threads. Here final states of child threads refer to post-states of child threads after they finish execution and their delayed formulae that need to be checked at join points. When joined, the post-state of a child thread will be visible and merged into the state of the main thread if its delayed formula is satisfied.

$$partLS(\kappa \wedge [\bigwedge \omega \ \# \ \bigwedge \psi] \wedge \pi) \stackrel{\texttt{def}}{=} (\bigwedge \psi \wedge \pi_1, \kappa \wedge \pi_1)$$
$$\text{where } \pi_1 := removeLS(\pi) \qquad \textbf{AUX}$$

$$partLS(\Phi_1 \vee \Phi_2) \stackrel{\texttt{def}}{=} (\gamma_1 \vee \gamma_2, \Phi_3 \vee \Phi_4)$$
$$\text{where } (\gamma_1, \Phi_3) := partLS(\Phi_1) \text{ and } (\gamma_2, \Phi_4) := partLS(\Phi_2)$$

$$\frac{\begin{array}{c} def(pn) := \texttt{pn}(w^*) \ \textbf{requires} \ \Phi_{pr} \ \textbf{ensures} \ \Phi_{po}; \ \{ \ s \ \} \\ \Delta \vdash \Phi_{pr} \rightsquigarrow \Delta_1 \quad \Delta_2 \stackrel{\texttt{def}}{=} \Delta_1 \ *_{\{\textbf{LS},\textbf{waitlevel}\}} \ \Phi_{po} \end{array}}{\{\Delta\} \ pn(w^*) \ \{\Delta_2\}} \qquad \textbf{L-CALL}$$

$$\frac{\begin{array}{c} spec(pn) := \texttt{pn}(w^*) \ \textbf{requires} \ \Phi_{pr} \ \textbf{ensures} \ \Phi_{po}; \ \{ \ s \ \} \\ (\gamma_{pr}, \Phi'_{pr}) := partLS(\Phi_{pr}) \qquad (\_, \Phi'_{po}) := partLS(\Phi_{po}) \\ \Delta \vdash \Phi'_{pr} \rightsquigarrow \Delta_1 \qquad \Delta_2 \stackrel{\texttt{def}}{=} \Delta_1 * v \mapsto \mathsf{thrd}\langle \gamma_{pr} \rightarrow \Phi'_{po} \rangle \\ \texttt{SAT}(\Phi_{pr} \wedge \textbf{LS}{=}\{\}) \qquad \Phi_{po} \wedge \textbf{LS}{=}\{\} \vdash \textbf{LS}'{=}\{\} \end{array}}{\{\Delta\} \ v := \textbf{fork}(pn, w^*) \ \{\Delta_2\}} \qquad \textbf{L-FORK}$$

$$\frac{\Delta \vdash \gamma_{pr} \quad \Delta_2 \stackrel{\texttt{def}}{=} \Delta * \Phi'_{po} \wedge dead(v)}{\{\Delta * v \mapsto \mathsf{thrd}\langle \gamma_{pr} \rightarrow \Phi'_{po} \rangle\} \ \textbf{join}(v) \ \{\Delta_2\}} \qquad \textbf{L-JOIN-1}$$

$$\{\Delta \wedge dead(v)\} \ \textbf{join}(v) \ \{\Delta \wedge dead(v)\} \qquad \textbf{JOIN-2}$$

$$\frac{\{\Delta \wedge b\} \ s_1 \ \{\Delta_1\} \qquad \{\Delta \wedge \neg b\} \ s_2 \ \{\Delta_2\}}{\{\Delta\} \ \textbf{if} \ b \ \textbf{then} \ s_1 \ \textbf{else} \ s_2 \ \{\Delta_1 \vee \Delta_2\}} \qquad \textbf{COND}$$

$$\{\Delta\} \ \texttt{lock} \ l = \textbf{new} \ \texttt{lock}(v) \ \{\Delta \wedge l{\neq}\texttt{null} \wedge l.mu{=}v \wedge l \notin \textbf{LS}\} \qquad \textbf{NEWLOCK}$$

Figure 4-8: Forward Verification Rules for Concurrency

In order to invoke a procedure call (L–CALL) in a sequential setting, a main thread should be in a state Δ that can entail the pre-condition Φ_{pr} of the procedure *pn*. For brevity, we omit the substitutions that link actual and formal parameters of the procedure prior to the entailment. We also omit the treatment of pass-by-ref parameters which can be handled by applying permissions on variables [85, 112]. After the entailment, the main thread subsumes the post-condition Φ_{po} of the procedure into its state. Note that the operator $*_{\{\mathbf{LS},\mathbf{waitlevel}\}}$ is a "composition with update" operator [103] to capture effects of executing the procedure on **LS** and **waitlevel**.

The auxiliary function *partLS* is used in concurrent settings to partition a formula into a delayed formula γ (which will be "delayed for checking") and a formula Φ. In case of a disjunctive formula, the corresponding delayed formula is also in a disjunctive form. This is to ensure that deadlock-free pre-conditions on lock acquisition can be more precisely guaranteed when "delayed checking". The auxiliary function *removeLS* removes constraints that are related to lockset and waitlevel because they are irrelevant in concurrent settings. The semantics of *removeLS* is straightforward, hence it is not presented.

The rules for fork and join demonstrate the *delayed lockset checking* technique. A **fork** creates a new thread executing concurrently with the main thread. When forking a new child thread (L–FORK), because lockset and waitlevel are local to each thread, the state of the main thread needs not entail constraints related to waitlevel and lockset in the pre-condition Φ_{pr} of the child thread. However, the main thread should be in a state that can entail the formula Φ'_{pr}. The delayed formula γ_{pr} is delayed for checking at a join point. Afterwards, a new thread node carrying the delayed formula γ_{pr} and the post-state Φ'_{po} of the corresponding forked procedure is created. The thread node is then combined with the residue Δ_1 to form a new state Δ_2. Note that constraints related to lockset and waitlevel in the post-condition Φ_{po} are also omitted (resulted in Φ'_{po}) because they are only local to the child thread and are irrelevant to the context of the main thread after the child thread is joined. Lastly, to guarantee the ownership semantics of locks, the L–FORK rule checks if the forked procedure with

an empty lockset in its pre-condition (i.e. $\Phi_{pr} \wedge \mathbf{LS}{=}\{\}$ is satisfiable) will finally end up with an empty lockset in its post-condition (i.e. $\Phi_{po} \wedge \mathbf{LS}{=}\{\} \vdash \mathbf{LS}'{=}\{\}$).

Joining a child thread with an identifier v (L–JOIN–1) requires that the state Δ of the main thread must entail the child thread's delayed formula γ_{pr}. The main thread then merges the post-state of the child thread Φ'_{po} into its state and the child thread disappears from the program state after joined. After a thread has been joined, it becomes dead (indicated by the pure *dead* predicate). Joining a dead thread is equivalent to a no-op, thus it does not incur delayed lockset checking (JOIN–2).

The rule for conditionals (COND) illustrates our support for *precise lockset reasoning*. We capture precise lockset by using disjunction in the post-state of the conditional statement. Together with disjunctive delayed formulae supported by the function *partLS* in L–FORK rule, the use of explicit disjunction in this rule enables more precise reasoning on locksets to ensure deadlock freedom.

Other verification rules are relatively straightforward. The NEWLOCK rule creates a new lock l with a locklevel v. Without specifying a locklevel, a lock is assumed to have an arbitrary non-zero locklevel. We assume that locklevel is immutable during a lock's lifetime. We currently implement acquire/release operations as the following primitives which can be uniformly handled by L–CALL.

```
void acquire(lock l)
  requires [waitlevel<l.mu # l∉LS]
   ensures LS'=LS∪{l};

void release(lock l)
   requires l∈LS
   ensures LS'=LS−{l};
```

The `acquire` primitive requires that locks are acquired in an increasing order of locklevels ($\mathbf{waitlevel}{<}l.mu$). This additionally implies that $l{\notin}\mathbf{LS}$ (but not vice versa). After acquiring the lock l, it is added to the thread's lockset **LS**. Reversely, a thread must hold a lock ($l{\in}\mathbf{LS}$) in order to release it using the `release` primitive.

After releasing the lock l, it is removed from the thread's lockset **LS**. The `acquire` and `release` primitives respectively ensure that a lock is not acquired or released more than once. The rest of verification rules used in our framework only operate in sequential settings, therefore they are standard as described in [103].

Soundness. We now state the main soundness theorem of our framework; detailed proofs are presented in Appendix C. Intuitively, for each program state, there is a wait-for graph corresponding to it. We prove that a program that has been successfully verified by our framework will never get stuck due to deadlocks, i.e. there does not exist a state whose wait-for graph contains a cycle.

Theorem 1 (Soundness)**.** *Given a program with a set of procedures P^i and their corresponding pre/post-conditions (Φ^i_{pr}/Φ^i_{po}), if our verifier derives a proof for every procedure P^i, i.e. $\{\Phi^i_{pr}\}P^i\{\Phi^i_{po}\}$ is valid, the program is deadlock-free.*

4.2.5 Supports for Recursive Locks

Our framework currently supports non-recursive locks. Nonetheless, *precise lockbags* could be integrated into our framework to support recursive locks (e.g. Java's ReentrantLock). In contrast to precise lockset **LS**, precise lockbag (denoted as **LB**) could include multiple occurrences of locks. The specifications for acquire/release operations for recursive locks are as follows:

```
void acquire(lock l)
  requires [waitlevel<l.mu # l∉LB]
   ensures LB′=LB∪{l};
  requires l∈LB
   ensures LB′=LB∪{l};

void release(lock l)
  requires l∈LB
   ensures LB′=LB−{l};
```

For verifying deadlock freedom, acquiring an unheld lock l ($l \notin$ **LB**) should ensure the increasing order of locklevels (**waitlevel**$<l.mu$). Afterward, l is added into **LB**.

If the current thread is holding the lock ($l \in \mathbf{LB}$), it simply adds another occurrence of l into **LB** without incurring the **waitlevel** check. Releasing the lock l removes an occurrence of l from **LB**. The rest of our framework remain unchanged. Note that, for simplicity, Fig. 4-8 includes the verification rules with one pair of pre/post condition. Multiple pre/post specifications can be handled in the same way as [26].

The notation of lockbags has also been used by Haack et al. [52] for reasoning about partial correctness of concurrent programs manipulating Java's reentrant locks. Fu et al. [45] rely on reentrant level (i.e. the number of times a lock has been acquired) for reasoning about reentrant locks in concurrent assembly code. However, in contrast to our framework which verifies deadlock freedom of concurrent programs, the above approaches only focus on partial correctness and do not ensure deadlock freedom.

4.3 Evaluation

We have integrated our framework into separation logic [115] and implemented it into a prototype tool, called ParaHIP[5], for verifying deadlock freedom and partial correctness of programs with fork/join concurrency and non-recursive locks against user-given specifications. To demonstrate the expressiveness of our framework, we did a comparison with Chalice [4, 90, 91], the state-of-the-art framework for verifying deadlock freedom, in terms of deadlock/deadlock-freedom scenarios that can be proven by the respective frameworks. The benchmark programs cover various scenarios such as double lock acquisition, interactions between thread and lock operations, and unordered locking. One scenario (e.g. double acquisition) is representative of many real-world programs. For example, the classical dining philosophers problem can be considered as instances of `ordered-locking` and `unordered-locking`. Therefore, although the scenarios are small, they can be considered as a core benchmark for evaluating expressiveness of deadlock verification systems. The sets of benchmark programs written for both Chalice and ParaHIP are available for online testing in our project website.

[5]The tool is available for both online use and download at `http://loris-7.ddns.comp.nus.edu.sg/~project/parahip/`.

Table 4.1: A Comparison between CHALICE and PARAHIP. A tick (✓) indicates that the corresponding scenario can be verified correctly by the respective verification framework. A cross (✗) indicates otherwise. A prefix "disj" indicates that the corresponding scenario requires disjunctive formulae to precisely capture different execution branches. The third column **Prop** indicates properties of a program, i.e. whether it is prone to deadlock due to double lock acquisition (D1), interactions between thread and lock operations (D2), or unordered locking (D3).

No	Scenario	Prop	CHALICE	PARAHIP	Comments
1	`double-acquire`	D1	✓	✓	CHALICE can handle D1 and D3
2	`ordered-locking`	D3	✓	✓	
3	`unordered-locking`		✓	✓	
4	`no-deadlock1`	D2	✗	✓	CHALICE cannot prove that these programs are deadlock-free
5	`no-deadlock3`		✗	✓	
6	`deadlock1`	D2	✗	✓	False alarms: CHALICE verifies these deadlocked scenarios as deadlock-free
7	`disj-deadlock`		✗	✓	
8	`deadlock2`	D2	✓	✓	CHALICE verifies these programs correctly
9	`deadlock3`		✓	✓	
10	`disj-no-deadlock`		✓	✓	
11	`no-deadlock2`		✓	✓	

The comparison results are presented in Table 4.1. Compared with CHALICE, PARAHIP allows more deadlocks to be prevented and also permits more programs to be declared as deadlock-free. Specifically, CHALICE is unable to correctly verify 4 out of 11 scenarios that express intricate interactions between thread and lock operations. The last column in the table briefly explains the reason behind.

The verification results of CHALICE on D2-type programs are of interest. It confirms our observation that CHALICE is not designed for verifying programs with interactions between fork/join and acquire/release. CHALICE is unable to verify deadlock-free programs (such as `no-deadlock1` and `no-deadlock3`) while it incorrectly verifies deadlocked programs (such as `deadlock1` and `disj-deadlock`) as deadlock-free. Furthermore, since CHALICE does pre-condition checking at the fork points, it is sometimes able to correctly verify D2-type programs such as `deadlock2` and `deadlock3`.

The experimental results were very surprising because CHALICE appears unsound. We communicated this issue with CHALICE' developers and confirmed that CHALICE's technical framework is indeed sound but its implementation does not consider programs with interactions between thread and lock operations [100]. Hence, the

question to investigate is whether CHALICE could be extended to handle those scenarios. To the best of our knowledge, CHALICE technical framework could, under the hood, encode fork/join as send/receive over channels, assign levels to the channels, and require that threads acquire locks and wait on channels in a strictly increasing order of (locks' and channels') levels (Section 4.4 of [91]). With this encoding, CHALICE becomes sound and it can automatically eliminate the false negatives in the programs `deadlock1` and `disj-deadlock`. However, it still does not correctly verify the programs `no-deadlock1` and `no-deadlock3` as deadlock-free. To be more expressive, CHALICE could be extended to allow programmers to explicitly annotate appropriate levels to thread identifiers and require that threads acquire locks and join threads in a strictly increasing order of (locks' and thread identifiers') levels. With this extended help from programmers, CHALICE could correctly verify all programs in Table 4.1. However, there are still programs (such as the program `fork-join-as-send-recv` in our project website) where it is impossible to find appropriate levels to assign to the thread identifiers for proving deadlock freedom. That program can be verified as deadlock-free in our framework without requiring extended help from programmers. Last but not least, while CHALICE is unable to verify programs that involve the multi-join pattern, PARAHIP is capable of handling the multi-join (thank to our "threads as resource" approach proposed in Chapter 3). In summary, compared with CHALICE, our framework is more expressive in handling interactions between fork/join and lock operations. It advocates the use of precise locksets and introduces the delayed lockset checking technique to more expressively prove deadlock freedom.

4.4 Discussion

This section discusses related works on specification and verification of deadlock freedom in shared-memory concurrency. Note that we do not consider non-termination due to infinite loops or recursion. Proving (non-)termination [6, 29, 87] and livelock freedom [108] is orthogonal to our framework, and could be separately extended.

In the context of concurrency verification, several recent frameworks have been proposed to reason about programs with non-recursive locks and dynamically-created threads [51, 61], recursive locks [45, 52], and low-level languages [45], all based on separation logic [115]. However, they focus on verifying partial correctness and ignore the presence of deadlocks. VERIFAST [65] also ignores deadlocks when verifying correctness of concurrent programs. CHALICE [90, 91], a verification framework based on implicit dynamic frames [119], is capable of preventing deadlocks. Initially, CHALICE uses locklevels and is able to prevent deadlocks due to double acquisition and unordered locking [90]. Later development on CHALICE [91] has proposed a technique to prevent deadlocks in programs that use both message passing via channels, and locking. Although it could encode join operations as send/receive over channels, there are programs (such as the program `fork-join-as-send-recv` in our website) where it is impossible for the encoding to find proper levels assigned to the channels for proving deadlock freedom. Our *delayed lockset checking* technique can enable proving deadlock freedom in the presence of interactions between fork/join and acquire/release based on *precise lockset* as an abstraction. Using the technique, we are able to prove more programs deadlock-free than previous work. We also showed how to incorporate our technique with the locklevel idea from CHALICE to form an expressive framework for specifying and verifying deadlock freedom of concurrent programs.

Besides verification frameworks, there are other approaches to detecting or preventing deadlocks in concurrent programs. They can be classified into dynamic and static approaches. There are many systems that detect deadlocks dynamically - see [22, 73, 95] to name just a few recent works on this topic. Dynamic systems have the advantage that they can check unannotated programs. However, they cannot guarantee the absence of deadlocks due to possibly insufficient test coverage. Static approaches such as those based on static analysis [102, 128] and type systems [17, 46, 50, 120] can ensure the absence of certain types of deadlocks. These systems have the advantage that fewer annotations are required. However, they tend to be less expressive than specification logics. Type systems such as [17, 120] use locklevels to

enforce a locking order while others use lock capabilities [50] and continuation effects [46] to verify programs with no natural ordering on the locks acquired. Nevertheless, existing systems [17, 46, 50, 120] do not ensure the absence of deadlocks due to interactions between thread and lock operations. It is interesting to apply our delayed lockset checking technique to enhance the capability of these type systems.

Deadlock-freedom has also been studied in other contexts, and notably in the setting of message-passing process algebra [80, 81, 82]. The notion of locklevels in our approach is similar to obligation and capability levels in these type systems [80, 81, 82]. However, they have only been applied in the context of π-calculus while our framework ensures deadlock freedom for a shared-memory concurrent language with dynamic creation of threads and locks. Although fork/join/acquire/release operations and shared variables could be encoded as send/receive operations over channels, such an encoding would be non-trivial [80, 129].

4.5 Summary

In this chapter, we presented an expressive deadlock-freedom verification framework for concurrent programs. A novel delayed lockset checking technique was introduced to cover deadlock scenarios due to interactions between thread and lock operations. We described an abstraction based on precise lockset to support verification for deadlock freedom. We then showed how our technique can be integrated with locklevels to form a formalism for verifying different deadlock scenarios such as those due to double acquisition, interactions between thread and lock operations, and unordered locking. Lastly, we implemented the proposed framework into ParaHIP, a prototype verifier based on separation logic reasoning, for specifying and verifying deadlock freedom and partial correctness of concurrent programs.

The presented framework provides a foundation towards more comprehensive verification of deadlock freedom. Currently, the framework is capable of reasoning about programs with fork/join and locks. However, besides locks, *barriers* are among

commonly-used concurrency constructs. In the next chapter, we will present our approach to verify *correct synchronization* of programs with barriers. Correct barrier synchronization is a weaker property than deadlock-freedom (i.e. it is deadlock freedom in the presence of a single barrier) and it serves as a starting point for our future work on more comprehensive verification of deadlock-free barrier synchronization.

Chapter 5

Verification of Barrier Synchronization

Overview. Software barriers are a kind of collective operations available in Pthreads, Java, .NET, OpenMP, and others. Threads participating in a barrier proceed in *phases*. A typical usage of barriers is presented in Fig. 5-1. When a thread issues a barrier wait, it waits until a pre-defined number of threads (all threads or just a group of threads) have also issued a barrier wait; after that, all participating threads proceed to the next phase. SPMD (Single Program, Multiple Data) programs, such as those written in OpenMP, typically have a single barrier to coordinate all threads in the programs. On the other hand, fork/join programs written in Pthreads, Java, and .NET could use more than one barrier to coordinate different (possibly non-disjoint) groups of threads. In Pthreads [2], barriers are *static*, i.e. the number of participants is fixed. In .NET framework [43], barriers are *dynamic* as the number of participants can vary during

```
//b has two participants
b = new barrier(2);

//Thread 1        //Thread 2
//Phase 0         //Phase 0
wait(b);          wait(b);
//Phase 1         //Phase 1
```

Figure 5-1: Typical Usage of Barriers

a program's execution. The java.util.concurrent library [49] supports both static and dynamic barriers (i.e. CyclicBarrier and Phaser respectively).

Barriers are commonly used in practice. For example, all twelve programs in SPLASH-2 benchmark suite [130] use at least one barrier and four out of twelve programs use more than one barrier for synchronization, covering numerous application domains such as computer graphics (volrend), water molecule simulation (water-spatial), and engineering (radix) among others. Therefore, verifying correct synchronization of barriers is desirable because it can provide compilers and analysers with important phasing information for improving the precision of their analyses and optimizations such as reducing false sharing [68], may-happen-in-parallel analysis [93, 134], and data race detection [76]. For example, given the information that a program is verified as correctly synchronized on a barrier, concurrency analysers [76, 93, 134] could significantly improve their analyses by exploiting the fact that two statements in different barrier phases cannot be executed in parallel. However, static verification of barrier synchronization in fork/join programs is hard because programmers must not only keep track of (possibly dynamic) number of participating threads, but also ensure that all participants proceed in correctly synchronized phases.

Verification approaches such as those based on separation logic [115] and implicit dynamic frames [119] often use an *access permission system*, such as fractional permissions [18] or counting permissions [15], as the basis for reasoning about race-free sharing of resources. There are *bounded resources* (e.g. barriers) which are typically shared among a *bounded* number (or a group) of concurrent threads. Unfortunately, when using existing permission systems [15, 18], a resource could be split off an unbounded number of times and hence unintentionally shared among an *unbounded* number of concurrent threads. Therefore, existing permission systems are not suitable for reasoning about bounded resources.

In this chapter, we first introduce a new permission system, called *bounded permissions*, to enable reasoning for bounded resources. We then present a logical approach for statically verifying correct synchronization of *static* and *dynamic* barriers

in fork/join programs. For verifying *static barriers*, the approach uses *bounded permissions* and *phase numbers* to keep track of the number of participants and barrier phases respectively. For verifying *dynamic barriers*, the approach introduces *dynamic bounded permissions* to additionally keep track of the additions and/or removals of participants. To the best of our knowledge, our work is the first effort to verify synchronization of both *static* and *dynamic barriers* in fork/join programs.

This chapter is organized as follows. Section 5.1 presents our fork/join programming language with barriers. Section 5.2 presents our approach. Specifically, Section 5.2.1 presents our bounded permission system. Section 5.2.2 shows an application of bounded permissions to verification of static barriers. Section 5.2.3 introduces dynamic bounded permissions and highlights our approach for verification of dynamic barriers. Section 5.2.3 presents our soundness arguments. Section 5.3 discusses our prototype implementation and its application to programs in SPLASH-2 suite. Section 5.4 summarizes related work. Section 5.5 concludes our work in this chapter.

5.1 A Fork/Join Programming Language with Barriers

t	::=	$\ldots \mid$ `barrier`	Type
s	::=	`barrier` $b = \textbf{new}$ `barrier`(n) $\mid$ $\textbf{destroy}(b) \mid \textbf{wait}(b)$ $\mid$ $\textbf{add}(b, m) \mid \textbf{remove}(b, m)$ $\mid$ $\ldots$	Statement

Figure 5-2: Programming Constructs for Barriers

Mainstream languages such as C/C++ (with Pthreads), Java, and .NET provide their own barrier constructs for synchronizing a group of threads. As our approach is language-independent, we use core programming constructs for barriers as presented

in Fig. 5-2. For brevity of presentation, in this chapter, we often use the parallel composition $(s_1||s_2)$; as an abbreviation for creating concurrent threads (we sometimes omit (); due to space limit). The parallel composition is just syntactic sugar which can easily be encoded via fork and join. `barrier` $b =$ **new** `barrier`(n) creates a new barrier b with the number of participants n. **destroy**(b) destroys the barrier b. A thread issues a barrier wait by calling **wait**(b). For dynamic barriers, **add**(b, m) and **remove**(b, m) adds and respectively removes m participants from the existing total number of participants.

5.2 Proposed Approach

5.2.1 Bounded Permissions

In this section, we present our bounded permission system for reasoning about bounded resources. Although we place our bounded permissions in the context of separation logic, bounded permissions can be generally applied to other logics such as implicit dynamic frames [119].

A permission system should distinguish full permission for total control (read, write, and destroy) from partial permission for shared access (read only: no thread can write or destroy) [18]. Permission accounting (e.g. the ability to split a permission into multiple partial permissions for shared access and to combine partial permissions into a full permission for exclusive write) is critical for reasoning about fork/join programs [15]. Besides the above properties, our bounded permission system additionally provides the notion of "*boundedness*" as the guarantee for reasoning about bounded resources.

Fig. 5-3 summarizes our bounded permission system. An assertion $x \overset{c,t}{\longmapsto} E$ represents a bounded permission to access the content E at the address x. A permission quantity is a pair of integers (c, t) where $0<c\leq t$; $c{=}t$ indicates a *full permission* while $c<t$ indicates a *partial permission*. Permissions with $c{=}1$ are called *unit permissions*. A permission can be split into two permissions (reading from left to right of the rule

Bounded permission: $x \overset{c,t}{\longmapsto} E$

Permission count:	c	Full permission:	$c = t$
Permission total:	t	Partial permission:	$c < t$
Permission invariant:	$0 < c \leq t$	Unit permission:	$c = 1$

Permission rules:

[SPLIT/COMBINE] $x \overset{c,t}{\longmapsto} E \land c{=}c_1{+}c_2 \land c_1{>}0 \land c_2{>}0 \iff x \overset{c_1,t}{\longmapsto} E * x \overset{c_2,t}{\longmapsto} E$

[SEP] $x_1 \overset{c_1,t_1}{\longmapsto} E * x_2 \overset{c_2,t_2}{\longmapsto} E \land (t_1{\neq}t_2 \lor c_1{+}c_2{>}t_1) \implies x_1{\neq}x_2$

Figure 5-3: Bounded Permission System

[SPLIT/COMBINE]). In the other direction, heap nodes can be combined using * iff their addresses coincide, they agree on their contents and their permissions can be combined arithmetically. Note that due to the invariant $0{<}c{\leq}t$, a unit permission cannot be split off. Besides the ability to split/combine permissions, the notion of separation ([SEP]) is important for reasoning about separation of resources [15, 115]. Two heaps agreeing on their contents are separated ($x_1{\neq}x_2$) if their permission totals are different or the sum of their permission counts is higher than the permission total.

We can create a new bounded-permission resource (with n being assigned to the permission total) and destroy it only in full permissions:

$$\begin{array}{lll} \{\ n > 0\ \} & \texttt{x = new(n);} & \{\ x \overset{n,n}{\longmapsto} \_\ \} \\ \{\ x \overset{n,n}{\longmapsto} \_\ \} & \texttt{destroy(x);} & \{\ \texttt{emp}\ \} \end{array} \tag{5.1}$$

Given a full permission, we are sure that no other thread can access the shared resource. Therefore, we can safely destroy it. In languages with automatic garbage collection, such a destroy operation is not necessary, but the full permission is still useful in guiding the garbage collector for safe collection.

Similarly, we need a full permission for writing and any permission (full or partial) for reading:

$$\begin{array}{lll} \{\ x \overset{n,n}{\longmapsto} \_\ \} & \texttt{[x] = E;} & \{\ x \overset{n,n}{\longmapsto} E\ \} \\ \{\ x \overset{c,t}{\longmapsto} E\ \} & \texttt{y = [x];} & \{\ x \overset{c,t}{\longmapsto} E \land y = E\ \} \end{array} \tag{5.2}$$

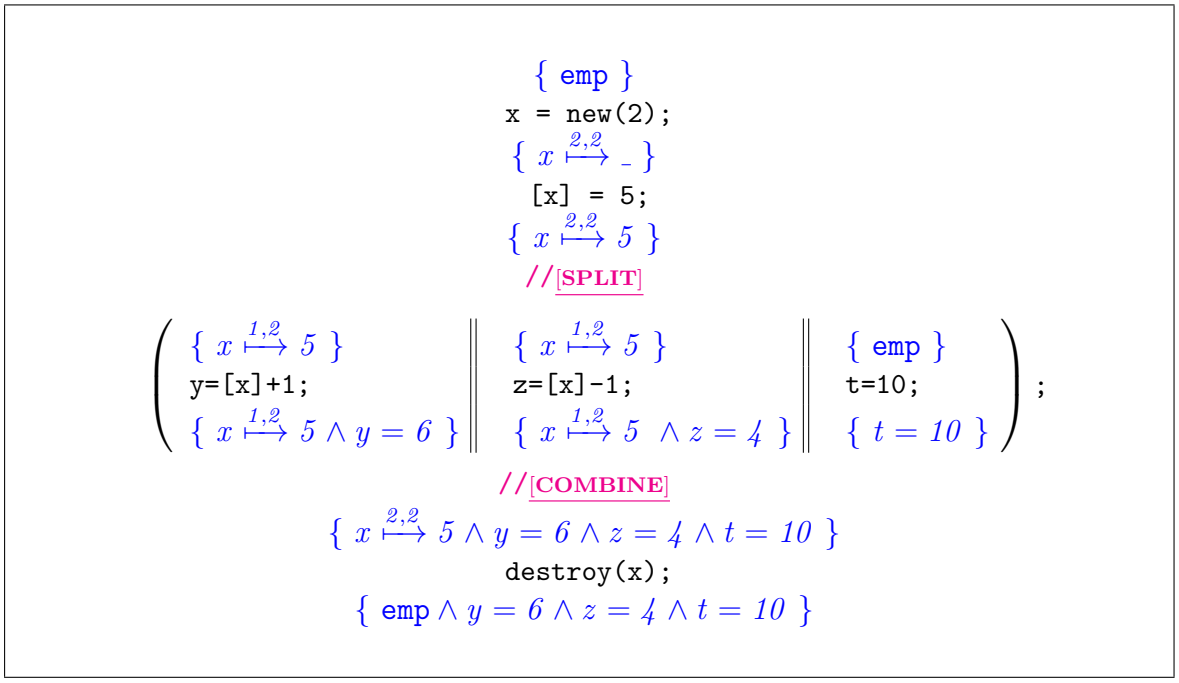

Figure 5-4: Example of Using Bounded Permissions

`[x]` is an abbreviation for accessing the content located at the address `x`. In the last rule, there is a side condition that y is not free in E.

Now, it is straightforward to verify the correctness of the program in Fig. 5-4, in which only two threads are intended to concurrently read the content at the location `x`. As a brief comparison, when using existing permission systems [15, 18], there is nothing to prevent `x` from being split off into more than two partial permissions and hence unintentionally accessed by more than two threads.

The following lemma states our guarantee on boundedness property.

Lemma 2 (Boundedness)**.** *Given a resource x with a full permission $x \overset{n,n}{\longmapsto} \_$ (n>0), there are at most n concurrent accesses to x, i.e. x is shared among at most n concurrent threads at a given time.*

Proof. A thread needs at least a unit permission $x \overset{1,n}{\longmapsto} \_$ to access x and there are at most n such unit permissions. □

5.2.2 Verification of Static Barriers

In this section, we present our approach to verifying correct synchronization of static barriers. We first define what it means for a program to be correctly synchronized.

Definition 3 (Correct Synchronization)**.** *A program is correctly synchronized with respect to a static barrier b iff:*

- *There are exactly a predefined number of threads participating in the barrier b's wait operations.*
- *Participating threads operate on b in the same numbers of phases.*

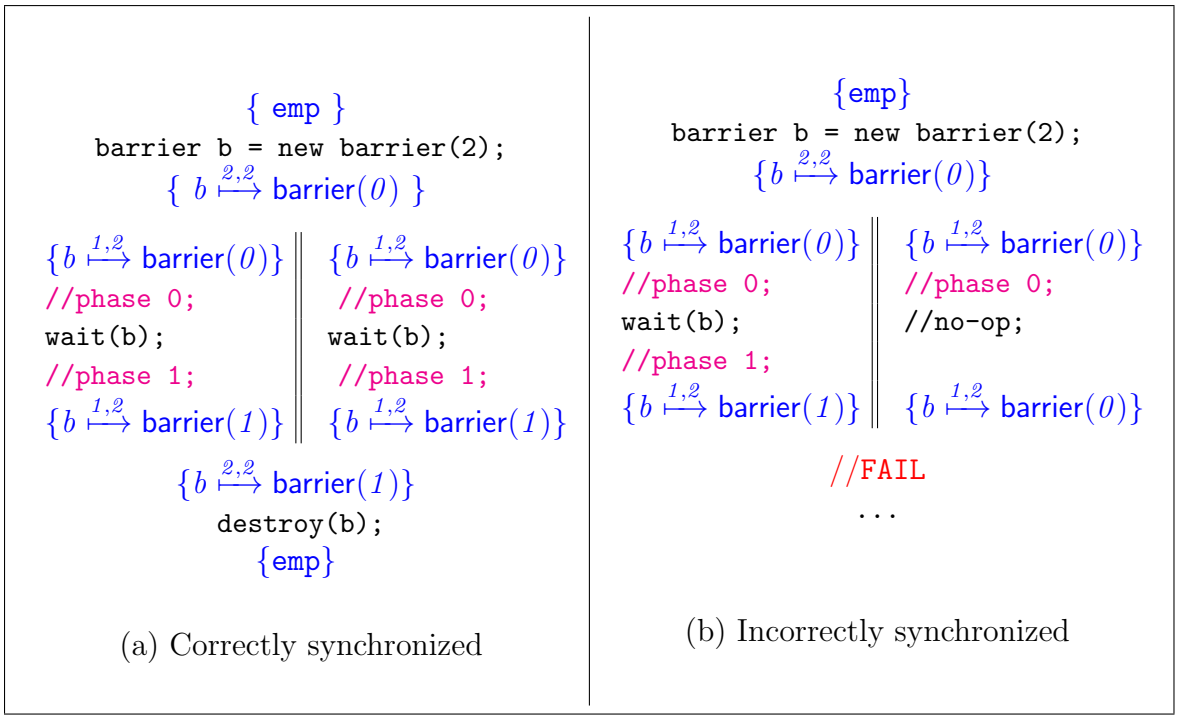

Figure 5-5: Barrier Synchronization

For illustration, the program in Fig. 5-5(a) is correctly synchronized while the program in Fig. 5-5(b) is not because the two threads in Fig. 5-5(b) operate in different numbers of phases. As shown in Section 5.2.1, bounded permissions can be used to

ensure that *at most* a predefined number of threads can access a resource at a given time. However, verification of barrier synchronization requires a stronger guarantee: *exactly* a predefined number of threads participate in a barrier wait. We enforce such a guarantee by requiring that a participating thread must hold a *unit permission* to perform a barrier wait. If a participant has more than a unit permission, it prohibits other participants from participating. An analogy is a meeting room with n keys distributed among n participants; a meeting takes place only when all participants have come. If a participant has more than one key, when he/she enters the room, at least one other participant will not be able to get in and the meeting cannot take place. We capture barrier phasing by using *phase numbers*, which increase by one after each barrier wait, and require that all participants end up with the same phase numbers. If participants have different phase numbers when completing their execution, some of them must have lost phasing and the program is not correctly synchronized.

A summary of our approach is presented in Fig. 5-6. An assertion $b \overset{c,t}{\longmapsto} \mathsf{barrier}(p)$ indicates a bounded permission (c, t) to access the barrier b which is at phase p. When creating a new barrier with the number of participants n, a full permission (i.e. $c{=}t{=}n$) of barrier b is created. We can safely destroy a barrier in its full permission. Waiting on a barrier b requires a unit permission $(1, n)$. This is a contributing factor to certify that there is exactly a predefined number of threads participating in the barrier b. After finishing waiting, the phase number p is increased by 1 and threads proceed to the next phase. The permission rules for split/combine ([S–SPLIT] and [S–COMBINE]) and separation [S–SEP] are similar to those of standard bounded permissions.

Our approach allows for local reasoning where each thread (more precisely each procedure) is verified separately. Intuitively, if threads participate in a barrier b, when they join together, their states must agree on the barrier b. Therefore, we enforce the requirement that concurrent threads must maintain a program in barrier-consistent (or *b-consistent*) states:

Bounded permission: $b \overset{c,t}{\longmapsto} \mathsf{barrier}(p)$

Permission count:	c	Permission invariant:	$0 < c \leq t$
Permission total:	t	Full permission:	$c = t$
Phase number:	p	Partial permission:	$c < t$
		Unit permission:	$c = 1$

Verification rules:

$\{\ n{>}0\ \}$	`barrier b = new barrier(n);`	$\{\ b \overset{n,n}{\longmapsto} \mathsf{barrier}(0)\ \}$
$\{\ b \overset{n,n}{\longmapsto} \mathsf{barrier}(\_)\ \}$	`destroy(b);`	$\{\ \mathsf{emp}\ \}$
$\{\ b \overset{1,n}{\longmapsto} \mathsf{barrier}(p)\ \}$	`wait(b);`	$\{\ b \overset{1,n}{\longmapsto} \mathsf{barrier}(p+1)\ \}$

Permission rules:

[S−SPLIT]

$$b \overset{c,t}{\longmapsto} \mathsf{barrier}(p) \wedge c{=}c_1{+}c_2 \wedge c_1{>}0 \wedge c_2{>}0 \implies b \overset{c_1,t}{\longmapsto} \mathsf{barrier}(p) * b \overset{c_2,t}{\longmapsto} \mathsf{barrier}(p)$$

[S−COMBINE]

$$b \overset{c_1,t}{\longmapsto} \mathsf{barrier}(p) * b \overset{c_2,t}{\longmapsto} \mathsf{barrier}(p) \implies b \overset{c,t}{\longmapsto} \mathsf{barrier}(p) \wedge c{=}c_1{+}c_2$$

[S−SEP]

$$b_1 \overset{c_1,t_1}{\longmapsto} \mathsf{barrier}(p) * b_2 \overset{c_2,t_2}{\longmapsto} \mathsf{barrier}(p) \wedge (t_1{\neq}t_2 \ \vee\ c_1{+}c_2{>}t_1) \implies b_1{\neq}b_2$$

Figure 5-6: Verification of Static Barriers

$$\frac{\begin{matrix}\{\Phi_1\}\ s_1\ \{\Phi_1'\} \\ \{\Phi_2\}\ s_2\ \{\Phi_2'\}\end{matrix} \qquad \begin{matrix}modifies(s_1) \cap FV(\Phi_2, \Phi_2') = \varnothing \\ modifies(s_2) \cap FV(\Phi_1, \Phi_1') = \varnothing\end{matrix} \qquad \Phi_1 * \Phi_2\ is\ b{-}consistent \qquad \Phi_1' * \Phi_2'\ is\ b{-}consistent}{\{\Phi_1 * \Phi_2\}\ s_1 \,||\, s_2\ \{\Phi_1' * \Phi_2'\}} \tag{5.3}$$

Compared with the original parallel composition rule discussed in Section 2.1.2, our parallel composition rule (5.3) additionally requires that concurrent threads begin and end in *b-consistent* states. That is, starting from a consistent state with respect to barriers in the program, threads concurrently operate on the barriers; if they terminate, they do so in a consistent state with respect to the barriers. Informally, a memory state is *b-consistent* if its barrier nodes agree on the phase numbers. After completing their execution, if the threads end up in a joined state $\Phi_1' * \Phi_2'$ which is not b-consistent, the program is rejected as it is incorrectly synchronized. A similar consistency check is also required for the frame rule, which is omitted here since it can be derived from the parallel composition rule (i.e. `s` is equivalent to `s||no-op`).

Definition 4 (Combined State)**.** *A combined state Φ_c of a memory state Φ is achieved by repeatedly applying the* [S–COMBINE] *rule until a fixpoint is reached.*

Such a fixpoint always exists as the [S–COMBINE] rule can only reduce the number of heap nodes.

Lemma 3. *A memory state Φ and its combined state Φ_c are equivalent.*

Proof. Φ_c is derived from Φ using [S–COMBINE] rule and Φ can be derived from Φ_c using [S–SPLIT] rule. □

Definition 5 (b-consistency)**.** *A combined state Φ_c is b-consistent iff for every pair of barrier nodes $b_1 \overset{c_1,t_1}{\longmapsto} \mathsf{barrier}(p_1)$ and $b_2 \overset{c_2,t_2}{\longmapsto} \mathsf{barrier}(p_2)$ in Φ_c, $b_1{=}b_2 \implies p_1{=}p_2$ holds.*

Corollary 2. *A memory state Φ is b-consistent iff its combined state Φ_c is b-consistent.*

Proof. It directly follows from Lemma 3 since Φ and Φ_c are equivalent. □

Example. The memory state $b_1 \overset{1,2}{\longmapsto} \mathsf{barrier}(p_1) * b_2 \overset{1,2}{\longmapsto} \mathsf{barrier}(p_1)$ is b-consistent. However, the memory state $b_1 \overset{1,2}{\longmapsto} \mathsf{barrier}(p_1) * b_2 \overset{1,2}{\longmapsto} \mathsf{barrier}(p_1{+}1)$ is not since, intuitively, it is possible for $b1$ and $b2$ to be aliased and thus the two aliased barrier nodes have inconsistent phase numbers on the same barrier.

We apply our approach to verification of the programs presented in Fig. 5-5. The program in Fig. 5-5(a) can be proven correctly synchronized. When verifying the program in Fig. 5-5(b), our verification system reports a failure when joining the two threads because the joined state is not b-consistent (i.e. the two barrier nodes have different phase numbers).

Fig. 5-7 shows another example which is rather complex due to intricate phasing. Our bounded permissions ensure that there are exactly two threads participating in the barrier `b` while the phase numbers capture exact phasing. Although the two threads operate in different while loops, our notion of phase numbers can certify that the two threads participate in the same numbers of phases. Therefore, the program is correctly synchronized.

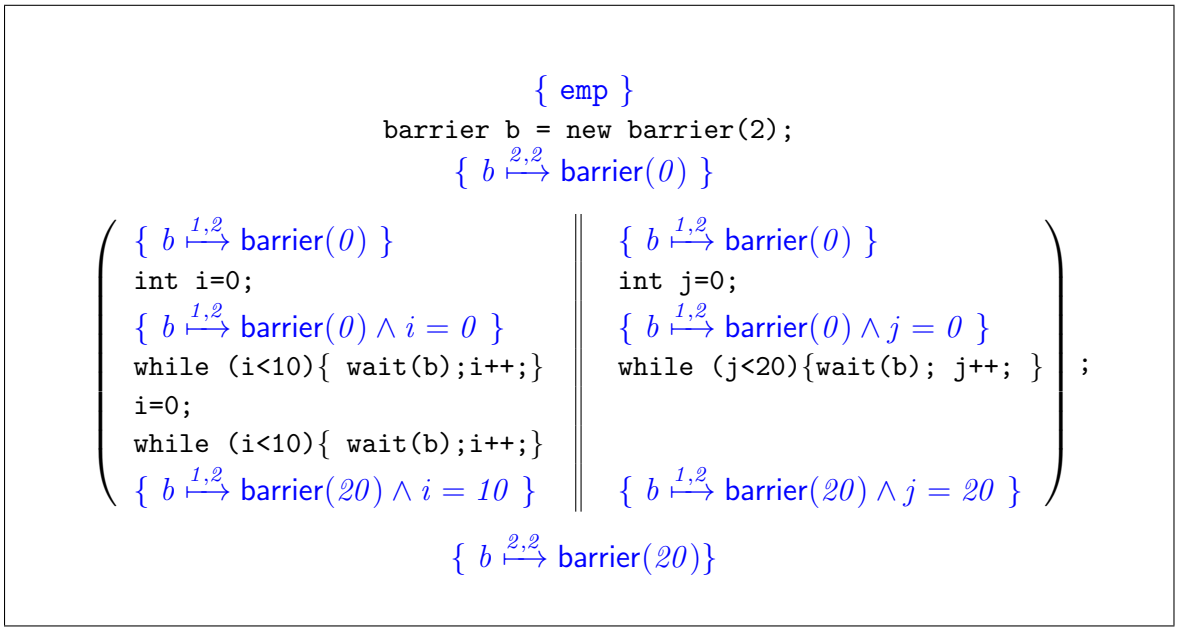

Figure 5-7: More Complex Example

Our approach is also capable of verifying programs with more intricate sharing and nested fork/join, as illustrated in Fig. 5-8. Inside `main`, the main thread creates two child threads executing the procedure `group` on two different barriers `b1` and b2. These two threads do not directly operate on their respective barrier but they create two grand-child threads to participate instead. Consequently, permissions of barrier `b1` and `b2` are transferred from the main thread to child threads and finally to the grand-child threads to create two different groups of grand-child threads participating on two different barriers. Based on the phase numbers, we can verify that threads participate in the same numbers of phases. Note that after joining back the child threads, the main thread gets back the full permissions for `b1` and `b2`. Programmers need not indicate the fact that b1 and b2 are different barriers. Verifiers can use our [S–SEP] rule to infer that information automatically. In the figure, for brevity of presentation, we focus on barrier nodes while ignoring thread nodes which can be easily added using our "threads as resource" approach (described in Chapter 3).

```
void main()
  requires emp
  ensures emp;
{
 { emp }
 barrier b1= new barrier(2);
 barrier b2= new barrier(2);
 { b1 ↦^{2,2} barrier(0) * b2 ↦^{2,2} barrier(0) }
 thrd idg1=fork(group,b1);
 { b2 ↦^{2,2} barrier(0)... }
 thrd idg2=fork(group,b2);
 { ... }
 join(idg1);
 join(idg2);
 { b1 ↦^{2,2} barrier(1) * b2 ↦^{2,2} barrier(1) }
 destroy(b1);destroy(b2);
 { emp }
}
```

```
void participant(barrier b)
   requires b ↦^{1,n} barrier(0)
   ensures b ↦^{1,n} barrier(1);
{ wait(b); }

void group(barrier b)
   requires b ↦^{2,2} barrier(0)
   ensures b ↦^{2,2} barrier(1);
{
 { b ↦^{2,2} barrier(0) }
 thrd id1=fork(participant,b);
 { b ↦^{1,2} barrier(0)... }
 thrd id2=fork(participant,b);
 { ... }
 join(id1);join(id2);
 { b ↦^{2,2} barrier(1) }
}
```

Figure 5-8: Verification of a Program with Static Barriers and Nested Fork/Join

5.2.3 Verification of Dynamic Barriers

Formalism

This section presents our approach to verifying correct synchronization of dynamic barriers. In contrast to static barriers whose number of participants are fixed, dynamic barriers allow the number of participants to be changed during a program's execution. For example, .NET framework allows threads to add and remove m participants to/from a barrier b dynamically via **add**(b, m) and **remove**(b, m).[1] We first present a variant of bounded permissions (called *dynamic bounded permissions*) to keep track of the additions and/or removals of barrier participants of each thread. We then introduce a set of verification and permission rules to reason about dynamic behaviors of dynamic barriers.

A summary of our approach is presented in Fig. 5-9. Compared to the

[1].NET indeed uses AddParticipants() and RemoveParticipants(); we write add() and remove() for brevity.

bounded permission in Section 5.2.1, a dynamic bounded permission of a barrier $b \overset{c,t,a}{\longmapsto} \mathsf{barrier}(p)$ adds an additional component a, called *permission addition*, to keep track of the additions and/or removals of barrier participants issued by each thread. Permission addition a is a rational number since when splitting a dynamic bounded permission, we require that the split-off permissions have proportional shares of a (details to be presented soon). We also introduce the notion of *zero permission* to capture the fact that a thread has dropped its participation to a barrier (c=0) but still retained its information about the addition and/or removals of participants. Our approach guarantees that zero permission can only be achieved by a thread deliberately removing its participation and cannot be produced by a permission split. A permission quantity (c, t, a) captures the local view of a thread on the barrier. With the presence of permission addition a, the full permission is achieved when $c = t + a$. Intuitively, the current number of participants is equal to the original number of participants plus the number of participants added or removed. One could recognize that dynamic bounded permission and bounded permission coincide when a=0.

The verification rules in Fig. 5-9 capture dynamic behaviors of dynamic barriers. Creating a new barrier results in a full permission of the barrier with a=0. Destroying a barrier requires a full permission (c=t+a). Waiting at a barrier requires a unit permission (c=1). Adding and removing m participants add and respectively subtract m from the permission count and the permission addition. The permission total t remains unchanged; it acts as a pivot for combining permissions when threads join together. A thread can only remove up to the permission count it has ($c \geq m$). If c=m, after removing, a thread is considered dropping its participation to the barrier. Adding participants requires c>0 to ensure that a drop-out thread could not re-participate in a barrier. This is necessary because when dropping out, a thread has lost phasing with other participants; therefore, it is unsafe to allow it to re-participate. An example program is presented in Fig. 5-10.

Due to the nature of dynamic barriers, a thread could either fully participate in a barrier or drop its participation in the middle of its execution. Permission rules

Dynamic bounded permission: $b \overset{c,t,a}{\longmapsto} \mathsf{barrier}(p)$

Permission count:	c	Permission invariant:	$0 \le c \le t + a$
Permission total:	t	Full permission:	$c = t + a$
Permission addition:	a	Partial permission:	$0 < c < t + a$
Phase number:	p	Unit permission:	$c = 1$
		Zero permission:	$c = 0$

Verification rules:

$$\{n{>}0\}\ \texttt{b = new barrier(n);}\ \{b \overset{n,n,0}{\longmapsto} \mathsf{barrier}(0)\}$$

$$\{b \overset{c,t,a}{\longmapsto} \mathsf{barrier}(\_) \wedge c{=}t{+}a\}\ \texttt{destroy(b);}\ \{\mathsf{emp}\}$$

$$\{b \overset{1,t,a}{\longmapsto} \mathsf{barrier}(p)\}\ \texttt{wait(b);}\ \{b \overset{1,t,a}{\longmapsto} \mathsf{barrier}(p+1)\}$$

$$\{b \overset{c,t,a}{\longmapsto} \mathsf{barrier}(p) \wedge c{>}0 \wedge m{>}0\}\ \texttt{add(b,m);}\ \{b \overset{c+m,t,a+m}{\longmapsto} \mathsf{barrier}(p)\}$$

$$\{b \overset{c,t,a}{\longmapsto} \mathsf{barrier}(p) \wedge c{\ge}m \wedge m{>}0\}\ \texttt{remove(b,m);}\ \{b \overset{c-m,t,a-m}{\longmapsto} \mathsf{barrier}(p)\}$$

Permission rules:

[D−SPLIT]

$$b \overset{c,t,a}{\longmapsto} \mathsf{barrier}(p) \wedge 0{<}c{\le}t{+}a \wedge 0{<}c_1{<}t{+}a_1 \wedge 0{<}c_2{<}t{+}a_2 \wedge c{=}c_1{+}c_2 \wedge a{=}a_1{+}a_2 \wedge a_1{=}\tfrac{c_1}{c}{\cdot}a \wedge a_2{=}\tfrac{c_2}{c}{\cdot}a \implies b \overset{c_1,t,a_1}{\longmapsto} \mathsf{barrier}(p) * b \overset{c_2,t,a_2}{\longmapsto} \mathsf{barrier}(p)$$

[D−COMBINE−1]

$$b \overset{c_1,t,a_1}{\longmapsto} \mathsf{barrier}(p) * b \overset{c_2,t,a_2}{\longmapsto} \mathsf{barrier}(p) \wedge c_1{\neq}0 \wedge c_2{\neq}0 \implies b \overset{c,t,a}{\longmapsto} \mathsf{barrier}(p) \wedge c{=}c_1{+}c_2 \wedge a{=}a_1{+}a_2$$

[D−COMBINE−2]

$$b \overset{c_1,t,a_1}{\longmapsto} \mathsf{barrier}(p_1) * b \overset{c_2,t,a_2}{\longmapsto} \mathsf{barrier}(p_2) \wedge c_1{=}0 \implies b \overset{c,t,a}{\longmapsto} \mathsf{barrier}(p_1) \wedge c{=}c_1{+}c_2 \wedge a{=}a_1{+}a_2 \wedge p{=}max(p_1,p_2)$$

[D−FULL]

$$b \overset{c,t,a}{\longmapsto} \mathsf{barrier}(p) \wedge c{=}t{+}a \wedge a{\neq}0 \wedge c{>}0 \implies b \overset{c,t+a,0}{\longmapsto} \mathsf{barrier}(p)$$

[D−SEP]

$$b_1 \overset{c_1,t_1,a_1}{\longmapsto} \mathsf{barrier}(p_1) * b_2 \overset{c_2,t_2,a_2}{\longmapsto} \mathsf{barrier}(p_2) \wedge (t_1{\neq}t_2 \ \vee\ c_1{+}c_2{>}t_1{+}a_1{+}a_2) \implies b_1{\neq}b_2$$

Figure 5-9: Verification of Dynamic Barriers

in Fig. 5-9 capture those dynamic behaviors. The rule [D−SPLIT] never splits into zero permissions; therefore, it ensures that a zero permission only appears due to a thread's drop-out. The rule also ensures that a full permission is never created by splitting a partial permission since it requires that the two split-off permissions

have proportional shares of a; that is $a_1{=}\frac{c_1}{c}{\cdot}a$ and $a_2{=}\frac{c_2}{c}{\cdot}a$. We provide the proof for this claim in Appendix D. When multiple threads join, some of them have fully participated in the barrier b while others might drop out midway. Therefore, the combine rules have to take into consideration several situations. First, combining two fully participating threads ($c_1{\neq}0$ and $c_2{\neq}0$) adds up their permission counts and permission additions ([**D–COMBINE–1**]). Because of their full participation, their phase numbers should be equal (both are p). Second, combining two threads while at least one of them dropped out ($c_1{=}0$) will pick up the maximum between their phase numbers ([**D–COMBINE–2**]). Intuitively, if a thread has dropped its participation in the middle of an execution, it did not participate in some later phases; therefore, its phase number is at most that of a fully-participating thread. The rule [**D–FULL**] reshuffles the full permission into an equivalent form. The rule [**D–SEP**] introduces the notion of separation in the context of dynamic bounded permissions.

Similar to static barriers, in order to ensure correct synchronization of dynamic barriers and to support local reasoning, our approach also requires that concurrent threads maintain a program in dynamic-barrier-consistent (*db-consistent*) states. However, the check for *db-consistency* is slightly more complex than that of *b-consistency* because in case of dynamic barriers the phase numbers of different barrier nodes of the same barrier need not be the same (due to the addition and removal of participants). Note that since dynamic barriers subsume static barriers (i.e. when $a = 0$), the definition of *db-consistency* also subsumes that of *b-consistency*

Definition 6 (Combined State)**.** *A combined state Φ_c of a memory state Φ is achieved by repeatedly applying the* [**D–COMBINE–1**] *and* [**D–COMBINE–2**] *rules until a fixpoint is reached.*

Lemma 4. *A memory state Φ and its combined state Φ_c are equivalent.*

Proof. Φ_c is derived from Φ using [**D–COMBINE–1**] and [**D–COMBINE–2**] rules. Φ can be derived from Φ_c using [**D–SPLIT**] and the following [**D–SPLIT2**] rule, which is modified from [**D–SPLIT**] rule to additionally allow splitting off zero permissions:

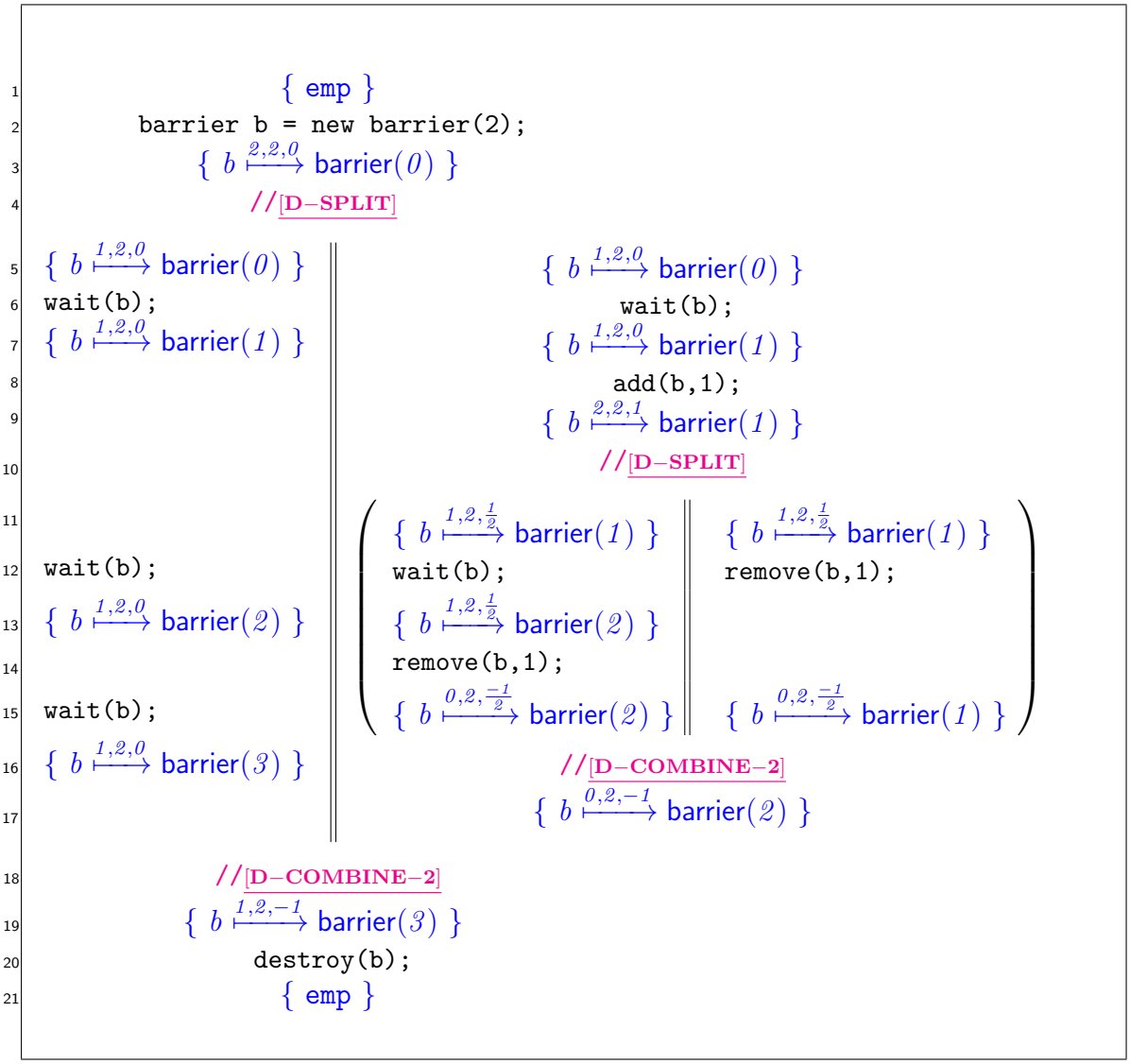

Figure 5-10: An Example of Verifying Synchronization of Dynamic Barriers

$$\frac{}{b \overset{c,t,a}{\longmapsto} \mathsf{barrier}(p) \wedge 0{<}c{\leq}t{+}a \wedge 0{<}c{\leq}t{+}a_1 \wedge 0{<}t{+}a_2 \wedge a{=}a_1{+}a_2 \wedge p{=}max(p_1,p_2) \implies b \overset{c,t,a_1}{\longmapsto} \mathsf{barrier}(p_1) * b \overset{0,t,a_2}{\longmapsto} \mathsf{barrier}(p_2)}\ [\textbf{D−SPLIT2}]$$

□

Definition 7 (db-consistency). *A combined state Φ_c is db-consistent iff for every pair of dynamic barrier nodes $b_1 \overset{c_1,t_1,a_1}{\longmapsto} \mathsf{barrier}(p_1)$ and $b_2 \overset{c_2,t_2,a_2}{\longmapsto} \mathsf{barrier}(p_2)$ in Φ_c, the*

following assertion holds:

$b_1{=}b_2 \Longrightarrow (\ (c_1{\neq}0 \wedge c_2{\neq}0 \wedge p_1{=}p_2) \vee (c1{=}0 \wedge p_1{\leq}p_2) \vee (c2{=}0 \wedge p_2{\leq}p_1)\ )$

Corollary 3. *A memory state* Φ *is db-consistent iff its combined state* Φ_c *is db-consistent.*

Proof. It directly follows from Lemma 4 since Φ and Φ_c are equivalent. □

Fig. 5-10 presents the proof outline of a program with dynamic barriers. The leftmost thread fully participates in `b` while the right thread participates in one phase, then adds another participant (line 8), and creates two child threads operating on `b`. The left child thread drops out after one phase while the right child thread drops out without participation. At the end of the parallel compositions, the permissions are combined together into a full permission. In our approach, for local reasoning, each thread is verified separately and is unaware of operations (such as add/remove) performed by other threads until they join together. Although sound, our approach is incomplete since it could reject programs that are correct at run-time. However, we believe that our static verification is generally a good practice for programmers to follow in order to avoid unexpected run-time behaviors, as pointed out next.

Static Verification as Good Practice

Our approach can statically verify that a program is correctly synchronized in the presence of static and dynamic barriers. For local reasoning, each thread is verified separately and has its own view on a barrier b (reflected in the bounded permission of b that it owns). Thus, a thread is unaware of operations (such as add/remove) performed by other threads until they join together. Although sound, our approach is incomplete since it could reject programs that are correct at run-time. For example, our static verification (with local reasoning) does not allow the program in Fig. 5-11(a) where the left thread is intended to remove the participation of the right thread. However, we believe that a more desirable way to implement this program is to let

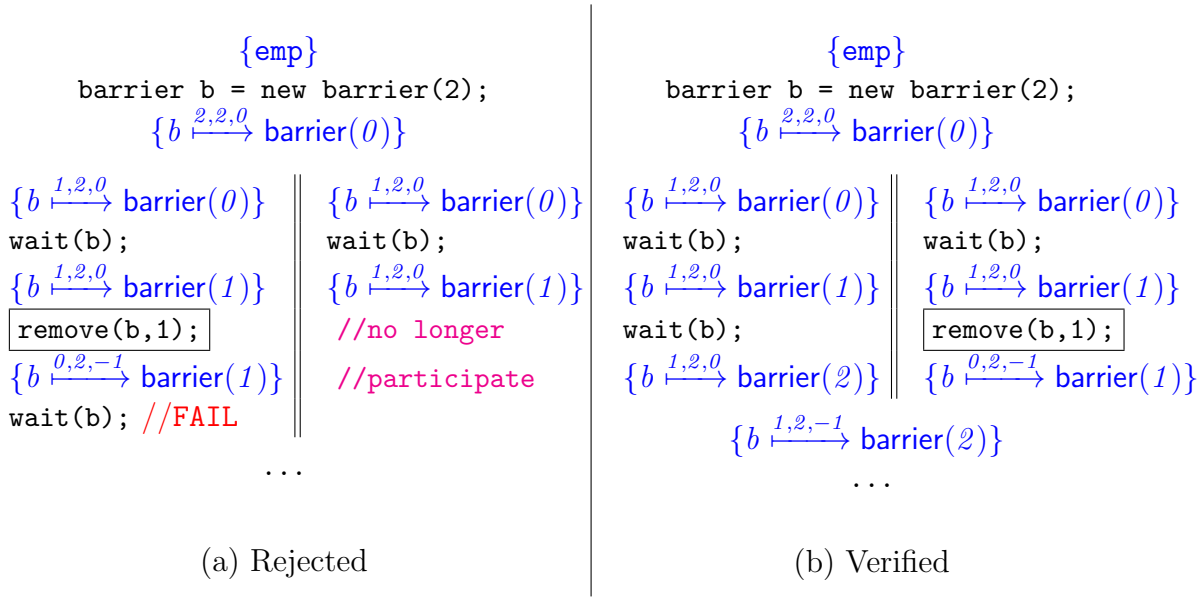

Figure 5-11: Dynamic Behaviors of Dynamic Barriers

the right thread deliberately drop its participation (as depicted in Fig. 5-11(b)). This more intuitive coding style is readily captured by our approach.

In many cases, our static verification is helpful for preventing harmful behaviors at run-time such as deadlocks due to inter-thread addition/removal of participants. One example is the program presented in Fig. 5-12(a) where the left thread adds one participant to the barrier `b` while the right thread creates one more thread participating in `b`. The programmer's intention is that, after adding one more participant, there will be three threads concurrently operating on the barrier. Unfortunately, the program is potentially deadlocked due to the following interleaving: $1 \mapsto 4 \mapsto 5 \mapsto 6 \mapsto 2 \mapsto 3$. In this interleaving, the left thread waits forever at statement 3 because it has to wait for two other participants to issue a barrier wait, though they have already completed their execution. Another example is the program in Fig. 5-12(b) where the left thread removes one participant while the right thread concurrently adds one participant. Although the total number of participants remains unchanged, the program is potentially deadlocked due to the interleaving $1 \mapsto 4 \mapsto 2 \mapsto 3 \mapsto 5 \mapsto 6$. Fortunately, such error-prone programs with inter-thread addition/removal of

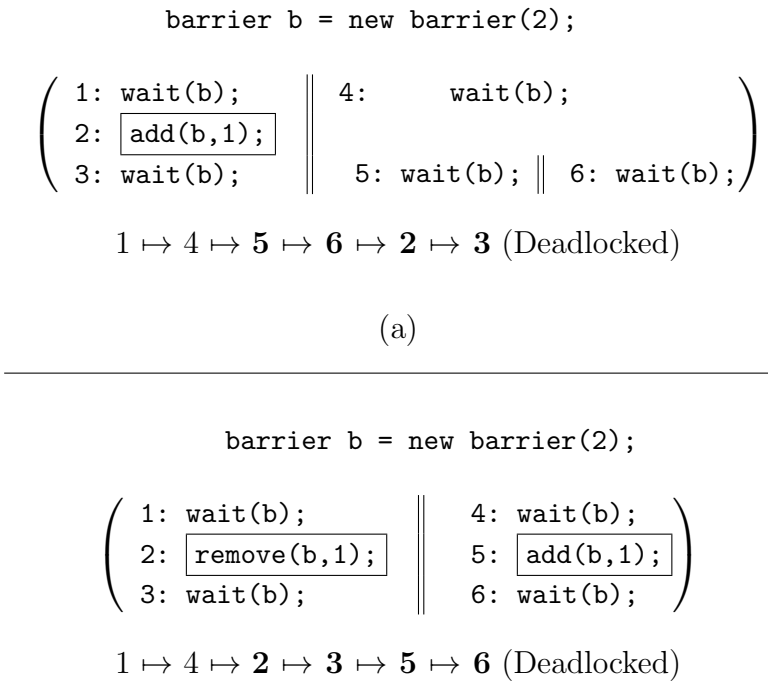

Figure 5-12: Potential Deadlocks due to Inter-thread Addition/Removal of Participants

participants are rejected by our approach. In summary, we believe that our static verification is generally a good practice for programmers to follow in order to avoid unexpected run-time behaviors.

Soundness

We show that our proposed approach guarantees correct synchronization of dynamic barriers. As dynamic barriers are more general than static barriers, the soundness also implies correct synchronization of static barriers. We first present an encoding of join operations in terms of barrier operations. This encoding simplifies the proof rules and soundness arguments to only focusing on barrier operations. We then proceed to the main soundness arguments of our approach. We now state the main soundness lemma; detailed definitions and proofs can be found in Appendix D.

Lemma 5 (Soundness of Verifying Barrier Synchronization)**.** *Given a program with a barrier b and a set of procedures P^i together with their corresponding pre/post-conditions (Φ^i_{pr}/Φ^i_{po}), if our verifier derives a proof for every procedure P^i, i.e. $\{\Phi^i_{pr}\}P^i\{\Phi^i_{po}\}$ is valid, then the program is correctly synchronized with respect to the barrier b.*

5.3 Experiments

We implemented our approach into our prototype verifier [2] for separation logic reasoning. We applied it to verifying static[3] barrier synchronization of all twelve simplified[4] programs in SPLASH-2 suite [130] against user-given specifications. SPLASH-2 suite is one of the most widely-used benchmarks for evaluating shared-memory systems. The suite consists of twelve realistic programs covering numerous application domains such as computer graphics (`volrend`), signal processing (`fft`), water molecule simulation (`water-spatial`), and general engineering (`radix`) among others. Besides the theoretical contributions, the empirical question we investigate is how well our approach handles intricate barrier synchronization. The results were promising as our approach was able to verify all but one program in SPLASH-2 suite with modest annotation. All experiments were done on a 3.20GHz Intel Core i7-960 processor with 12GB memory running Ubuntu Linux 14.04. The suite of benchmark programs and other examples are provided in our project website.

The experimental results are presented in Table 5.1. The column *#Bar* shows the number of barriers used in the corresponding program. The column *LOC* shows

[2]The tool is available for both online use and download at `http://loris-7.ddns.comp.nus.edu.sg/~project/veribsync/`.

[3]As dynamic barriers have just been available recently since .NET 4.0 (April 2010) and Java 7 (July 2011), we are not aware of existing concurrency benchmarks that use dynamic barriers. Nonetheless, we applied our prototype on a set of textbook programs which represent typical usage of dynamic barriers. The programs are available in our project website.

[4]As verifying full functional correctness of these programs is beyond the scope of this chapter, our experiments were conducted on a set of simplified programs where parts of programs that are not related to barriers were omitted. All related parts such as branching conditions and loops were retained to ensure that barrier synchronizations in the simplified programs are similar to those of the original programs.

Table 5.1: Annotation Overhead and Verification Time of SPLASH-2 Suite

Program	Description	#Bar	LOC	LOAnn	Overhead	Time
`ocean`	large-scale ocean simulation	1	60	5	8%	1.53
`radix`	integer radix sort	2	68	7	10%	13.56
`lu`	blocked LU decomposition	1	79	12	15%	8.60
`barnes`	Barnes-Hut for N-body problem	1	84	12	14%	1.63
`raytrace`	optimized ray tracing	1	94	7	7%	0.74
`fft`	complex 1D FFT	1	101	8	8%	1.55
`water-nsquared`	water simulation w/o spatial structure	3	113	16	14%	11.66
`water-spatial`	water simulation w/ spatial structure	3	117	18	15%	11.65
`cholesky`	blocked sparse cholesky factorization	1	131	10	8%	0.70
`fmm`	adaptive fast multipole for N-body	1	175	20	11%	1.66
`volrend`	optimized ray casting	2	232	36	16%	17.86
`radiosity`	hierarchical diffuse radiosity method	1	83	-	-	-
Average	-	-	-	-	11%	6.47

the total number of non-blank, non-comment, non-annotation lines of source code, counted by `sloccount` (v2.26). The column *LOAnn* shows the total number lines of annotation. Annotation *overhead* is computed as $\frac{LOAnn}{LOC}$ (the lower, the better). Verification times are in seconds. Our verifier was able to verify barrier synchronization of all but one program in SPLASH-2 suite with the verification time of several seconds. We discuss the reason why our verifier was not able to verify `radiosity` program in Section 6.2. The verification time and annotation overhead depend on characteristics of the programs. Programs that have complicated non-linear constraints and/or use multiple barriers in many execution branches (such as `radix`, `lu`, `water-*`, and `volrend`) require higher verification time and annotation overhead. On average, our verifier requires annotation overhead of 11%, which is modest compared with that of 100% reported in the literature [63].[5] Much of the annotation and verification time are dedicated for functional correctness properties of the programs such as branching conditions and loops. As annotation efforts for these properties are also necessary for verifying functional correctness of concurrent programs, we believe that existing logics for verifying functional correctness can easily integrate our approach into their logics and benefit from our guarantee of correct barrier synchronization.

[5]To be precise, the annotation overhead in [63] also includes the specification for functional correctness. Although verifying functional correctness is not our main goal, we also need to specify them for verifying barrier synchronization.

5.4 Discussion

This section discusses related works regarding access permission systems and static verification of barrier synchronization. We also discuss related works regarding other advanced forms of barriers such as X10's clocks and phasers [117, 118] which have recently been introduced in the context of async/finish programs.

Access Permissions

Boyland first introduced fractional permissions for reasoning about non-interference of concurrent programs [18]. Bornat et. al. added counting permissions [15]. Recently, various permission systems such as binary tree share model [34], Plaid's permission system [11], and borrowing permissions [101] have been proposed. In a nutshell, they are akin to fractional and counting permissions.

Importantly, not every program is suitable for fractional permissions and counting permissions. Programs that allow sharing resources among only a bounded number of threads need another alternative treatment. Fractional and counting permissions could not reason about those programs because, when using these permission systems, there is nothing to prevent a resource from being split off an unbounded number of times and shared among an unbounded number of threads. Given any fractional permission f where $0<f\leq 1$, it is always possible to split f into two fractions f_1 and f_2 where $f_1+f_2=f$ and $f_1, f_2>0$. Similarly, in counting permissions, given a central *permission authority* holding a source permission n, it is always possible to split off into a new source permission $n+1$ (held by the central authority) and a read permission -1 for sharing. On the other hand, in our bounded permission system, any non-unit permission (c, t) where $1<c\leq t$ (either partial or full permissions) can be split off without the presence of a central authority, and a bounded permission can only be split off a bounded number of times (up to unit permissions). Therefore, bounded permissions enable reasoning about bounded resources such as barriers.

Verification of Barrier Synchronization

Most existing works on verifying barrier synchronization focus on SPMD programs [5, 68, 76, 77, 93, 133, 134]. In SPMD programs, the fact that threads execute the

same code makes verification more tractable. SMPD programs also assume that barriers are global and all threads need to participate in barrier operations. Hence, existing techniques for SPMD programs cannot be directly applied to fork/join programs. This work fills in the gap and addresses barriers in the context of fork/join concurrency where concurrent threads could execute different pieces of code while participating in barrier operations. Furthermore, we do not restrict that all threads should participate, i.e. a group of threads can participate in a certain barrier. We also support verification of dynamic barriers whose number of participants can vary during a program's execution. We are not aware of any related works capable of verifying dynamic barriers in fork/join programs.

To the best of our knowledge, the most closely related work is by Hobor and Gherghina [63]: they propose a specification logic for verifying partial correctness of programs with static barriers. Based on the global *phase transition specification* of a barrier, they can also verify that participants proceed in correct phases. However, there are several critical differences. First, they do not handle dynamic barriers. Second, they require a global specification of each barrier, whereby programmers have to specify pre-state and post-state for each thread for every phase transition over the barrier. However, there are programs (such as that in Fig. 5-7) where our approach using phase numbers can verify, but it is not possible to capture a global specification for its barrier [47]. Though the global specification of each barrier is an extra annotation burden, they can facilitate resource re-distribution at synchronization points to ensure functional partial correctness. Our current approach using phase numbers is considerably simpler, but has not yet been designed to support resource re-distribution. This may be important for more complex usage of barrier synchronization.

Advanced Forms of Barriers

There are various implementations of barriers [56], and several implementations have been verified in [96]. Our specification in Fig. 5-6 and 5-9 can serve as a common interface for verifying different implementations. Besides traditional barriers, other

advanced forms of barriers such as X10's clocks and phasers [117, 118] are also used in the context of async/finish programs. Note that Java 7's Phaser [49] only includes a subset of capabilities of the phasers proposed in [118], i.e. Java 7's Phaser is similar to dynamic barriers used in .NET [43] (which are the main topic of Section 5.2.3). Compared with traditional barriers, clocks and phasers are more dynamic in nature and are only applied to the more tractable context of async/finish programs.

Barrier synchronization in async/finish programs is generally more tractable than that in fork/join programs for two main reasons. First, thread creation and join in async/finish programs are lexically-scoped while those in fork/join are non-lexically-scoped, i.e. fork and join operations can be invoked in different program scopes. Second, there are restrictions on the usage of clocks and phasers in async/finish programs [117, 118]. For example, in X10 programs, a newly-spawn thread has to explicitly register and directly operate on a clock, and it can only register to the clock that its parent has already registered to. These restrictions reject many useful programs such as those with nested and/or non-lexical fork/join concurrency. On the other hand, in fork/join programs written in mainstream languages such as C/C++ (with Pthreads), Java, and .NET, there aren't such restrictions. A new thread does not need to register but can still freely own or pass a barrier to other threads. Because of these reasons, one could not directly apply analyses and verification techniques of clocks in async/finish programs (e.g. those in [74, 97]) to traditional barriers in fork/join programs. In contrast, we conjecture that one could adapt our proposed approach to statically verifying correct synchronization of clocks and phasers.

5.5 Summary

We described a specification and verification approach for ensuring correct synchronization of software barriers. Barriers, provided by many mainstream languages such as C/C++ (with Pthreads), Java, and .NET, are hard to handle in fork/join programs because programmers must not only pay special attention to the (possibly dynamic) number of participating threads, but also ensure that threads proceed in correctly

synchronized phases. To our knowledge, this is the first work that statically ensures the correct synchronization of both *static* and *dynamic* barriers in fork/join programs. The keys of our approach are the *bounded permissions* and *phase numbers* to keep track of the number of participating threads and barrier phases respectively. Not restricted to only barriers, bounded permissions can be generally used to reason about any resources that are shared among a bounded number of concurrent threads. Our approach has been proven sound, and a prototype of it has been implemented for verifying barrier synchronization of all but one of the simplified programs in SPLASH-2 benchmark suite within several seconds.

Chapter 6

Conclusions and Future Work

6.1 Thesis Summary

In today's multi-cores era, verification of shared-memory concurrent programs has become an important research challenge to improve reliability of software systems. In this thesis, we addressed key aspects of verifying concurrent programs, namely verifying partial correctness, data-race freedom, and synchronization properties (i.e. deadlock freedom and correct barrier synchronization) of shared-memory concurrent programs manipulating commonly-used constructs such as fork/join, locks, and barriers. This thesis makes the following three major contributions. First, we proposed the *threads as resource* approach for verifying partial correctness and data-race freedom of programs with first-class threads using fork/join concurrency. The approach enables flexible treatment of threads and allows for threads' liveness to be explicitly tracked. The "threads as resource" approach provided an infrastructure for our subsequent contributions on verification of deadlock freedom and barrier synchronization. Second, we developed an expressive framework for verifying different deadlock-freedom scenarios including double lock acquisition, interactions between thread fork/join and lock acquire/release, and unordered locking. In particular, our framework guaranteed deadlock freedom of programs with interactions between thread fork/join and lock acquire/release operations, which have not been fully studied. Third, we introduced

an approach using *bounded permissions* for verifying both static and dynamic barrier synchronization in fork/join programs. Static verification of barrier synchronization is desirable but hard and has almost been neglected in the context of fork/join programs. In this thesis, we provided the first approach for verifying both static and dynamic barrier synchronization in fork/join programs.

Our main contributions are detailed below.

1. **Threads as Resource:** In mainstream languages, threads are first-class in that they can be dynamically created, stored in data structures, passed as parameters, and returned from procedures. Reasoning about first-class threads is challenging because threads are dynamic and non-lexically-scoped in nature. A thread can be dynamically created in a procedure (or a thread), but shared and joined in other procedures (or threads). There exist approaches that can support threads in fork/join programs, e.g. [51, 60, 65, 91]. These approaches support reasoning about threads in a restricted way where threads are often represented by unique tokens that can neither be split nor shared. As such, they do not fully consider threads as first-class and are incapable of verifying intricate fork/join behaviors such as the *multi-join* pattern.

   In Chapter 3, we proposed "*threads as resource*" to support more expressive treatment of first-class threads. Our approach allows the ownership of a thread (and its resource) to be flexibly split, combined, and (partially) transferred across procedure and thread boundaries. We illustrated the utility of our approach in handling three problems. First, we verified the *multi-join pattern* where threads are shared among concurrent threads and are joined multiple times in different threads. Second, using inductive predicates, we showed how our approach naturally captures the *threadpool idiom* where threads are stored in data structures. Lastly, we presented how thread liveness is precisely tracked. To demonstrate the feasibility of the approach, we implemented it in a tool. Experimental results demonstrated its expressiveness while achieving reasonable verification performance. The tool was able to verify a set of small-sized

but intricate concurrent programs in less than three seconds. The "threads as resource" approach provided an infrastructure for our subsequent works on verifying deadlock freedom and barrier synchronization. Last but not least, inspired by the notion of "threads as resource", we presented our proposed "*flow-aware resource predicate*", a variant of Concurrent Abstract Predicates (CAP) [33, 36, 121]. Flow-aware resource predicates explicitly track resources that flow into and out of their shared abstraction. Resources of such a predicate can be more flexibly split and transferred across procedure and thread boundaries, in a similar way to "threads as resource". This allows for verification of various concurrency mechanisms, including and beyond first-class threads.

2. **Verification of Deadlock Freedom:** Several recent verification frameworks have been proposed to reason about concurrent programs with dynamically-created threads and locks [45, 51, 52, 61]. Most are designed to ensure program correctness but make no explicit mention of potential deadlock problems. This is a kind of partial correctness consideration whereby non-termination and deadlock problems are ignored. Chalice [90, 91], the state-of-the-art deadlock verification framework, uses locklevels as a handle to prevent deadlocks that arise from double acquisition and unordered locking. However, Chalice does not fully guarantee deadlock freedom, as it is not designed to handle deadlocks due to the interactions between thread and lock operations.

   In Chapter 4, we proposed an expressive specification and verification framework for ensuring deadlock freedom of programs that manipulate non-recursive locks. We introduced a novel *delayed lockset checking* technique to guarantee deadlock freedom of programs with interactions between thread and lock operations. With disjunctive formulae, we highlighted how an abstraction based on *precise lockset* can be supported in our framework. By combining our technique with locklevels, we proposed a *unified formalism* for ensuring deadlock freedom from (1) double lock acquisition, (2) interactions between thread fork/join and lock acquire/release, and (3) unordered locking. We conducted an experimental

comparison with Chalice on a set of hand-crafted programs covering various scenarios such as double lock acquisition, interactions between thread and lock operations, and unordered locking. Experimental evaluation showed that, our approach correctly verifies all programs while Chalice is unable to correctly verify 4 out of 11 programs with intricate interactions between thread fork/join and lock acquire/release. Currently, our framework only supports non-recursive locks. Lockbags could be incorporated into the framework to verify programs with recursive locks as well.

3. **Verification of Barrier Synchronization:** Besides locks, barriers are among the most commonly-used concurrency constructs. Verifying correct synchronization of barriers has been shown to be desirable at least in the context of SPMD programs [68, 76, 93, 134], as this can provide compilers and analysers with important phasing information for improving the precision of their analyses and optimizations such as reducing false sharing [68], may-happen-in-parallel analysis [93, 134], and data race detection [76]. However, in the context of fork/join programs, verification of barrier synchronization has almost been neglected. Static verification of barrier synchronization in fork/join programs is hard because programmers not only must keep track of (possibly dynamic) number of participating threads, but also have to ensure that all participants progress in correctly synchronized phases.

   In Chapter 5, we proposed an approach for verifying correct synchronization of *static* and *dynamic barriers* in fork/join programs. This is achieved by a new permission system, called *bounded permissions*, to enable reasoning about bounded resources (which are typically shared among only a bounded number (or a group) of concurrent threads). For verifying *static barriers*, *bounded permissions* and *phase numbers* are used to keep track of the number of participants and barrier phases respectively. For verifying *dynamic barriers*, we introduced *dynamic bounded permissions* to further keep track of the additions and/or removals of participants. To the best of our knowledge, our work is the

first to verify synchronization of both *static* and *dynamic barriers* in fork/join programs. This approach has proven to be sound, and a prototype of it has been applied to verify barrier synchronization of all but one of the simplified programs in SPLASH-2 benchmark suite within several seconds. The expressivity of our approach can be enhanced to allow for resource re-distribution to verify more complex barrier synchronization usage. Automatically discovery of barrier specification while still ensuring deadlock freedom in the presence of multiple barriers is another possible enhancement. Reasoning about programs with both barriers and locks is beyond the capabilities of current state-of-the-art systems, and is a challenging topic to pursue.

6.2 Future Directions

Our above works offer two main directions for future investigation: (1) more expressive verification of software barriers, and (2) specification and verification of concurrent programs under C/C++11 relaxed memory model.

More Expressive Verification of Software Barriers

A closely related future work is to increase the expressiveness of our approach for verification of barrier synchronization. Although the experiment in Section 5.3 showed that the approach could handle many intricate programs, there are still challenging open problems that we list below.

Functional Correctness vs. Barrier Synchronization

In our approach, threads are correctly synchronized on a barrier if they end up with the same (determinable) phase numbers. However, there are programs (such as `radiosity` program in SPLASH-2 suite [130]) where the phase numbers are tightly coupled with functional correctness, and are difficult or unable to be determined statically. A fragment of `radiosity` program is shown in Fig. 6-1. The bar-

rier `barrier` is used within the while loop which terminates only when the solution converges (by calling the procedure `init_ray_tasks` to check for convergence). The `init_ray_tasks` procedure only allows one thread (the first thread entering) to check for convergence and to update a global variable while other threads only read that variable. Such barrier phasing, therefore, is deeply correlated with functional correctness of the program (i.e. the convergence) which could not be captured by our existing approach. However, the approach could be extended to verify this type of programs by considering the use of existential phase numbers, and resource re-distribution for capturing the complex exchange of resource across barrier phases. This could potentially allow us to reason about resource distribution and deadlock freedom of programs with both barriers and locks.

```
/* ... perform ray-gathering till
the solution converges */
while( init_ray_tasks(...) ) {
  wait(barrier);
  process_tasks(...);
}
```

Figure 6-1: A Fragment of `radiosity`

Deadlock-free Multiple Barriers

Correct synchronization is a property weaker than deadlock freedom (or termination): it ensures deadlock freedom in case of a single barrier. When using multiple barriers, their synchronization patterns could potentially lead to deadlocks. For example, the program in Fig. 6-2 deadlocks because `thread1` blocks at barrier `b1` waiting for `thread2` to participate while `thread2` also blocks at barrier `b2` waiting for `thread1` to participate. We plan to extend our existing approach with barrier expressions to capture patterns of par-

```
b1 = new barrier(2);
b2 = new barrier(2);

//thread1        ||  //thread2
wait(b1);        ||    wait(b2);
wait(b2);        ||    wait(b1);
```

Figure 6-2: Deadlock due to Multiple Barriers

ticipating in multiple barriers. Together with the phase numbers, by proving that the barrier expressions of different participants are compatible, we could guarantee deadlock freedom. Patterns of participating in multiple barriers have been used in verification of SPMD programs with static barriers [5, 77, 134]. However, adapting them to verification of fork/join programs with dynamic barriers is non-trivial. This is not only because we need to address the unstructured nature of fork/join programs (in SPMD programs, threads execute the same code while, in fork/join programs, they execute different pieces of code), but also because we need to handle dynamic allocation/deallocation and addition/removals of participants in a modular way.

Inferring Barrier Specification

Specifications are important for verifying shared-memory programs. Nonetheless, manually writing specifications is laborious and tedious. Hence, it is often desirable to infer the specifications automatically [23, 24, 86, 92]. However, no existing work is able to infer barrier specifications. Our goal is to precisely infer the number of threads participating in a barrier and the current barrier phase. In the presence of multiple barriers, our approach will also discover whether the synchronization pattern could potentially deadlock. Furthermore, since concurrent threads progress in phases, threads could access different resources in different phases, i.e. the resources are differently distributed among threads in different phases. Previous work [63] deals with program verification in the presence of resource re-distribution, but it does not provide any inference support. We could identify the set of resources used by each thread in each barrier phase and automatically discover the global barrier specification that describes the resource re-distribution, even in the presence of varying number of synchronized threads, while still ensuring deadlock freedom.

Verification of C/C++11 Concurrent Programs

In this thesis, we assumed a sequentially consistent memory model. However, current multi-processor architectures such as ARM and Power use relaxed memory models.

To exploit these multi-processors efficiently, modern programming languages provide weak guarantees on the ordering of concurrent memory accesses, i.e. different threads observe memory operations in different orders. The relaxed memory model presented in the new C/C++ language standards (ISO/IEC 9899:2011 and ISO/IEC 14882:2011) is a case in point. The memory model allows for atomic memory operations – such as relaxed atomic, consume atomic, acquire atomic, release atomic, and sequentially consistent atomic – with weaker semantics than sequential consistency. However, using these atomic operations is very challenging because programmers have to reason about all these subtle semantics to ensure correctness. Hence, there is growing interest in specifying and verifying concurrent programs under the C/C++11 relaxed memory model [8, 104, 125]. Our ambition is to apply our experience in specification and verification of concurrent programs to verify the correctness of C/C++11 programs with the new relaxed memory model.

Specifically, while most current works focus on the release-acquire fragment of C/C++ [8, 124, 125], the release-consume fragment is still very challenging for verification. In the fragment, only dependent ownerships are allowed to be transferred via an atomic (release) write and an atomic (consume) read. Our idea is to record dependencies using dependency specifications and allow data dependencies to be explicitly captured and transferred (i.e. "dependencies as resource"). We then use the dependency specifications to enable the transfer of dependent ownerships, and consequently allow reasoning about programs written under the release-consume fragment of C/C++11.

References

[1] NetBSD Problem Report 42900. http://gnats.netbsd.org/42900.

[2] The Open Group Base Specifications Issue 7 IEEE Std 1003.1-2008. http://pubs.opengroup.org/onlinepubs/9699919799/basedefs/pthread.h.html.

[3] Cambridge University Study States Software Bugs Cost Economy $312 Billion Per Year, (retrieved on 26 Nov 2013). http://www.prweb.com/releases/2013/1/prweb10298185.htm.

[4] Chalice @ rise4fun from Microsoft, (retrieved on 29 July 2014). http://rise4fun.com/Chalice/.

[5] A. Aiken and D. Gay. Barrier Inference. In *ACM Symposium on Principles of Programming Languages*, pages 342–354, 1998.

[6] M. F. Atig, A. Bouajjani, M. Emmi, and A. Lal. Detecting Fair Non-termination in Multithreaded Programs. In *International Conference on Computer-Aided Verification*, pages 210–226, 2012.

[7] T. Ball, E. Bounimova, V. Levin, R. Kumar, and J. Lichtenberg. The Static Driver Verifier Research Platform. In *International Conference on Computer-Aided Verification*, pages 119–122, 2010.

[8] M. Batty, M. Doddsm, and A. Gotsman. Library Abstraction for C/C++ Concurrency. In *ACM Symposium on Principles of Programming Languages*, pages 235–248, 2013.

[9] J. Berdine, C. Calcagno, and P. W. O'Hearn. Smallfoot: Modular Automatic Assertion Checking with Separation Logic. In *Formal Methods for Components and Objects*, pages 115–137, 2005.

[10] J. Berdine, C. Calcagno, and P. W. O'Hearn. Symbolic Execution with Separation Logic. In *Asian Symposium on Programming Languages And Systems*, pages 52–68, 2005.

[11] K. Bierhoff and J. Aldrich. Modular Typestate Checking of Aliased Objects. In *ACM Conference on Object-Oriented Programming Systems, Languages, and Applications*, pages 301–320, 2007.

[12] S. Blom and M. Huisman. The VerCors Tool for Verification of Concurrent Programs. In *International Symposium on Formal Methods*, pages 127–131, 2014.

[13] S. Blom, J. Kiniry, and M. Huisman. How Do Developers Use APIs? A Case Study in Concurrency. In *IEEE International Conference on Engineering of Complex Computer Systems*, pages 212–221, 2013.

[14] H.-J. Boehm. Threads Cannot Be Implemented as a Library. In *ACM SIGPLAN Conf. on Programming Language Design and Implementation*, pages 261–268, 2005.

[15] R. Bornat, C. Calcagno, P. O'Hearn, and M. Parkinson. Permission Accounting in Separation Logic. In *ACM Symposium on Principles of Programming Languages*, pages 259–270, 2005.

[16] R. Bornat, C. Calcagno, and H. Yang. Variables as Resource in Separation Logic. *Electronic Notes in Theoretical Computer Science*, 155:247–276, 2006.

[17] C. Boyapati, R. Lee, and M. C. Rinard. Ownership Types for Safe Programming: Preventing Data Races and Deadlocks. In *ACM Conference on Object-*

Oriented Programming Systems, Languages, and Applications, pages 211–230, 2002.

[18] J. Boyland. Checking Interference with Fractional Permissions. In *International Static Analysis Symposium*, pages 55–72, 2003.

[19] S. Brookes. Variables as Resource for Shared-Memory Programs: Semantics and Soundness. *Electronic Notes in Theoretical Computer Science*, 158:123–150, 2006.

[20] S. Brookes. A Semantics for Concurrent Separation Logic. *Theoretical Computer Science*, 375:227–270, 2007.

[21] D. R. Butenhof. *Programming with POSIX Threads*. Addison-Wesley Professional, 1997.

[22] Y. Cai and W. K. Chan. MagicFuzzer: Scalable Deadlock Detection for Large-scale Applications. In *International Conference on Software Engineering*, pages 606–616, 2012.

[23] C. Calcagno, D. Distefano, P. W. O'Hearn, and H. Yang. Compositional Shape Analysis by Means of Bi-Abduction. *Journal of ACM*, 58(6):26, 2011.

[24] C. Calcagno, D. Distefano, and V. Vafeiadis. Bi-abductive Resource Invariant Synthesis. In *Asian Symposium on Programming Languages And Systems*, pages 259–274, 2009.

[25] C. Calcagno, M. Parkinson, and V. Vafeiadis. Modular Safety Checking for Fine-Grained Concurrency. In *International Static Analysis Symposium*, pages 233–248, 2007.

[26] W.N. Chin, C. David, H.H. Nguyen, and S. Qin. Multiple Pre/Post Specifications for Heap-Manipulating Methods. In *IEEE High Assurance Systems Engineering Symposium*, pages 357–364, 2007.

[27] E. G. Coffman, M. J. Elphick, and A. Shoshani. System Deadlocks. *ACM Computing Surveys*, 3(2):67–78, 1971.

[28] B. Cook, A. Podelski, and A. Rybalchenko. Terminator: Beyond Safety. In *International Conference on Computer-Aided Verification*, pages 415–418, 2006.

[29] B. Cook, A. Podelski, and A. Rybalchenko. Proving Program Termination. *Communications of the ACM*, 54(5):88–98, 2011.

[30] E. D. Demaine. C to Java: Converting Pointers into References. *Concurrency - Practice and Experience*, 10(11–13):851–861, 1998.

[31] T. Dinsdale-Young, L. Birkedal, P. Gardner, M. Parkinson, and H. Yang. Views: Compositional Reasoning for Concurrent programs. In *ACM Symposium on Principles of Programming Languages*, pages 287–300, 2013.

[32] T. Dinsdale-Young, M. Dodds, P. Gardner, M. Parkinson, and V. Vafeiadis. Concurrent Abstract Predicates. In *European Conference on Object-Oriented Programming*, pages 504–528, 2010.

[33] T. Dinsdale-Young, M. Dodds, P. Gardner, M. J. Parkinson, and V. Vafeiadis. Concurrent Abstract Predicates. In *European Conference on Object-Oriented Programming*, pages 504–528, 2010.

[34] R. Dockins, A. Hobor, and A. W. Appel. A Fresh Look at Separation Algebras and Share Accounting. In *Asian Symposium on Programming Languages And Systems*, pages 161–177, 2009.

[35] M. Dodds, X. Feng, M. Parkinson, and V. Vafeiadis. Deny-Guarantee Reasoning. In *European Symposium on Programming*, pages 363–377, 2009.

[36] M. Dodds, S. Jagannathan, and M. J. Parkinson. Modular Reasoning for Deterministic Parallelism. In *ACM Symposium on Principles of Programming Languages*, pages 259–270, 2011.

[37] A. Dolzmann, A. Seidl, and T. Sturm. Redlog User Manual - Edition 3.1, for REDLOG Version 3.06. 2006.

[38] X. Feng. Local Rely-Guarantee Reasoning. In *ACM Symposium on Principles of Programming Languages*, pages 315–327, 2009.

[39] X. Feng, R. Ferreira, and Z. Shao. On the Relationship between Concurrent Separation Logic and Assume-Guarantee Reasoning. In *European Symposium on Programming*, pages 173–188, 2007.

[40] X. Feng and Z. Shao. Modular Verification of Concurrent Assembly Code with Dynamic Thread Creation and Termination. In *ACM SIGPLAN International Conference on Functional Programming*, pages 254–267, 2005.

[41] P. Ferrara and P. Müller. Automatic Inference of Access Permissions. In *International on Verification, Model Checking, and Abstract Interpretation*, pages 202–218, 2012.

[42] C. Flanagan, K. R. M. Leino, M. Lillibridge, G. Nelson, J.B. Saxe, and R. Stata. Extended Static Checking for Java. In *ACM SIGPLAN Conf. on Programming Language Design and Implementation*, pages 234–245, 2002.

[43] A. Freeman. *Pro .NET 4 Parallel Programming in C#*. Apress, 2010.

[44] M. Frigo, C. E. Leiserson, and K. H. Randall. The Implementation of the Cilk-5 Multithreaded Language. In *ACM SIGPLAN Conf. on Programming Language Design and Implementation*, pages 212–223, 1998.

[45] M. Fu, Y. Zhang, and Y. Li. Formal Reasoning about Concurrent Assembly Code with Reentrant Locks. In *IEEE International Symposium on Theoretical Aspects of Software Engineering*, pages 233–240, 2009.

[46] P. Gerakios, N. Papaspyrou, and K. F. Sagonas. A Type and Effect System for Deadlock Avoidance in Low-level Languages. In *Workshop on Types in Languages Design and Implementation*, pages 15–28, 2011.

[47] C. Gherghina. Personal communication, 2013.

[48] C. Gherghina, C. David, S. Qin, and W.N. Chin. Structured Specifications for Better Verification of Heap-Manipulating Programs. In *International Symposium on Formal Methods*, pages 386–401, 2011.

[49] J. F. González. *Java 7 Concurrency Cookbook*. Packt Pub Limited, 2012.

[50] C. S. Gordon, M. D. Ernst, and D. Grossman. Static Lock Capabilities for Deadlock Freedom. In *Workshop on Types in Languages Design and Implementation*, pages 67–78, 2012.

[51] A. Gotsman, J. Berdine, B. Cook, N. Rinetzky, and M. Sagiv. Local Reasoning for Storable Locks and Threads. In *Asian Symposium on Programming Languages And Systems*, pages 19–37, 2007.

[52] C. Haack, M. Huisman, and C. Hurlin. Reasoning about Java's Reentrant Locks. In *Asian Symposium on Programming Languages And Systems*, pages 171–187, 2008.

[53] C. Haack, M. Huisman, and C. Hurlin. Permission-Based Separation Logic for Multithreaded Java Programs. *Newsletter of the NVTI*, 2011.

[54] C. Haack and C. Hurlin. Separation Logic Contracts for a Java-Like Language with Fork/Join. In *International Conference on Algebraic Methodology and Software Technology*, pages 199–215, 2008.

[55] S. Heule, K. R. M. Leino, P. Müller, and A. J. Summers. Fractional Permissions Without the Fractions. In *International Workshop on Formal Techniques for Java-like Programs*, 2011.

[56] J. M. D. Hill and D. B. Skillicorn. Practical Barrier Synchronisation. In *Euromicro International Conference on Parallel, Distributed, and Network-Based Processing*, pages 438–444, 1998.

[57] C. A. R. Hoare. An Axiomatic Basis for Computer Programming. *Communications of the ACM*, 12(10):576–580, 1969.

[58] C. A. R. Hoare. *Towards a Theory of Parallel Programming*, pages 61–71. Academic Press, 1972.

[59] C. A. R. Hoare. The Verifying Compiler: A Grand Challenge for Computing Research. *Journal of ACM*, 50:63–69, 2003.

[60] A. Hobor. *Oracle Semantics*. PhD thesis, Princeton University, 2008.

[61] A. Hobor, A. W. Appel, and F. Z. Nardelli. Oracle Semantics for Concurrent Separation Logic. In *European Symposium on Programming*, pages 353–367, 2008.

[62] A. Hobor and C. Gherghina. Barriers in Concurrent Separation Logic. In *European Symposium on Programming*, pages 276–296, 2011.

[63] A. Hobor and C. Gherghina. Barriers in Concurrent Separation Logic: Now With Tool Support! *Logical Methods in Computer Science*, 8(2), 2012.

[64] S. Ishtiaq and P. W. O'Hearn. BI as an Assertion Language for Mutable Data Structures. In *ACM Symposium on Principles of Programming Languages*, pages 14–26, 2001.

[65] B. Jacobs and F. Piessens. Expressive Modular Fine-grained Concurrency Specification. In *ACM Symposium on Principles of Programming Languages*, pages 271–282, 2011.

[66] B. Jacobs, J. Smans, P. Philippaerts, F. Vogels, W. Penninckx, and F. Piessens. VeriFast: a Powerful, Sound, Predictable, Fast Verifier for C and Java. In *International Conference on NASA Formal Methods*, pages 41–55, 2011.

[67] B. Jacobs, J. Smans, and F. Piessens. A Quick Tour of the VeriFast Program Verifier. In *Asian Symposium on Programming Languages And Systems*, pages 304–311, 2010.

[68] T. E. Jeremiassen and S. J. Eggers. Static Analysis of Barrier Synchronization in Explicitly Parallel Programs. In *International Conference on Parallel Architecture and Compilation Techniques*, pages 171–180, 1994.

[69] C. B. Jones. Specification and Design of (Parallel) Programs. In *IFIP Congress*, pages 321–332, 1983.

[70] C. B. Jones. *Wanted: a Compositional Approach to Concurrency*, pages 5–15. 2003.

[71] C. B. Jones. Balancing Expressiveness in Formal Approaches to Concurrency. Technical report, Newcastle University, 2013.

[72] C. B. Jones, P. W. O'Hearn, and J. Woodcock. Verified Software: A Grand Challenge. *IEEE Computer*, 39(4):93–95, 2006.

[73] P. Joshi, M. Naik, K. Sen, and D. Gay. An Effective Dynamic Analysis for Detecting Generalized Deadlocks. In *International Symposium on Foundations of Software Engineering*, pages 327–336, 2010.

[74] S. Joshi, R. K. Shyamasundar, and S. K. Aggarwal. A New Method of MHP Analysis for Languages with Dynamic Barriers. In *IEEE International Parallel and Distributed Processing Symposium Workshops*, pages 519–528, 2012.

[75] R. Kaivola, R. Ghughal, N. Narasimhan, A. Telfer, J. Whittemore, S. Pandav, A. Slobodov, C. Taylor, V. Frolov, E. Reeber, and A. Naik. Replacing Testing with Formal Verification in Intel CoreTM i7 Processor Execution Engine Validation. In *International Conference on Computer-Aided Verification*, pages 414–429, 2009.

[76] A. Kamil and K. A. Yelick. Concurrency Analysis for Parallel Programs with Textually Aligned Barriers. In *International Workshop on Languages and Compilers for Parallel Computing*, pages 185–199, 2005.

[77] A. Kamil and K. A. Yelick. Enforcing Textual Alignment of Collectives Using Dynamic Checks. In *International Workshop on Languages and Compilers for Parallel Computing*, pages 368–382, 2009.

[78] W. Kelly, V. Maslov, W. Pugh, E. Rosser, T. Shpeisman, and D. Wonnacott. *The Omega Library Version 1.1.0 Interface Guide*, November 1996.

[79] N. Klarlund and A. Møller. *MONA Version 1.4 User Manual*. BRICS, Department of Computer Science, Aarhus University, 2001.

[80] N. Kobayashi. Type-based Information Flow Analysis for the Pi-calculus. *Acta Informatica*, 42(4–5):291–347, 2005.

[81] N. Kobayashi. A New Type System for Deadlock-Free Processes. In *International Conference on Concurrency Theory*, pages 233–247, 2006.

[82] N. Kobayashi and D. Sangiorgi. A Hybrid Type System for Lock-Freedom of Mobile Processes. *ACM Transactions on Programming Languages and Systems*, 32(5), 2010.

[83] C. Laffra. A C++ to Java Translator. In *Advanced Java: Idioms, Pitfalls, Styles and Programming Tips*, chapter 4. Prentice Hall Computer Books, 1996.

[84] D.K. Le, W.N. Chin, S. Qin, and Y.M. Teo. Flow-Aware Resource Predicates for Concurrency Verification. Technical report, National University of Singapore, 2014.

[85] D.K. Le, W.N. Chin, and Y.M. Teo. Variable Permissions for Concurrency Verification. In *International Conference on Formal Engineering Methods*, pages 5–21, 2012.

[86] Q.L. Le, C. Gherghina, S. Qin, and W.N. Chin. Shape Analysis via Second-Order Bi-Abduction. In *International Conference on Computer-Aided Verification*.

[87] T.C. Le, C. Gherghina, A. Hobor, and W.N. Chin. A Resource-Based Logic for Termination and Non-termination Proofs. In *International Conference on Formal Engineering Methods*, pages 267–283, 2014.

[88] E. A. Lee. The Problem with Threads. *Computer*, 39:33–42, 2006.

[89] K. R. Leino, P. Müller, and J. Smans. Verification of Concurrent Programs with Chalice. In *Foundations of Security Analysis and Design V*, pages 195–222. 2009.

[90] K. R. M. Leino and P. Müller. A Basis for Verifying Multi-threaded Programs. In *European Symposium on Programming*, pages 378–393, 2009.

[91] K. R. M. Leino, P. Müller, and J. Smans. Deadlock-Free Channels and Locks. In *European Symposium on Programming*, pages 407–426, 2010.

[92] B. Li, I. Dillig, T. Dillig, K. L. McMillan, and M. Sagiv. Synthesis of Circular Compositional Program Proofs via Abduction. In *Tools and Algorithms for Construction and Analysis of Systems*, pages 370–384, 2013.

[93] Y. Lin. Static Nonconcurrency Analysis of OpenMP Programs. In *International Workshop on OpenMP*, 2005.

[94] S. Lu, S. Park, E. Seo, and Y. Zhou. Learning from Mistakes: a Comprehensive Study on Real World Concurrency Bug Characteristics. In *International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 329–339, 2008.

[95] Z. D. Luo, R. Das, and Y. Qi. Multicore SDK: A Practical and Efficient Deadlock Detector for Real-World Applications. In *International Conference on Software Testing, Verification and Validation*, pages 309–318, 2011.

[96] A. Malkis and A. Banerjee. Verification of Software Barriers. In *ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 313–314, 2012.

[97] F. Martins, V. T. Vasconcelos, and T. Cogumbreiro. Types for X10 Clocks. In *Workshop on Programming Language Approaches to Concurrency and Communication-centric Software*, pages 111–129, 2010.

[98] T. Mattson, B. Sanders, and B. Massingill. *Patterns for Parallel Programming*. Addison-Wesley Professional, first edition, 2004.

[99] L. Mendonça de Moura and N. Bjørner. Z3: an efficient SMT solver. In *Tools and Algorithms for Construction and Analysis of Systems*, pages 337–340, 2008.

[100] P. Müller. Personal communication, 2013.

[101] K. Naden, R. Bocchino, J. Aldrich, and K. Bierhoff. A Type System for Borrowing Permissions. In *ACM Symposium on Principles of Programming Languages*, pages 557–570, 2012.

[102] M. Naik, C.-S. Park, K. Sen, and D. Gay. Effective Static Deadlock Detection. In *International Conference on Software Engineering*, pages 386–396, 2009.

[103] H.H. Nguyen, C. David, S. Qin, and W.N. Chin. Automated Verification of Shape and Size Properties via Separation Logic. In *International on Verification, Model Checking, and Abstract Interpretation*, pages 251–266, 2007.

[104] B. Norris and B. Demsky. CDSChecker: Checking Concurrent Data Structures Written with C/C++ Atomics. In *ACM Conference on Object-Oriented Programming Systems, Languages, and Applications*, pages 131–150, 2013.

[105] P. W. O'Hearn. Resources, Concurrency and Local Reasoning. In *International Conference on Concurrency Theory*, pages 49–67, 2004.

[106] P. W. O'Hearn. Resources, Concurrency, and Local Reasoning. *Theoretical Computer Science*, 375:271–307, 2007.

[107] S. Okur and D. Dig. How Do Developers Use Parallel Libraries? In *International Symposium on Foundations of Software Engineering*, page 54, 2012.

[108] J. Ouaknine, H. Palikareva, A. W. Roscoe, and J. Worrell. Static Livelock Analysis in CSP. In *International Conference on Concurrency Theory*, pages 389–403, 2011.

[109] S. Owicki. *Axiomatic Proof Techniques for Parallel Programs*. PhD thesis, 1975.

[110] S. Owicki and D. Gries. An Axiomatic Proof Technique for Parallel Programs I. *Acta Informatica*, 6:319–340, 1976.

[111] S. Owicki and D. Gries. Verifying Properties of Parallel Programs: an Axiomatic Approach. *Communications of the ACM*, 19:279–285, 1976.

[112] M. Parkinson, R. Bornat, and C. Calcagno. Variables as Resource in Hoare Logics. In *IEEE Logic In Computer Science*, pages 137–146, 2006.

[113] D. Patterson. The Trouble with Multi-core. *IEEE Spectrum*, 47:28–32, 2010.

[114] U. S. Reddy and J. C. Reynolds. Syntactic Control of Interference for Separation Logic. In *ACM Symposium on Principles of Programming Languages*, pages 323–336, 2012.

[115] J. Reynolds. Separation Logic: A Logic for Shared Mutable Data Structures. In *IEEE Logic In Computer Science*, pages 55–74, 2002.

[116] L.G. Roberts. Beyond Moore's Law: Internet Growth Trends. *IEEE Computer*, 33:117–119, 2000.

[117] V. A. Saraswat and R. Jagadeesan. Concurrent Clustered Programming. In *International Conference on Concurrency Theory*, pages 353–367, 2005.

[118] J. Shirako, D. M. Peixotto, V. Sarkar, and W. N. Scherer III. Phasers: a Unified Deadlock-free Construct for Collective and Point-to-point Synchronization. In *International Conference on Supercomputing*, pages 277–288, 2008.

[119] J. Smans, B. Jacobs, and F. Piessens. Implicit Dynamic Frames. *ACM Transactions on Programming Languages and Systems*, 34(1):2, 2012.

[120] K. Suenaga. Type-Based Deadlock-Freedom Verification for Non-Block-Structured Lock Primitives and Mutable References. In *Asian Symposium on Programming Languages And Systems*, pages 155–170, 2008.

[121] K. Svendsen, L. Birkedal, and M. J. Parkinson. Modular Reasoning about Separation of Concurrent Data Structures. In *ESOP*, pages 169–188, 2013.

[122] N. Tillmann and J. De Halleux. Pex: White Box Test Generation for .NET. In *International Conference on Tests and Proofs*, pages 134–153, 2008.

[123] T. Tuerk. *A Separation Logic Framework for HOL*. PhD thesis, University of Cambridge, 2011.

[124] A. Turon, V. Vafeiadis, and D. Dreyer. GPS: Navigating Weak Memory with Ghosts, Protocols, and Separation. In *ACM Conference on Object-Oriented Programming Systems, Languages, and Applications*, pages 691–707, 2014.

[125] V. Vafeiadis and C. Narayan. Relaxed Separation Logic: A Program Logic for C11 Concurrency. In *ACM Conference on Object-Oriented Programming Systems, Languages, and Applications*, pages 867–884, 2013.

[126] V. Vafeiadis and M. Parkinson. A Marriage of Rely/Guarantee and Separation Logic. In *International Conference on Concurrency Theory*, pages 256–271, 2007.

[127] J. Villard, É Lozes, and C. Calcagno. Proving Copyless Message Passing. In *Asian Symposium on Programming Languages And Systems*, pages 194–209, 2009.

[128] A. Williams, W. Thies, and M. D. Ernst. Static Deadlock Detection for Java Libraries. In *European Conference on Object-Oriented Programming*, pages 602–629, 2005.

[129] J. M. Wing. FAQ on Pi-calculus. In *Microsoft Internal Memo*, 2002.

[130] S. C. Woo, M. Ohara, E. Torrie, J. P. Singh, and A. Gupta. The SPLASH-2 Programs: Characterization and Methodological Considerations. In *International Symposium on Computer Architecture*, 1995.

[131] W. Xiong, S. Park, J. Zhang, Y. Zhou, and Z. Ma. Ad Hoc Synchronization Considered Harmful. In *USENIX Conference on Operating Systems Design and Implementation*, pages 1–8, 2010.

[132] H. Yang and P. W. O'Hearn. A Semantic Basis for Local Reasoning. In *Foundations of Software Science and Computation Structures*, pages 402–416, 2002.

[133] Y. Zhang and E. Duesterwald. Barrier Matching for Programs with Textually Unaligned Barriers. In *ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 194–204, 2007.

[134] Y. Zhang, E. Duesterwald, and G. R. Gao. Concurrency Analysis for Shared Memory Programs with Textually Unaligned Barriers. In *International Workshop on Languages and Compilers for Parallel Computing*, 2007.

Appendix A

Variable Permissions

Overview. Access permissions have recently attracted much attention for reasoning about heap-manipulating concurrent programs [15, 18, 41, 51, 55, 61, 65]. Each heap location is associated with a permission and a thread can access a location if and only if it has the access permission for that location. Permissions can be flexibly transferred among callers and callees of the same threads or among different threads. A thread needs a certain fraction of a permission to read a location, but it has to own the full permission in order to perform a write. This guarantees data-race freedom in the presence of concurrent accesses to heap locations.

Program variables[1] can also be shared among threads and are prone to data races. Therefore, one may adopt a similar scheme, designed for heap locations, to reason about variables. "Variables as resource" [16, 112] indeed uses such a permission scheme for variables. Each variable x is augmented with a predicate $Own(x, \pi)$ where π denotes the permission to access x. The permission domain is either (0,1] for fractional permissions [18] or $[0,\infty)$ for counting permissions [15]. This allows variables to be treated in the same way as heap locations. However, this permission scheme is more complex and places higher burden on programmers to figure out the fraction to be associated to a variable and how to perform permission accounting properly [15]. To the best of our knowledge, we are not aware of any existing verifiers that have

[1] We mean either global variables or local variables; as distinct from heap locations.

fully implemented the idea. SMALLFOOT [9] uses side-conditions to outlaw conflicting accesses to variables. This, however, requires subtle, global, and hard-to-check conditions that a compiler should ensure [16, 114]. Similarly, CHALICE [89, 90], a program verifier developed for concurrency verification, does not support permissions for variables in method bodies. Even VERIFAST [65, 66], the state-of-the-art verifier, still does not naturally support concurrency reasoning using variables, though it has support for variables by simulating them as heap locations. Consequently, existing verification systems narrow the programmers' choice to heap locations instead of variables for shared accesses by concurrent threads at the expense of losing the expressivity and simplicity that variables provide.

In this appendix, we argue that variables with their own characteristics could be treated in a much simpler way than heap locations. Firstly, each variable is distinct; therefore, aliasing issue required for heap locations can be ignored for variables in most cases. Secondly, if several threads need to concurrently *read* a variable, the main thread holding the full permission of the variable can just give each child thread a copy of the variable through pass-by-value mechanism. If concurrent threads require *write* access to the same variable, this shared variable can be protected by a mutex lock whose invariant holds the full permission of the variable. Lastly, if only one thread requires a write access to a given variable, we can simply pass the full permission of the variable into the thread (through pass-by-reference) whose permission is only returned when the child thread joins the main thread. This scheme allows concurrent but race-free accesses to variables.

Nonetheless, there are two scenarios where the above scheme is inadequate. The first scenario occurs in languages such as C/C++ when some variables can be aliased through the use of the address-of operator &. The second scenario occurs when concurrent threads require phased accesses to shared variables, e.g. concurrent threads safely read prior to writing to shared variables. In both scenarios, we propose to automatically translate the affected variables into pseudo-heap locations where a more complex heap permission scheme is utilized.

Because of the above observations, we propose to simply assign a permission of either full or zero to a variable. We can utilize heap (or pseudo-heap) locations to complement our concurrent programming model, *where necessary*, and also readily use variables, *where sufficient*. The net result is a rich but still verifiable programming paradigm for concurrent threads. We shall show that our treatment of variable permissions is sound and expressive to capture programming models such as POSIX threads [21] and Cilk [44]. To relieve programmers from annotation efforts, we shall demonstrate an algorithm to automatically infer variable permissions by only looking at procedure specifications. We shall also provide a translation scheme to handle the variable aliasing (that can also be used for variables requiring phased accesses) and thus complement our treatment of variable permissions.

Contributions. In this appendix, we make the following contributions:

- A simpler treatment of variable permissions to ensure safe concurrent accesses to program variables, as distinct from heap locations (Section A.1 and A.2.2). We also demonstrate the applicability of our scheme to popular programming models such as POSIX threads and Cilk (Section A.2.5).

- An algorithm to automatically infer variable permissions from procedure specifications. This helps to reduce program annotations (Section A.2.3).

- A translation scheme to eliminate variable aliasing for the purpose of program verification (Section A.2.4). We present how to translate programs with pointers and address-of operator (&) into our core language (Section A.2.1).

A.1 Motivating Example

This section illustrates our treatment of variable permissions to reason about concurrent programs. Figure A-1 shows an example illustrating the widely-used task-decomposition pattern in concurrent programming. The `main` procedure invokes the `creator` procedure to create a concurrent task and later performs a join to collect its

result. In this example, the `main` procedure creates two local variables `x` and `y`, and passes them to the `creator`. The `creator` forks a child thread that increases `x` by 1, and itself increases `y` by 2. The identifier `tid` of the child thread is returned to the `main` procedure which will later perform a `join`.

This example shows a fairly complicated inter-procedural passing of variables between the main thread and the child thread. It poses two challenges: (i) how to describe the fact that any accesses to `x` after forking the child thread and before joining it are unsafe, and (ii) how to propagate this fact across procedure boundaries. These issues can be resolved soundly and modularly by our proposed variable permissions.

Modular reasoning is achieved by augmenting the program's specifications with variable permissions: @*full*[...] and @*value*[...]. In pre-conditions (specified after `requires` keyword), @*full*[v^*] and @*value*[v^*] denote lists of pass-by-reference and pass-by-value parameters. If a variable is passed by reference, the caller transfers the full permission of that variable to the callee. If a variable is passed by value, only a copy of that variable is passed to the callee and the caller still has the full permission of that variable. In post-conditions (after

```
void  inc(ref int i,int j)
  requires  @full[i] ∧ @value[j]
  ensures   @full[i] ∧ i'=i+j;
 {   i=i+j; }

thrd  creator(ref int x,ref int y)
requires  @full[x, y]
ensures   res ↦ thrd⟨@full[x]∧x'=x+1⟩
          ∧ @full[y] ∧ y'=y+2
 {   thrd  tid = fork(inc,x,1);
     inc(y,2);
     return tid;
 }

void  main()
  requires emp  ensures  emp;
 {   thrd  id;
     int  x=0,y=0;
     id = creator(x,y);
     ...
     join(id);
     assert (x'+y'=3);
 }
```

Figure A-1: A Motivating Example

`ensures` keyword), @*full*[v^*] specifies the transfer of full permissions from the callee back to the caller via pass-by-reference parameters. Note that callers and callees can be in a single thread in case of normal procedure calls or in different threads in case of asynchronous calls via fork/join.

In this example, the `main` procedure transfers the full permissions of `x` and `y` to the `creator` (specified in its precondition as @*full*[x, y]). When forking a new child thread executing the `inc` procedure, the main thread transfers the full permission of `x` to the child thread (using pass-by-reference mechanism). This effect can be seen in the post-condition of the `creator` where we have two concurrent threads: after giving up the full permission of `x`, the main thread retains the full permission of `y` (@*full*[y]) while the child thread (represented by the thread node $res \mapsto \mathsf{thrd}\langle @full[x] \wedge x'{=}x{+}1\rangle$) holds the full permission of `x` (@*full*[x]). Thus, prior to invoking a `join` to merge back the child thread, the main thread has zero permission of `x` and is not allowed to access it (neither read nor write). This ensures data-race freedom since only one thread at a time can have the full permission of `x`.

In the specification, we use a thread node (i.e. "threads as resource") to capture the child thread and the keyword *res* to represent the return value of a procedure call (in case of `creator`, the return value is the thread identifier `tid` of the child thread). Additionally, we use *primed notation* to handle updates to variables. The primed version x' of a variable x denotes its latest value; the unprimed version x denotes its initial value (i.e. its value at the beginning of the procedure). Note that a variable x and its primed version x' can be related but are two different logical variables.

One may think that this treatment of variable permissions can be easily captured through parameter passing, e.g. for each reference parameter `v`, just add an @*full*[v] in the main thread of both pre- and post-conditions. However, this simple assumption may not hold in the context of concurrency. The key question is which thread holds full permission of a given variable. The full permission can belong to the main thread in the pre-condition but later it is transferred to a child thread in the post-condition and vice versa. For example, in the `creator`, the main thread has @*full*[x] in the

pre-condition but this permission is later transferred to the child thread in the post-condition. In summary, the goal of our scheme is to succinctly manage the transfer of variable permissions among threads in a sound and modular manner.

A.2 Proposed Approach

A.2.1 Programming and Specification Languages

proc_decl	::=	*ret_type pn*(*param**) *spec** { *s* }	Procedure declaration
param	::=	*type v* | **ref** *type v*	Parameter

Figure A-2: Programming Language with Pass-by-Reference

We enhance our core programming language in Section 3.2.1 with the abilities to pass parameters both by reference (**ref**) and by value (Fig. A-2).

Separation formula	Φ	::=	$\bigvee(\exists v^* \cdot \kappa \wedge \nu \wedge \pi)$
Heap formula	κ	::=	$\mathtt{emp} \mid \iota \mid \kappa_1 * \kappa_2$
Atomic heap formula	ι	::=	$v \overset{\varepsilon}{\mapsto} C(v^*) \mid v \mapsto \mathsf{thrd}\langle\Phi\rangle$
Vperm formula	ν	::=	$@zero[v^*] \mid @full[v^*] \mid @value[v^*]$ $\mid \nu_1 \wedge \nu_2 \mid \nu_1 \vee \nu_2$
Pure formula	π	::=	...

Fractional permission variable $\varepsilon \in (0,1]$ $\quad C \in$ Data names $\quad v \in$ Variables

Figure A-3: Specification Language with Variable Permissions

Figure A-3 shows our rich specification language for concurrent programs manipulating variables and heap locations. For variables, we use variable permissions. For heap locations, we support fractional permissions ε [18]. Most of the language features are similar to those explained in previous chapters. The newly added com-

ponent is the vperm formula ν describing permissions of variables. We will elaborate more about ν in Section A.2.2.

A.2.2 Verification Rules

Formalism.

In order to ensure safe concurrent accesses to variables, we use two key annotations for variable permissions:

- $@full[v^*]$ specifies the full permissions of a list of variables v^*. In pre-conditions, it means that v^* is a list of pass-by-reference parameters. In post-conditions, it captures the return of permissions to callers.
- $@value[v^*]$ only appears in pre-conditions to specify a list of pass-by-value parameters v^*.

$$@full[S] \ \wedge \ v \not\in S \vdash @full[v] \quad \rightsquigarrow \quad fail \qquad \textbf{FAIL-1}$$

$$@full[S] \ \wedge \ v \not\in S \vdash @value[v] \quad \rightsquigarrow \quad fail \qquad \textbf{FAIL-2}$$

$$\frac{v \in S}{@full[S] \vdash @full[v] \quad \rightsquigarrow \quad @full[S - \{v\}]} \qquad \textbf{P-REF}$$

$$\frac{v \in S}{@full[S] \vdash @value[v] \quad \rightsquigarrow \quad @full[S]} \qquad \textbf{P-VAL}$$

$$@full[S_1] \wedge @full[S_2] \quad \rightsquigarrow \quad @full[S_1 \cup S_2] \qquad \textbf{NORM-1}$$

$$@full[S_1] \vee @full[S_2] \quad \rightsquigarrow \quad @full[S_1 \cap S_2] \qquad \textbf{NORM-2}$$

$$@full[S_1] \wedge @value[S_2] \quad \rightsquigarrow \quad @full[S_1 \cup S_2] \qquad \textbf{BEGIN}$$

Figure A-4: Entailment Rules on Variable Permissions

Variable permissions can be transferred among callers and callees of the same thread, and among distinct threads. The entailment rules for variable permissions are shown in Figure A-4. A main thread (or a caller) that does not have full permission of a variable cannot pass that full permission to another thread (or a callee)

either by reference or by value (**FAIL–1** and **FAIL–2**). After passing a variable by reference, a main thread (or a caller) loses the full permission of that variable (**P–REF**). However, for a pass-by-value variable, it will still retain the full permission (**P–VAL**). The normalization rules **NORM–1** and **NORM–2** soundly approximate sets of full permissions. At the beginning of a procedure, a main thread has full permissions of its pass-by-reference and pass-by-value parameters (**BEGIN**). Since $@value[v^*]$ only appears in pre-conditions, the rule **BEGIN** indicates that a callee will have the full permissions of both pass-by-reference arguments and copies of pass-by-value arguments. The rules presented are simple, and this is precisely how we would like the readers to feel. Simplicity has its virtue and we hope that this would encourage safer concurrent programs to be written.

In our implementation, we also support $@zero[\cdots]$ as a dual to $@full[\cdots]$ annotation. The former denotes a set of variables that possibly have zero permission. This is useful for more concise representation since only a small fraction of variables typically lose their permissions temporarily.

$$\frac{\Delta \vdash @full[v]}{\{\Delta\} \ldots = \ldots\ v\ \ldots \{\Delta\}} \qquad \textbf{VAR–READ}$$

$$\frac{\Delta \vdash @full[v]}{\{\Delta\}\ v = \ldots \{\Delta\}} \qquad \textbf{VAR–WRITE}$$

Figure A-5: Forward Verification Rules for Manipulating Variables

Most verification rules in the presence of variable permissions remain unchanged compared with those described in Fig. 3-4 of Chapter 3. Note that the transfer of variable permissions across procedures and threads is performed during fork, join, and procedure calls. The rules in Fig. A-5 additionally require that a thread needs to hold a full permission to manipulate (either read or write) a program variable.

Soundness Proof: We sketch how our variable permission scheme (Section A.2.2) ensures safe concurrency (data-race freedom). We prove that our scheme maintains the invariant that the full permission of each variable belongs to at most one thread at any time.

Definition 8 (Data-race Freedom)**.** *A program is data-race free if there do not exist a state* $\Delta = \Delta_{t1} * \Delta_{t2}$ *and a variable* `x` *such that* $\Delta_{t1} \vdash @full[x]$ *and* $\Delta_{t2} \vdash @full[x]$.

Definition 9 (Permission Invariant)**.** *For every variable* `x`*, its full permission belongs to at most one thread at any time.*

Theorem 4 (Non-duplicable Permissions)**.** *For every variable* `x`*, its full permission cannot be duplicated.*

Proof. By induction on entailment rules in Figure A-4. □

Lemma 6 (Soundness of Variable Permission Scheme)**.** *Given a program with a set of procedures* P^i *and their corresponding pre/post-conditions* (Φ^i_{pr}/Φ^i_{po}) *enhanced with variable permissions, if our verification system derives a proof for every procedure* P^i*, i.e.* $\{\Phi^i_{pr}\}\ P^i\ \{\Phi^i_{po}\}$ *is valid, then the program is free from data races.*

Proof. It follows from Theorem 4. Using our variable permission scheme, the full permission of each variable in a program belongs to at most one thread at any time; therefore, the program is data-race-free.

We prove the soundness of our variable permission scheme by contradiction.
Hypothesis: There are data races, i.e. there are two threads that have full permission of the same variable `x` at the same time.

The two threads can be: a main thread and a child thread (Case 1), or both child threads (Case 2).
Case 1: A main thread and a child thread have the full permission of the same variable.

Case 1.1: The child thread obtains the full permission after being forked by the main thread. Therefore, the variable `x` has to be passed by reference to the

child thread (**P–REF** rule in Figure A-4). Afterwards, the main thread loses the full permission because the permission is non-duplicable. This contradicts to the hypothesis.

Case 1.2: The child thread obtains the full permission from the lock invariant after acquiring a mutex lock. In our scheme, if a variable is protected by a mutex lock, the lock's invariant holds the full permission of the variable. Therefore, if the main thread has the full permission of `x`, it also has to acquire the full permission from the lock invariant. This leads to contradiction because two threads are not allowed to successfully acquire a lock at the same time.

Case 2: Two child threads have the full permission of the same variable.

Case 2.1: Child threads obtains the full permissions after being forked by another main thread. This is impossible because once the firstly-forked thread have acquired the full permission of the variable, the full permission is no longer available to be transferred to the second thread.

Case 2.2: Child threads obtain the full permission of `x` from the lock invariant after acquiring a mutex lock. This is impossible because two threads are not allowed to successfully acquire a lock at the same time.

□

A.2.3 Inferring Variable Permissions

In this section, we investigate inference for variable permissions. Approaches in permission inference for variables [114] and heap locations [41, 55] require entire program code and/or its specifications for their global analysis. The simplicity of our variable permission scheme offers opportunities for automatically and modularly inferring variable permissions by only looking at procedure specifications.

Our inference is based on following key observations. Firstly, local variables of a procedure cannot escape from their lexical scope; therefore, they are not allowed to appear in post-conditions. Secondly, scopes of pass-by-value parameters are only within their procedures; therefore, @*value*[...] only exists in pre-conditions and up-

dates to these parameters need not be specified in post-conditions. Thirdly, for each procedure with its *R-complete* pre/post-conditions, updates to its reference parameters must be specified in its post-condition via *primed notations*. Lastly, because child threads carry the post-conditions of their corresponding forked procedures, their states include information about updates to variables that were passed by reference to their forked procedures.

Definition 10 (Primed Notations and R-complete Specifications)**.** *Primed notations represent the latest values of program variables; unprimed notations denote either logical variables or initial values of program variables. A procedure specification is R-complete if all updates to its pass-by-reference parameters are specified in the pre/post conditions using primed notations.*

Algorithm 1 Inferring variable permissions from procedure specifications

Input: Φ_{pr}, Φ_{po}: pre/post-conditions of a procedure without variable permissions
Input: V_{ref}, V_{val} : sets of pass-by-reference and pass-by-value parameters
Output: Pre/post-conditions with inferred variable permissions

1: V_{post}:=V_{ref}
2: /*Infer @*full*[...] annotations for post-condition*/
3: **for** each thread Δ in Φ_{po} **do**
4: /*Set of free variables that are updated in Δ using primed notations*/
5: V_m:=$\{v : v \in FV(\Delta) \wedge isPrimed(v)\}$
6: **if** $(V_m - V_{post}) \neq \phi$ **then** Error
7: **else**
8: Δ:=$\Delta \wedge$ @*full*$[V_m]$
9: V_{post}:=$V_{post} - V_m$
10: **end if**
11: **end for**
12: /*excluding reference parameters not updated in post-condition*/
13: V_{pre}:=$V_{ref} - V_{post}$
14: /*Infer @*full*[...] annotations for pre-condition's child threads*/
15: /*in the same way as with those in post-condition but replace V_{post} by V_{pre}*/
16: **for** each child thread Δ_t in Φ_{pr} **do**
17: ...
18: **end for**
19: For the main thread Δ in Φ_{pr}: $\Delta := \Delta \wedge$ @*full*$[V_{pre}] \wedge$ @*value*$[V_{val}]$
20: **return** Φ_{pr},Φ_{po}

We present our inference in Algorithm 1. For each procedure, the algorithm starts inference for the post-condition first. For each thread in the post-condition (either

Table A.1: Inferring Variable Permissions for Procedure `creator` in Figure A-1

Input		Intermediate values	Inferred
V_{ref}:={x, y}, V_{val}:={}			
Φ_{po}:=	y'=y+2	V_{post}:={x, y}, V_m:={y}	@*full*[y]
	$res \mapsto \mathsf{thrd}\langle x'=x+1 \rangle$	V_{post}:={x}, V_m:={x}	@*full*[x]
Φ_{pr}:=	`true`	V_{pre}:={x, y}	@*full*[x, y]

main thread or child thread), the full permissions are inferred by computing the pass-by-reference parameters that are updated in each thread's specification via primed notations. The `if` statement in line 6 detects an error if there are some primed variables that (1) are not reference parameters or (2) belonged to other threads in the previous iterations. The subtraction in line 9 removes from the set of reference parameters V_{post} those variables whose inferred full permissions already belonged to the current thread. This ensures that only one thread in the specification holds the full permission of a variable. Because child threads in the pre-condition carry the post-conditions of their corresponding forked procedures, we infer variable permissions for these child threads in the same way as with those in the post-condition. Note that the main thread is the currently active execution thread; therefore, its state in the pre-condition does not include primed variables. The main thread of the pre-condition holds full permissions of variables whose are updated (specified in the post-condition) and do not belong to any child threads. The subtraction in line 13 is necessary because there are certain variables that are passed by reference but their full permissions do not belong to any threads (see Section A.2.5 for more discussions). Finally, permission annotation @*value*[...] of pass-by-value parameters is added into the main thread of the pre-condition. In the algorithm, the main thread indicates the main executing threads while the child thread(s) are represented by thread nodes. For illustration, we present a running example in Table A.1.

Soundness Proof: We give the soundness sketch of our inference algorithm. We first prove that the inferred full permission of each variable belongs to at most one thread in a procedure's *R-complete* specification. Then, we prove that with the inferred

variable permissions, the procedure is free from data races.

Theorem 5 (Precise Inference)**.** *The inferred full permission of each parameter belongs to at most one thread in a procedure's R-complete specification.*

Proof. We prove by contradiction.

HYPOTHESIS: There exists a parameter `x` whose inferred full permission belongs to more than one thread in the procedure's pre/post-conditions.

CASE 1: The parameter `x` is passed by reference.

CASE 1.1: The parameter `x` is not protected by any mutex lock. Because the specification is *R-complete*, by Definition 10, updates to `x` are specified in the specification using *primed notation*.

CASE 1.1.1: Inferring the permission of `x` in the post-condition.

Without lost of generosity, assuming that the full permission of `x` belongs to two threads in the post-condition, i.e. $@full[x]$ is in the state of the two threads. Because the algorithm iterates over each thread in a sequential manner (line 3-11), assuming that the two threads are visited in iterations `i` and `j` respectively ($i<j$). Let V^i_{post} and V^i_m denote the value of V_{post} and V_m after `i`-th iteration. Therefore, we have $x \in V^i_m$ and $x \in V^j_m$ with $i<j$. As a consequence, we have $x \in V^{j-1}_{post}$ (because $V_m - V_{post} = \phi$). By induction on the value of `j`, we have $x \in V^i_{post}$. This is impossible because of the subtraction in line 9.

CASE 1.1.2: Inferring the permission of `x` in the pre-condition.

Similar to CASE 1.1.1 but replace V_{post} by V_{pre}.

CASE 1.2: The parameter `x` is protected by some mutex lock.

In our scheme, if a variable `x` is protected by a mutex lock, only the lock's invariant holds the full permissions of `x`. This contradicts to the hypothesis. Note that in this case, updates to variable `x` are captured in the lock invariant. Therefore, neither threads hold the full permission of `x`. Formally, for every iteration `i`, $x \notin V^i_m$.

CASE 2: The parameter `x` is passed by value.

Because the main thread is the main execution thread, the permission $@value[...]$ of pass-by-value parameters is added to the main thread of the precondition (line 19).

This contradicts to the hypothesis. Note that @*value*[...] does not exist in the post-condition and updates to pass-by-value parameters are not allowed to be specified in the post-condition (to prevent them from escaping from their lexical scope). □

Theorem 6 (Soundness)**.** *With the inferred variable permissions, the procedure is free from data races.*

Proof. This follows from the preciseness of our inference algorithm (Theorem 5) and the soundness of our underlying permission scheme (Lemma 6). □

Corollary 7 (Soundness of Inference and Verification)**.** *Given a procedure* `P` *with its R-complete pre/post-conditions (Φ_{pr}/Φ_{po}) without variable permissions, and our inference algorithm results in new pre/post-conditions (Φ'_{pr}/Φ'_{po}) with inferred variable permissions, if our verification system derives a proof, i.e.* $\{\Phi'_{pr}\}$ `P` $\{\Phi'_{po}\}$ *is valid, then the procedure* `P` *is free from data races.*

Proof. It follows from Theorem 6. □

A.2.4 Eliminating Variable Aliasing

In this section, we investigate the problem of variable aliasing. Aliasing occurs when a data location can be accessed through different symbolic names (i.e. variable names). For example in C/C++, variables can be aliased by the use of address-of operator (&). This poses challenges to program verification in general and concurrency verification in particular. Figure A-6(a) shows a problematic example where `p` and `x` are aliased due to the assignment p=&x. After passing `x` by reference to a child thread, although the main thread does not have permission to access `x`, it can still access the value of `x` via its alias `*p` and therefore incurs possible data races. Our goal is to ensure safe concurrent accesses to variables even in the presence of aliasing, e.g. to outlaw racy accesses to the value of `x`.

We propose a translation scheme to eliminating variable aliasing by unifying pointers to program variables and pointers to heap locations. The translation is automatic

```
void  inc(ref int i,int j)
requires  @full[i] ∧ @value[j]
ensures  @full[i] ∧ i'=i+j;
 {   i=i+j; }

void  main()
 requires  emp  ensures  emp;
{
  int x=0;
  int* p=&x;
  thrd id = fork(inc,x,1);
  /*accesses to *p are racy*/
   ...
  join(id);

}
```

(a) Original Program

```
void  inc(int_ptr i,int j)
requires  i ↦ int_ptr(old_i) ∧ @value[i,j]
ensures  i ↦ int_ptr(new_i) ∧ new_i=old_i + j;
 {   i.val=i.val+j; }

void  main()
 requires  emp  ensures  emp;
{
  int_ptr x =  new int_ptr(0);
  int_ptr p=x;
  thrd id = fork(inc,x,1);
  /*accesses to p.val or x.val are illegal*/
   ...
  join(id);
  delete(x);
}
```

(b) Translated Program

Figure A-6: An Example of Eliminating Variable Aliasing

and transparent to programmers. We refer to each variable (or parameter) whose `&x` appears in the program as an *addressable* variable. Intuitively, for each *addressable* variable, our translation scheme transforms it into a pointer to a pseudo-heap location by the following substitution ρ=[`int` $\mapsto$ `int_ptr`,`&x` $\mapsto$ `x`, `x` $\mapsto$ `x.val`]. Our approach covers values of any type (including primitive and data types). For each type `t`, there is a corresponding type `t_ptr` to represent the type of pointers to pseudo-heap locations holding a value of type `t`. The value located at a pseudo-heap location is accessed via its `val` field (e.g. `x.val`).

Definition 11 (Pseudo-heap Locations)**.** *Pseudo-heap locations are heap-allocated locations used for verification purpose only. Each pseudo-heap location represents a transformed program variable and captures the original value of the variable in its* `val` *field.*

Our scheme also translates program pointers into pointers to heap-allocated locations by the following substitution ρ=[int* $\mapsto$ int_ptr, *p $\mapsto$ p.val]. For pointers that point to another pointer, our translation is also applicable, e.g. int^{**} is translated into int_ptr_ptr. The translation scheme ensures that the semantics of the translated program is equivalent to that of the original program. By transforming addressable variables into pseudo-heap locations, reasoning about aliased variables has been translated to reasoning about aliased heap locations which is easier to handle (e.g. using separation logic [115]).

Our translation rules are presented in Figure A-7. As a part of the translation, we first transform the program to ensure that variables are of distinct names. Afterwards, we analyze the program to identify a set V of addressable variables that are passed by reference. Our translation starts with such a set of variables and gradually adds more addressable variables in. We use the notation V $\models e1 \hookrightarrow e2$ to indicate that given the aforementioned set V, the translation rules transform a program code e1 with pointers and & operators into a new program e2 expressible in our core language (Section A.2.1). Most of the rules are straightforward. The most difficult part is to translate addressable variables that are passed by reference. Because scopes of reference parameters are beyond their procedures, we have to ensure that all instances of these variables are transformed into pseudo-heap locations. This is to ensure that any possible effects on the original variables can be entirely captured in the pseudo-heap locations.

An example translation is shown in Figure A-6(b). The addressable variable x of type int is transformed into a pointer to a pseudo-heap location of type int_ptr. The program pointer p becomes a pointer to the location which x refers to. Variable x will then be passed to a child thread. The procedure inc is also translated to reflect the fact that its reference parameter i has been transformed. In the specification, i::int_ptr$\langle old_i \rangle$ represents the fact that i is a variable of type int_ptr pointing to a pseudo-heap location containing certain value old_i. The original value of x is indeed captured in the value of the pseudo-heap location. In the translated program, when

$$\frac{not(isProcCall(e_1)) \quad v \in FV(e_1) \cap V \quad \rho{=}[\&v \mapsto v, v \mapsto v.val] \quad e_1'{=}\rho\ e_1 \quad V \models \{\ e_2\} \hookrightarrow \{e_2'\}}{V \models \{e_1;\ e_2\} \hookrightarrow \{e_1'; e_2'\}}\ [\textbf{TRANS–EXP}]$$

$$\frac{\rho{=}[{}^*p \mapsto p.val] \quad e_1{=}\rho\ e}{V \models \{t^*\ p;\ e\} \hookrightarrow \{t\_ptr\ p; e_1\}}\ [\textbf{TRANS–POINTER}]$$

$$\frac{(\&v \in e \vee v \in V) \quad V_1{=}V \cup \{v\} \quad V_1 \models e \hookrightarrow e_1}{V \models \{t\ v;\ e\} \hookrightarrow \{t\_ptr\ v = new\ t\_ptr(0); e_1; delete(v)\}}\ [\textbf{TRANS–VAR–DECL}]$$

$$\frac{\&v \in e \quad p\ fresh \quad \rho{=}[v \mapsto p] \quad e_1{=}\rho\ e \quad V_1{=}\rho\ V \quad V_2{=}V_1 \cup \{p\} \quad V_2 \models e_1 \hookrightarrow e_2}{V \models t\ pn(t\ v, ...)\{e\} \hookrightarrow t\ pn(t\ v, ...)\{t\_ptr\ p = new\ t\_ptr(v); e_2; delete(p)\}}\ [\textbf{TRANS–PARAM–VAL}]$$

$$\frac{v \in V \quad V \models e \hookrightarrow e_1 \quad (\Phi_{pr}', \Phi_{po}'){=}transSpec(v : t, \Phi_{pr}, \Phi_{po})}{\begin{array}{c}V \models t\ pn(\textbf{ref}\ t\ v, ...)\ \textbf{requires}\ \Phi_{pr}\ \textbf{ensures}\ \Phi_{po}\{e\} \\ \hookrightarrow t\ pn(t\_ptr\ v, ...)\ \textbf{requires}\ \Phi_{pr}'\ \textbf{ensures}\ \Phi_{po}'\{e_1\}\end{array}}\ [\textbf{TRANS–PARAM–REF}]$$

$$\frac{\begin{array}{c}fresh\ old\_v, new\_v \quad \rho{=}[v \mapsto old\_v, v' \mapsto new\_v] \\ \Phi_{pr1}{=}\rho\ \Phi_{pr} \quad \Phi_{pr}'{=}v{::}\texttt{t\_ptr}\langle old\_v\rangle\ \texttt{*}\ \Phi_{pr1} \\ \Phi_{po1}{=}\rho\ \Phi_{po} \quad \Phi_{po}'{=}v{::}\texttt{t\_ptr}\langle new\_v\rangle\ \texttt{*}\ \Phi_{po}\end{array}}{transSpec(v : t, \Phi_{pr}, \Phi_{po}){:=}(\Phi_{pr}', \Phi_{po}')}\ [\textbf{TRANS–SPEC}]$$

$$\frac{\begin{array}{c}V \models t\ pn(..., t\ v, ..., \textbf{ref t u}, ...)\ \textbf{requires}\ \Phi_{pr}\ \textbf{ensures}\ \Phi_{po}\{e\} \hookrightarrow \\ t\ pn(..., t\ v, ..., \textbf{ref t u}, ...)\ \textbf{requires}\ \Phi_{pr}'\ \textbf{ensures}\ \Phi_{po}'\{e_1\} \\ v \in V \quad \rho{=}[\&v \mapsto v, v \mapsto v.val] \quad v'{=}\rho\ v\end{array}}{V \models pn(..., v, ..., u, ...) \hookrightarrow pn(..., v', ..., u....)}\ [\textbf{TRANS–CALL}]$$

Figure A-7: Translation Rules for Eliminating Variable Aliasing

the main thread passes variable `x` to the child thread, the pseudo-heap location that `x` points to is also passed to the child thread. Therefore, before the child thread joins,

the main thread cannot access the pseudo-heap location (e.g. via `p.val`) because it no longer owns that location. Note that the pseudo-heap location is deleted at the end to prevent memory leak.

We propose this translation for verification purpose *only* and do not recommend it for compilation use due to performance deficiency since accessing heap-allocated locations is typically more costly than accessing program variables. Variable aliasing may also occur via parameter-passing when two reference parameters of a procedure refer to the same actual variable. Our variable permission scheme (as presented in Section A.2.2) disallows the possibility because a caller cannot have two full permissions of a variable to pass it by reference twice.

A.2.5 Discussion

Applicability of the Proposed Variable Permissions

In this section, we discuss the application of our variable permission scheme to popular concurrent programming models such as POSIX threads and Cilk.

Pthreads is considered one of the most popular concurrent programming models for C/C++ [21]. In Pthreads, when creating a new child thread, a main thread passes a pointer to a heap location to the child thread. We model this argument passing by giving a copy of that pointer to the child thread. Furthermore, Pthreads uses global variables to facilitate sharing among threads. If several threads need to concurrently read a shared global variable, the main thread holding the full permission of that variable can just give each child thread a copy of that variable through pass-by-value mechanism. If concurrent threads require write access to the same variables, these variables can be protected by mutex locks whose invariants hold full permissions of the variables. This allows concurrent but race-free accesses to shared global variables. In our system, mutable global variables are automatically converted into pseudo reference parameters for each procedure (that uses them) prior to verification. For shared global variables that are protected by mutex locks, although they are converted into

pseudo reference parameters, none of concurrent threads have the variables' full permissions. It is the locks' invariants that capture the full permissions. Permission annotations for these variables in each procedure are automatically inferred as shown in Section A.2.3.

Cilk is a well-known concurrent programming model originally developed at MIT and recently adopted by Intel [44]. In Cilk, the `spawn` keyword is used to create a new thread and to return the value of the procedure call instead of a thread identifier. Before the child thread ends, any accesses to that return value are unsafe. Our `fork` can have the same effect by passing an additional variable by reference to capture the return value. This guarantees data-race freedom because only the child thread has the full permission of that variable. More importantly, compared with Pthreads, Cilk provides more flexible parameter passing when creating a child thread. Multiple variables can be passed to a child thread either by value or by reference. This flexible passing can be naturally handled by our pass-by-value and pass-by-reference scheme.

Phased Accesses to Shared Variables

Our variable permission is designed as a simpler permission scheme that can be used where sufficient. For immutable variables that are shared by concurrent threads, the general guideline is to pass copies of those variables to the threads to enjoy safe accesses to those copies. Mutable variables can be shared but should be protected by mutex locks to ensure race-freedom because there are some threads mutating the variables. However, there is still a class of complex sharing patterns that cannot be directly handled by our scheme. For example, a thread holds a certain permission to read a shared variable and is guaranteed that no other threads can modify the variable (read phase). Later, it acquires additional permissions from other threads and/or lock invariants, and combines them into a full permission to modify the shared variable (write phase). This kind of *phased accesses* to shared variables cannot be verified without splitting a full permission into smaller partial permissions. In this case, the thread can hold a partial permission while the rest of permissions belong to other threads and/or lock invariants.

Under this circumstance, we propose to detect those variables that are accessed in a phased way, and transform them into pseudo-heap locations where a more complex reasoning scheme is utilized [51, 61, 65]. The translation is done in a similar way as shown in Section A.2.4. As a result, our general guideline is to readily use variables in most cases where the proposed variable permission scheme is sufficient, and to automatically and uniformly transform variables into pseudo-heap locations where necessary, i.e. in complex scenarios such as aliasing and phased accesses.

A.3 Comparative Remarks

In 1970s, Owicki-Gries [111] came up with the very first tractable proof method for concurrent programs that prevents conflicting accesses to variables using side-conditions. However, these conditions are subtle and hard for compilers to check because they involve examining the entire program [16, 114]. Recently, concurrent separation logic (CSL) [105] has been proposed to nicely reason about heap-manipulating concurrent programs but CSL still relies on side-conditions for dealing with variables. Smallfoot verifier [9] uses CSL as its underlying logic and therefore suffers from the same limitation. In contrast, our scheme brings variable permissions into the logic and therefore makes it easier to check for conflicting accesses to variables. "Variables as resource" [16, 112] has proposed to apply permission systems [15, 18], originally designed for heap locations, to variables. Recently, Reddy et. al. [114] reformulate the treatment of variables using the system of syntactic control of interference. They share the same idea of applying fractional permissions [18] to variables. However, these more complex permission schemes place higher burden on programmers to figure out the permission fractions used to associate to variables. To the best of our knowledge, we are not aware of any existing verifiers that have fully implemented the idea. Chalice [89, 90] ignores the treatment of variables in method bodies while Verifast [65, 66] simulates variables as heap locations. Although the underlying semantics of Holfoot [123] formalizes "variables as resource", its automatic verification system, which is based on Smallfoot, does not allow sharing variables

using fractional permissions. In contrast, our variable permission scheme is simpler, using either full or zero permissions, but is expressive enough to support popular programming models such as Pthreads [21] and Cilk [44]. Furthermore, while previous approaches assume theoretical programming languages without dynamic thread creation [16, 112, 123] and procedure [114], our variable permission scheme is integrated into a practical language with fork/join concurrency. We also presented an algorithm to automatically infer variable permissions and therefore reduce programmers' efforts for annotations. There is some work on automatic inference of access permissions in the literature [41, 55] but they only address permissions for heap locations. Reddy et. al. [114] is the very first work on inferring permissions for variables. However, their approach is different from ours. Firstly, while their approach is a two-pass algorithm over entire program syntax tree and proof outline, our approach can infer variable permissions directly from procedure specifications. Secondly, their work targets programs written in a theoretical language without procedures and dynamic thread creation while our approach supports more realistic programs with procedures and fork/join concurrency. Lastly, most work on verification has often disallowed variable aliasing by using side-conditions [105, 111] or via assertions [16, 51]. Therefore, our presented translation scheme to eliminate variable aliasing is orthogonal to their work since we provide a way to transform addressable variables into pointers to pseudo-heap locations, and thus enable reasoning about their behaviors in the same way as heap locations [51, 105]. In contrast to several informal translation tools [30, 83] which attempt to translate C/C++ programs with pointers into Java, we present a translation scheme with its formal semantics. Another difference is that while they focus on language translation, we aim towards facilitating program verification.

A.4 Summary

We have proposed a new permission system to ensure data-race freedom when accessing variables. Our scheme is simple but expressive to capture programming models such as POSIX threads and Cilk. Through a simple permission scheme for variables,

we have extended formal reasoning to popular concurrent programming paradigms that rely on variables. We have provided an algorithm to automatically infer variable permissions and thus reduced program annotations. We have also shown a translation scheme to eliminate variable aliasing and to facilitate verification of programs with aliases on variables. By intergrating our variable permission system into concurrent separation logic, we form a comprehensive reasoning framework capable of reasoning about race-free accesses to both heap-based data structures and stack-based program variables.

Appendix B

Soundness Proof for Threads as Resource

In this appendix, we discuss the soundness of our "threads as resource" approach. We first present the interleaving operational semantics of the language. We then prove the soundness of our approach with respect to the operational semantics.

Operational Semantics.

We define the interleaving operational semantics of programs with fork/join concurrency.

Definition 12 (Well-formedness)**.** *A program is well-formed if the following conditions hold:*

- *In the program text, there exists a procedure called main, which indicates the entry point of the program.*
- *Procedure names are unique within a program. Procedure parameters are unique within a procedure. Free variables in the body of a procedure are the procedure parameters.*
- *A normal procedure call or a fork statement mentions only procedure names*

defined in the program text. The number of actual parameters and formal parameters are equal.

A thread can be in one of three states: *running*, *dead*, and *aborted*. Our verification framework ensures that no thread ends up in an *aborted* state. A program state is *non-aborting* if no thread is in an *aborted* state. A program state is *final* if all threads are in a *dead* state.

Definition 13 (Thread State)**.** *A thread state σ is one of the following states:*

- $\mathbf{run}(s, \Gamma)$ *stating that the thread is running with remaining statement s and environment Γ. For brevity, Γ is assumed to be a partial function from object names to object references and from stack variables to values. Environment Γ resembles stack and heap in a program. An update at v with o in Γ is denoted as $\Gamma[v \mapsto o]$.*
- **dead** *stating that a thread has completed its execution.*
- **aborted** *stating a thread has performed an illegal operation, such as null-pointer dereference.*

Definition 14 (Program State)**.** *A program state Ψ consists of a thread specification pool Θ and a set of threads T. Θ maps from a thread identifier to its aggregate resource (which is book-kept when the thread is forked). Each thread in T is a pair of (τ, σ) representing thread identifier τ and thread state σ. The thread identifier τ is of type* `thrd` *while the thread state σ is defined above.*

We use m to indicate the identifier of the main thread, i.e. the thread executing the main procedure of the program. We denote Θ_m as a thread specification pool containing only information of the main thread. In other words, $\Theta_m(i) =$ `emp` if $\tau{=}m$, and $\Theta_m(\tau)$ is undefined otherwise.

Definition 15 (Execution)**.** *Execution of a program starts in the initial program state: $(\Theta_m, \{(m, \mathbf{run}(s, \emptyset))\})$, where s is the code fragment of the main procedure.*

$$(\Theta, \{(\tau, \mathbf{run}(\mathbf{if}\ \mathit{true}\ \mathbf{then}\ s_1\ \mathbf{else}\ s_2; s, \Gamma))\} \cup T) \rightarrow (\Theta, \{(\tau, \mathbf{run}(s_1; s, \Gamma))\} \cup T)$$

$$(\Theta, \{(\tau, \mathbf{run}(\mathbf{if}\ \mathit{false}\ \mathbf{then}\ s_1\ \mathbf{else}\ s_2; s, \Gamma))\} \cup T) \rightarrow (\Theta, \{(\tau, \mathbf{run}(s_2; s, \Gamma))\} \cup T)$$

$$\frac{\mathit{eval}(e, \Gamma) = b}{(\Theta, \{(\tau, \mathbf{run}(\mathbf{if}\ e\ \mathbf{then}\ s_1\ \mathbf{else}\ s_2; s, \Gamma))\} \cup T) \rightarrow (\Theta, \{(\tau, \mathbf{run}(\mathbf{if}\ b\ \mathbf{then}\ s_1\ \mathbf{else}\ s_2; s, \Gamma))\} \cup T)}$$

$$\frac{\mathit{spec}(pn) := \mathtt{pn}(w_1, \ldots, w_n)\ \mathbf{requires}\ \Phi_{pr}\ \mathbf{ensures}\ \Phi_{po};\ \{\ s_1\ \} \quad s_1' = [v_1/w_1, \ldots, v_n/w_n]s_1}{(\Theta, \{(\tau, \mathbf{run}(pn(v_1, \ldots, v_n); s, \Gamma))\} \cup T) \rightarrow (\Theta, \{(\tau, \mathbf{run}(s_1'; s, \Gamma))\} \cup T)}$$

$$\frac{\begin{array}{c}\mathit{spec}(pn) := \mathtt{pn}(w_1, \ldots, w_n)\ \mathbf{requires}\ \Phi_{pr}\ \mathbf{ensures}\ \Phi_{po};\ \{\ s_1\ \} \\ \forall i \in \{1, \ldots, n\} \bullet \Gamma(v_i) = o_i \qquad \mathit{fresh}(\tau_1) \qquad \Gamma' = \Gamma[v \mapsto \tau_1] \\ \Gamma_1 = [w_1 \mapsto o_1, \ldots, w_n \mapsto o_n] \qquad \mathit{typeof}(\tau_1) = \mathtt{thrd} \\ \Theta_1 = \Theta[\tau_1 \mapsto \Phi_{po}]\end{array}}{\begin{array}{l}(\Theta, \{(\tau, \mathbf{run}(v = \mathbf{fork}(pn, v_1, \ldots, v_n); s, \Gamma))\} \cup T) \rightarrow \\ (\Theta_1, \{(\tau, \mathbf{run}(s, \Gamma'))\} \cup \{(\tau_1, \mathbf{run}(s_1; \mathtt{halt}, \Gamma_1))\} \cup T)\end{array}}$$

$$(\Theta, \{(\tau, \mathbf{run}(\mathtt{halt}, \Gamma))\} \cup T) \rightarrow (\Theta, \{(\tau, \mathbf{dead})\} \cup T)$$

$$\frac{\boxed{\exists (\tau_1, \mathbf{dead}) \in T \bullet \Gamma(v) = \tau_1}}{(\Theta, \{(\tau, \mathbf{run}(\mathbf{join}(v); s, \Gamma))\} \cup T) \rightarrow (\Theta, \{(\tau, \mathbf{run}(s, \Gamma))\} \cup T)}$$

Figure B-1: Selected Small-step Operational Semantics of Well-formed Programs with First-class Threads

Fig. B-1 shows the small-step operational semantics. A premise marked with box denotes the fact that threads must block and wait for the premise to become true. For example, joining with a thread blocks until the thread is dead. In Fig. B-1, $\mathit{spec}(pn)$ denotes the specification of the procedure pn in the program, $\mathit{eval}(e, \Gamma)$ denotes the evaluation of the expression e in the environment Γ. The rules for fork and join are of special interest. In the fork rule, a new thread is spawned and the return value

v points to its identifier τ_1 of type `thrd`. The resource carried by τ_1 is book-kept in Θ_1. We explicitly add a `halt` statement to signify the end of each newly spawned thread. As a quick observation, a thread identifier corresponds to a thread node in our logic. Any threads (joiners) knowing the identifier can perform a join operation to join with the newly-created thread (joinee). In the join rule, if the joinee has not yet finished its execution (i.e. it is not in a **dead** state), the joiners have to wait for the joinee to finish its execution. Note that when a joinee is joined, it will not be removed from the set of threads. This allows for the multi-join pattern and enables the joiners to immediately proceed without waiting in case the joinee has already finished its execution. There is a direct relation between the **dead** state of a thread during run-time and its *dead* predicate during verification-time.

Semantics

An separation formula is interpreted with respect to a thread identifier k of the active thread and the program state (Θ, T). The interpretation of most parts of our separation logic formulae is standard and can be found elsewhere [51]. Here we focus on the most interesting part. The interpretations of the thread node $t \mapsto \mathsf{thrd}\langle\Phi\rangle$ and the $dead(t)$ predicate are as follows:

$$(\Theta, \{(k, \mathbf{run}(s, \Gamma))\} \cup T) \models_k t \mapsto \mathsf{thrd}\langle\Phi\rangle \iff \exists\tau \cdot \Gamma(t){=}\tau \wedge \Theta(\tau){=}\Phi_1 \wedge \Phi \sqsubseteq \Phi_1$$

$$(\Theta, \{(k, \mathbf{run}(s, \Gamma))\} \cup T) \models_k dead(t) \iff \exists(\tau, \mathbf{dead}) \in T \ \cdot \ \Gamma(t){=}\tau$$

Intuitively, $\Phi{=}\Phi_1$ when there is only one thread node of the thread t in the program. $\Phi \sqsubset \Phi_1$ indicates the presence of multiple nodes of the same thread t. Our approach is sound due to two main arguments. First, our approach respects separation property, i.e. the resources carried by two thread nodes of the same thread are well-separated (Lemma 7). Second, our approach ensures that the total resource carried by all thread nodes of the same thread t is equal to the aggregate resource Φ_1 captured in Θ. The resource Φ_1 is book-kept when the thread t is forked and is flexibly split off and combined by our approach. To prove the above claim, we show that our "threads as resource" approach neither invents new resource nor destroys existing resource.

Therefore, it guarantees that the total resource of the program is not changed by our verification rules (Lemma 8). Note that our verification-time *dead(t)* predicate resembles the **dead** state of a thread during run-time.

Lemma 7 (Separation of Resources)**.** *If there exist two thread nodes* $t \mapsto \mathsf{thrd}\langle\Phi_1\rangle$ *and* $t \mapsto \mathsf{thrd}\langle\Phi_2\rangle$ *of the same thread* t*,* Φ_1 *and* Φ_2 *are well-separated. In other words,* $\Phi_1 * \Phi_2$ *is a valid separation logic formula.*

Proof. Splitting and combining thread nodes resort to splitting and combining their carried resources (described in the rule **R–THRD1** of Fig. 3-5). The splitting and combining of the resources follow the standard fractional permission accounting [15, 18]. Therefore, this lemma could be proven by induction on the structure of separation formulae (Fig. 3-3). □

Lemma 8 (Conservation of Total Resource)**.** *The total resource of a program is unchanged under "threads as resource" verification approach.*

Proof. Except for fork and join, the verification rules in Fig. 3-4 and sub-structural rules in Fig. 3-5 neither invent new resource nor destroy existing resource. Fork and join create and respectively consume thread nodes. Thread nodes can be considered as placeholders for the resources that they carry. Hence, this lemma can be proven by induction on the sub-structural rules (Fig. 3-5). □

Lemma 9 (Soundness of Threads as Resource)**.** *Given a program with a set of procedures* P^i *together with their corresponding pre/post-conditions* (Φ^i_{pr}/Φ^i_{po})*, if our verifier derives a proof for every procedure* P^i*, i.e.* $\{\Phi^i_{pr}\}P^i\{\Phi^i_{po}\}$ *is valid, then the program is race-free and partially correct.*

Proof. Data-race freedom is ensured due to the use of separation logic and fractional permissions, and the fact that resources carried in thread nodes are well-separated (Lemma 7). Partial correctness directly follows from Lemma 7, Lemma 8, and induction on the derivation of $\{\Phi^i_{pr}\}P^i\{\Phi^i_{po}\}$.

□

APPENDIX B. SOUNDNESS PROOF FOR THREADS AS RESOURCE

Appendix C

Soundness Proof for Verification of Deadlock Freedom

In this appendix, we prove that our framework proposed in Chapter 4 guarantees deadlock freedom with respect to the language described in Fig. 4-5. The deadlock problem is well-known, and one of the most cited definitions of deadlocks is by Coffman et al. [27]. Four conditions must hold for a deadlock to occur: (1) "mutual exclusion", (2) "no preemption", (3) "wait for", and (4) "circular wait". In our framework, the first three deadlock conditions hold: use of (mutex) locks (condition 1), a lock cannot be preempted until it is released (condition 2), threads may have to wait when acquiring a lock or joining another thread (condition 3), and we ensure deadlock freedom by breaking the "circular wait" (condition 4).

Our proof is inspired by the proof for deadlock freedom made by Leino et al. [91]. In contrast to their proof which focuses on lock operations and channel send/receive, our proof focuses on lock operations and thread fork/join instead. As a reminder, there is a wait-for graph corresponding to each program state. We prove that for each program that has been successfully verified by our framework, there does not exist a state whose wait-for graph contains a cycle.

A thread can be in one of three states: *running*, *dead*, and *aborted*. Our verification framework ensures that no thread ends up in an *aborted* state. A program state is

non-aborting if no thread is in an *aborted* state. A program state is *final* if all threads are in a *dead* state.

Definition 16 (Thread State)**.** *A thread state σ is one of the following states:*

- $\mathbf{run}(s, \Gamma)$ *stating that the thread is running with remaining statement s and environment Γ. For brevity, Γ is assumed to be a partial function from object names to object references and from stack variables to values. Environment Γ resembles stack and heap in programs. An update at v with o in Γ is denoted as $\Gamma[v \mapsto o]$.*
- **dead** *stating that a thread has completed its execution.*
- **aborted** *stating a thread has performed an illegal operation, such as null-pointer dereference.*

Definition 17 (Program State)**.** *In the presence of (mutex) locks, a program state Ψ consists of:*

- *L representing a partial function from locks to locklevels. Thus, $L(o)$ denotes the locklevel of lock o. A lock is already allocated if $o \in dom(L)$.*
- *T representing a set of threads. Each thread is a tuple $(\tau, \sigma, \varrho, ls)$ consisting of thread identifier τ, thread state σ, set of locks ϱ which the thread intends to acquire since the beginning of its execution, and set of locks ls currently held by the thread.*

For simplicity, we omit the thread specification pool Θ from the program state. Θ is used to capture resource belonged to a thread and can be handled in the same way as described in the soundness proof of "threads as resource" (Appendix B). We use m to denote the identifier of the main thread executing the main procedure of the program.

Definition 18 (Execution)**.** *Execution of a program starts in the initial program state: $(\ \emptyset, \{\ (m, \mathbf{run}(s, \emptyset), \emptyset, \emptyset)\ \})$, where s is the code of the main procedure.*

Fig. C-1 shows the small-step operational semantics. A premise marked with box denotes the fact that threads must block and wait for the premise to become true. For example, a thread can only acquire a lock which is not held by any thread. A premise marked with light grey indicates conditions that need to hold, otherwise the thread has performed an illegal operation and it transitions to an aborted state. For example, a thread will abort if it attempts to release a lock without holding it. Our framework ensures that the premises in light grey hold, i.e. threads cannot transition to aborted states. The rules presented require that a thread starts and completes its execution with an empty lockset.

In Fig. C-1, $def(pn)$ denotes the definition of the procedure pn in the program, $eval(e, \Gamma)$ denotes the evaluation of the expression e in the environment Γ, $delayed(\Phi, \Gamma)$ denotes the set of locks that a thread intends to acquire since the beginning of its execution (i.e. the delayed lockset). $delayed(\Phi, \Gamma)$ is defined in Definition 19 based on the thread's pre-condition Φ and an environment Γ.

Definition 19 (Delayed Lockset)**.** *Let Φ be a specification (described in Section 4.2.3) whose free variables are in dom(Γ). The delayed lockset of Φ is defined as follows:*

$$delayed(\Phi_1 \vee \Phi_2, \Gamma) = delayed(\Phi_1, \Gamma) \cup delayed(\Phi_2, \Gamma)$$
$$delayed([\bigwedge \omega \ \# \ \bigwedge \psi] \wedge \pi, \Gamma) = delayed(\bigwedge \psi, \Gamma)$$
$$delayed(\psi_1 \wedge \psi_2, \Gamma) = delayed(\psi_1, \Gamma) \cup delayed(\psi_2, \Gamma)$$
$$delayed(x \in \mathbf{LS}, \Gamma) = \{\Gamma(x)\}$$

Definition 20 (Wait-for Graph)**.** *Each program state (L, { $(\tau_1, \sigma_1, \varrho_1, ls_1)$, ..., $(\tau_n, \sigma_n, \varrho_n, ls_n)$ }) forms a directed wait-for graph whose nodes are the threads in the program state. This graph contains an arc from thread $(\tau_{t_1}, \sigma_{t_1}, \varrho_{t_1}, ls_{t_1})$ to thread $(\tau_{t_2}, \sigma_{t_2}, \varrho_{t_2}, ls_{t_2})$ if one of the following conditions holds:*

- *Thread t_1 blocks waiting for thread t_2 to release a lock. In other words, σ_{t_1} is $\mathbf{run}(\mathbf{acquire}(x); s, \Gamma_{t_1})$,$\Gamma_{t_1}(x) \in ls_{t_2}$, and σ_{t_1} cannot go to an aborted state.*
- *Thread t_1 blocks waiting for thread t_2 to terminate. In other words, σ_{t_1} is $\mathbf{run}(\mathbf{join}(\tau_{t_2}); s, \Gamma_{t_1})$, and σ_{t_1} cannot go to an aborted state.*

$$\frac{o \notin dom(L) \qquad typeof(o) = \texttt{lock} \qquad \Gamma(w) = level \qquad level{>}0 \qquad \Gamma' = \Gamma[v \mapsto o] \qquad L' = L[o \mapsto level]}{(L, \{(\tau, \mathbf{run}(v = \mathbf{new}\ \texttt{lock}(w); s, \Gamma), \varrho, ls)\} \cup T) \rightarrow (L', \{(\tau, \mathbf{run}(s, \Gamma'), \varrho, ls)\} \cup T)}$$

$$\frac{def(pn) := \texttt{pn}(w_1, \ldots, w_n)\ \mathbf{requires}\ \Phi_{pr}\ \mathbf{ensures}\ \Phi_{po};\ \{\ s_1\ \} \qquad s_1' = [v_1/w_1, \ldots, v_n/w_n]s_1}{(L, \{(\tau, \mathbf{run}(pn(v_1, \ldots, v_n); s, \Gamma), \varrho, ls)\} \cup T) \rightarrow (L, \{(\tau, \mathbf{run}(s_1'; s, \Gamma), \varrho, ls)\} \cup T)}$$

$$\frac{def(pn) := \texttt{pn}(w_1, \ldots, w_n)\ \mathbf{requires}\ \Phi_{pr}\ \mathbf{ensures}\ \Phi_{po};\ \{\ s_1\ \} \qquad \forall i \in \{1, \ldots, n\} \bullet \Gamma(v_i) = o_i \qquad fresh(\tau_1) \qquad typeof(\tau_1) = \texttt{thrd} \qquad \Gamma_1 = [w_1 \mapsto o_1, \ldots, w_n \mapsto o_n] \qquad \varrho_1 = delayed(\Phi_{pr}, \Gamma') \qquad \Gamma' = \Gamma[v \mapsto \tau_1]}{(L, \{(\tau, \mathbf{run}(v = \mathbf{fork}(pn, v_1, \ldots, v_n); s, \Gamma), \varrho, ls)\} \cup T) \rightarrow (L, \{(\tau, \mathbf{run}(s, \Gamma'), \varrho, ls)\} \cup \{(\tau_1, \mathbf{run}(s_1, \Gamma_1), \varrho_1, \emptyset)\} \cup T)}$$

$$\frac{\boxed{\exists(\tau_1, (\mathbf{dead}, \Gamma), \_, \_) \in T \bullet \Gamma(v) = \tau_1}}{(L, \{(\tau, \mathbf{run}(\mathbf{join}(v); s, \Gamma), \varrho, ls)\} \cup T) \rightarrow (L, \{(\tau, \mathbf{run}(s, \Gamma), \varrho, ls)\} \cup T)}$$

$$\frac{\Gamma(x) = o \qquad \boxed{\forall(\_, \_, \_, ls_t) \in T \bullet o \notin ls_t} \qquad ls' = ls \cup \{o\} \qquad o \notin ls \qquad \forall l \in dom(L) \bullet l \in ls \Rightarrow L(l) < L(o)}{(L, \{(\tau, \mathbf{run}(\mathbf{acquire}(x); s, \Gamma), \varrho, ls)\} \cup T) \rightarrow (L, \{(\tau, \mathbf{run}(s, \Gamma), \varrho, ls')\} \cup T)}$$

$$\frac{\Gamma(x) = o \qquad o \in ls \qquad ls' = ls - \{o\}}{(L, \{(\tau, \mathbf{run}(\mathbf{release}(x); s, \Gamma), \varrho, ls)\} \cup T) \rightarrow (L, \{(\tau, \mathbf{run}(s, \Gamma), \varrho, ls')\} \cup T)}$$

$$\frac{ls = \emptyset}{(L, \{(\tau, \mathbf{run}(\texttt{skip}, \Gamma), \varrho, ls)\} \cup T) \rightarrow (L, \{(\tau, \mathbf{dead}, \varrho, \emptyset)\} \cup T)}$$

Figure C-1: Small-step Operational Semantics for Well-formed Programs with Threads and Locks

Each program state Ψ has a corresponding directed wait-for graph. A *deadlock* occurs if the wait-for graph contains a cycle. Theorem 8 states that an arc in the graph between t_1 and t_2 implies that t_1's waitlevel is smaller than t_2's waitlevel or lockset ls_1 of t_1 does not contain the lock that t_2 is waiting to acquire while t_1 is waiting for t_2 at a join point. Theorem 9 states that, for each program state, there is

always a thread that is able to make progress. Following from Theorem 9, Theorem 10 states the main soundness theorem for deadlock-freedom.

Theorem 8 (Arc in Wait-for Graph)**.** *If the wait-for graph corresponding to a non-aborting program state has an arc from* $(\tau_{t_1}, \sigma_{t_1}, \varrho_{t_1}, ls_{t_1})$ *to* $(\tau_{t_2}, \sigma_{t_2}, \varrho_{t_2}, ls_{t_2})$*, then one of the following properties holds:*

- $max\{L(o) \mid o \in ls_{t1}\} < max\{L(o) \mid o \in ls_{t2}\}$
- σ_{t_1} *equals* $\mathbf{run}(\mathbf{join}(\tau_{t_2}); s, \Gamma_{t_1})$*, and* $ls_{t_1} \cap \varrho_{t_2} = \emptyset$

Proof. Since there is an arc from t_1 to t_2, t_1 cannot go into an aborted state. We consider two cases:

- **Acquire**. If the first statement of t_1 is $\mathbf{acquire}(x)$ and Γ is t_1's environment with $\Gamma(x) = o$, then it follows from the premise that

  $$\forall l \in dom(L) \bullet l \in ls_{t_1} \Rightarrow L(l) < L(o) \quad \text{or} \quad max\{L(l) \mid l \in ls_{t_1}\} < L(o)$$

  Because $o \in ls_{t_2}$, this implies $L(o) \leq max\{L(l) \mid l \in ls_{t_2}\}$. The first property holds.

- **Join**. The delayed lockset checking ensures that t_1 is not holding any locks that t_2 is going to acquire, that is, $ls_{t_1} \cap \varrho_{t_2} = \emptyset$. The second property holds.

□

Theorem 9 (Deadlock Freedom)**.** *If a program state* Ψ *is non-final and non-aborting, then* Ψ *is not stuck.*

Proof. By proving that there is always a thread that is able to make progress, i.e. the graph corresponding to Ψ contains a non-final thread t that has no outgoing arc. If the first statement s_1 of t is neither **acquire** nor **join**, then t can make progress. If s_1 is an $\mathbf{acquire}(x)$, then no other thread holds the lock x (otherwise t would have an outgoing arc). Hence, t can acquire x. If s_1 is $\mathbf{join}(id)$, the thread with identifier id has completed its execution (otherwise t would have an outgoing arc). Therefore, t can make progress. □

Theorem 10 (Soundness)**.** *Given a program with a set of procedures P^i and their corresponding pre/post-conditions (Φ^i_{pr}/Φ^i_{po}), if our verifier derives a proof for every procedure P^i, i.e. $\{\Phi^i_{pr}\}P^i\{\Phi^i_{po}\}$ is valid, the program is deadlock-free.*

Proof. It follows from Theorem 9 that each program that has been successully verified by our framework never gets stuck due to deadlocks. □

Appendix D

Soundness Proof for Verification of Barrier Synchronization

In this appendix, we show that our approach proposed in Chapter 5 guarantees correct synchronization of dynamic barriers. As dynamic barriers are more general than static barriers, the soundness also implies correct synchronization of static barriers. We first present an encoding of join operations in terms of barrier operations. This encoding simplifies the proof rules and soundness arguments to only focusing on barrier operations. We then proceed to the soundness arguments of our verification approach. Note that our approach currently does not consider non-termination due to infinite loops/recursion or deadlocks.

Encoding of Join Operations

Join operations can be encoded via barriers. Intuitively, each forked procedure receives an extra parameter `b` of type `barrier` and a unit permission to wait on that barrier. Before forking a child thread, a new barrier with two participants is created and passed to the child thread. The child thread will wait on that barrier before it terminates. A thread can join another thread by waiting on the corresponding barrier of the latter.

We present details of the encoding. Given a forked procedure *pn* which is defined

as $\texttt{pn}(\texttt{w}_1, \ldots, \texttt{w}_\texttt{n})$ **requires** Φ_{pr} **ensures** Φ_{po}; { s }, we (1) create a clone `pn_clone` of `pn`, (2) add one more parameter b of type `barrier` to its list of its parameters, (3) add a barrier wait at the end of the procedure, and (4) modify its specification as follows:

$\texttt{pn\_clone}(\texttt{w}_1, \ldots, \texttt{w}_\texttt{n}, \texttt{b})$
requires $\Phi_{pr} * b \overset{1,2}{\longmapsto} \mathsf{barrier}(0)$
ensures $\Phi_{po} * b \overset{1,2}{\longmapsto} \mathsf{barrier}(1)$;
{ `s; wait(b);` }

Then, we encode `thrd id=fork(pn,`$\texttt{w}_1$`, ..., `$\texttt{w}_\texttt{n}$`);` as `barrier b = new barrier(2); thrd id=fork(pn_clone,`$\texttt{w}_1$`, ..., `$\texttt{w}_\texttt{n}$`,b);` and encode `join(id)` as `wait(b)`. It is easy to see that the encoding results in correct synchronization of the newly added barrier `b`: two threads (the forker and the forkee) have unit permissions to access `b` and they both wait on `b` just once.

Soundness of Dynamic Bounded Permissions

We prove that, besides boundedness, our dynamic bounded permission system exercises properties of a standard access permission system: it allows concurrent reads and exclusive write. That is, we prove that, when using our verification and permission rules in Fig. 5-9, splitting and combining from any partial permissions never result in a full permission unless all partial permissions of b are combined. In this section, for brevity, we often refer to a permission $b \overset{c,t,a}{\longmapsto} \mathsf{barrier}(p)$ by its quantity (c, t, a).

Let S_b and t_b denote the set of all partial permissions and respectively the permission total of a barrier b.

Corollary 11 (Full Permission)**.** *Combining all partial permissions of a barrier b results in a full permission of b.*

Proof. First, the permission total t_b of a barrier b can only be safely changed by the rule [D–FULL]. Otherwise, t_b remains unchanged under the rest of permission rules and verification rules in Fig. 5-9. Hence, we would like to prove that $\sum_{(c_i,\_,\_)\in S_b} c_i = t_b + \sum_{(\_,\_,a_i)\in S_b} a_i$ holds. We prove it by induction on the verification and permission

rules. The equality trivially holds when the barrier b is created. Destroy and wait operations does not affect the quantity of permissions. Add and remove operations add and respectively subtract the same amount to/from c and a of a barrier node, hence the equality holds under the operations. All permission rules also maintain the equality. □

Corollary 12 (Permission Invariant)**.** $\forall(c, t_b, a) \in S_b\ ,\ c{>}a.$

Proof. The invariant $c{>}a$ trivially holds when a barrier b is created. Destroy and wait operations does not affect the quantity of permissions. Add and remove operations add and respectively subtract the same amount to/from c and a of a barrier node, hence the invariant holds under the operations.

We prove that split/combine rules also maintain the invariant.

For the rule [**D–SPLIT**], we have:

- $c{>}a$ or $\frac{a}{c}{<}1$
- $a_1{=}\frac{c_1}{c}{\cdot}a$ and $a_2{=}\frac{c_2}{c}{\cdot}a$

Hence, we conclude that $c_1{>}a_1$ and $c_2{>}a_2$.

For the combine rules [**D–COMBINE–1**] and [**D–COMBINE–2**], we have:

- $c_1{>}a_1$ and $c_2{>}a_2$
- $c{=}c_1{+}c_2$ and $a{=}a_1{+}a_2$

Hence, we conclude that $c{>}a$. □

Lemma 10 (Soundness of Dynamic Bounded Permission)**.** *Given a barrier b, our approach ensures that splitting and combining from any partial permissions of b never result in a full permission unless all partial permissions of b are combined.*

Proof. First, it follows from Corollary 11 that combining all partial permissions in S_b resulting in a full permission of b. We then show that it is impossible to combine a strict subset of S_b into a full permission of b.

Assume there exists a strict subset S of all partial permissions of b such at combining partial permissions in S results in a full permission of b. We have $S \subset S_b$. We define $\bar{S}$ the set of partial permissions of b not in S, that is $S_b = S \cup \bar{S}$.

Combining all permissions in S_b results in a full permission:

$$\sum_{(c_i,\_,\_)\in S_b} c_i \quad = \quad t_b \quad + \sum_{(\_,\_,a_i)\in S} a_i \tag{D.1}$$

As $S_b = S \cup \bar{S}$ and (D.1), we have:

$$\sum_{(c_k,\_,\_)\in S} c_k \quad + \quad \sum_{(c_j,\_,\_)\in \bar{S}} c_j \quad = \quad t_b \quad + \quad \sum_{(\_,\_,a_k)\in S} a_k \quad + \quad \sum_{(\_,\_,a_j)\in \bar{S}} a_j \tag{D.2}$$

Combining permissions in S also results in a full permission:

$$\sum_{(c_k,\_,\_)\in S} c_k \quad = \quad t_b \quad + \sum_{(\_,\_,a_k)\in S} a_k \tag{D.3}$$

From (D.2) and (D.3), we have the equality:

$$\sum_{(c_j,\_,\_)\in \bar{S}} c_j \quad = \quad \sum_{(\_,\_,a_j)\in \bar{S}} a_j \tag{D.4}$$

This contradicts to Corollary 12 as $c{>}a$ forall (c, t_b, a); hence $\sum_{(c_j,\_,\_)\in \bar{S}} c_j \quad > \quad \sum_{(\_,\_,a_j)\in \bar{S}} a_j$. □

Soundness of Verifying Barrier Synchronization

We first define what it means for a program to be correctly synchronized with respect to a dynamic barrier.

Definition 21 (Compatible Phasing)**.** *Given a dynamic barrier b with the last phase p (also called final phase), a thread is said to operate on b in a compatible number of phases p_1 iff:*

- *If it fully participates in b (i.e. it does not drop out), then p1=p.*
- *If it drops out, then p1≤p.*

Definition 22 (Correct Dynamic Synchronization)**.** *A program is correctly synchronized with respect to a dynamic barrier b iff:*

- *There are exactly a predefined number of threads participating in the barrier b's wait operations.*
- *Participating threads operate on b in compatible numbers of phases.*

Note that in case of static barriers, threads are not allowed to drop out. Therefore, compatible phasing implies that all participants fully participate and operate in the same numbers of phases.

In a program with barriers, a thread can be in one of four states: *running*, *waiting*, *dead*, and *aborted*. Our verification approach ensures that no thread reaches an *aborted* state. A program state is *non-aborting* if neither of threads are in an *aborted* state. A program state is *final* if all threads are in a *dead* state.

Definition 23 (Thread State)**.** *A thread state σ is one of the following states:*

- **run**(s, Γ) *stating that the thread is running with remaining statement s and environment Γ. For brevity, Γ is assumed to be a partial function from object names to object references and from stack variables to values. Environment Γ resembles stack and heap in programs.*
- **wait**(o, s, Γ) *stating that the thread is waiting at barrier object o with remaining statement s and environment Γ.*
- **dead** *stating that a thread has completed its execution.*
- **aborted** *stating a thread has performed an illegal operation.*

Threads in a program wait at barrier points and proceed in phases. We distinguish between local phase and global phase of a barrier. When a participant reaches a barrier point, it increments its local phase. When all participants have reached that point, the global phase will be incremented. If a thread still participates in a barrier,

its local phase is at most one ahead of the global phase. Intuitively, after reaching a barrier point and incrementing its local phase, a participant can only proceed if its local phase is equal to the global phase. This semantics has the advantage that a participant only needs to know its local phase and the global phase without worrying about the phases of other participants.

Definition 24 (Program State)**.** *A program state Ψ consists of:*

- *G representing a partial function from barrier objects to tuples (i, t, p) where i is the number of participants that have been suspended (i.e. waiting to proceed to the next phase), t is the total number of participants, and p is the current global phase of barrier object o. We write $G_i(o)$, $G_t(o)$, and $G_p(o)$ denote i, t, and p respectively. A barrier object o is already allocated if $o \in dom(G)$.*
- *T representing a set of threads. Each thread is a tuple (τ, σ, L) consisting of thread identifier τ, thread state σ, and a local barrier map L. L maps barriers to their corresponding local phases.*

For simplicity, we omit the thread specification pool Θ from the program state. Θ is used to capture resource belonged to a thread and can be handled in the same way as described in the soundness proof of "threads as resource" (Appendix B). We use m to denote the identifier of the main thread executing the main procedure of the program.

Definition 25 (Execution)**.** *Execution of a program starts in the initial program state: $(\ \emptyset,\ \{(m, \mathbf{run}(s, \emptyset), \emptyset)\}\ )$, where s is the code of the main procedure.*

Small-step operational semantics is presented in Fig. D-1. In the figure, $def(pn)$ denotes the definition of the procedure pn in the program, $eval(e, \Gamma)$ denotes the evaluation of the expression e in the environment Γ. A premise marked with light grey indicates conditions that need to hold, otherwise the thread has performed an illegal operation and it transitions to an aborted state. For example, a thread adds or removes to/from a barrier a negative number of participants. Our verification rules

ensure that the premises in light grey hold, i.e. threads cannot transition to aborted states.

$$\frac{o \notin dom(G) \qquad typeof(o) = \texttt{barrier} \qquad \Gamma(n) = num \qquad num{>}0 \qquad \Gamma' = \Gamma[b \mapsto o] \qquad G' = G[o \mapsto (0, num, 0)] \qquad L' = L[o \mapsto 0]}{(G, \{(\tau, \texttt{b = new barrier(n);s}, \Gamma), L)\} \cup T) \rightarrow (G', \{(\tau, \textbf{run}(s, \Gamma'), L')\} \cup T)}$$

$$\frac{def(pn) := \texttt{pn}(\texttt{w}_1, \ldots, \texttt{w}_\texttt{n}) \textbf{ requires } \Phi_{pr} \textbf{ ensures } \Phi_{po}; \{\ \texttt{s}_1\ \} \qquad \texttt{s}'_1 = [\texttt{v}_1/\texttt{w}_1, \ldots, \texttt{v}_\texttt{n}/\texttt{w}_\texttt{n}]\texttt{s}_1}{(G, \{(\tau, \textbf{run}(\texttt{pn}(\texttt{v}_1, \ldots, \texttt{v}_\texttt{n}); \texttt{s}, \Gamma), L)\} \cup T) \rightarrow (G, \{(\tau, \textbf{run}(\texttt{s}'_1; \texttt{s}, \Gamma), L)\} \cup T)}$$

$$\frac{\begin{array}{c}def(pn) := \texttt{pn}(\texttt{w}_1, \ldots, \texttt{w}_\texttt{n}) \textbf{ requires } \Phi_{pr} \textbf{ ensures } \Phi_{po}; \{\ \texttt{s}_1\ \} \\ \forall i \in \{1, \ldots, n\} \bullet \Gamma(\texttt{v}_\texttt{i}) = o_i \qquad fresh(\tau_1) \qquad typeof(\tau_1) = \texttt{thrd} \\ \Gamma_1 = [\texttt{w}_1 \mapsto o_1, \ldots, \texttt{w}_\texttt{n} \mapsto o_n] \qquad \Gamma' = \Gamma[\texttt{v} \mapsto \tau_1] \\ L_1 = [(o_i, G_p(o_i)) \mid \Gamma(\texttt{v}_\texttt{i}) = o_i \wedge typeof(o_i) = \texttt{barrier}]\end{array}}{\begin{array}{c}(G, \{(\tau, \textbf{run}(\texttt{v} = \texttt{fork}(\texttt{pn}, \texttt{v}_1, \ldots, \texttt{v}_\texttt{n}); \texttt{s}, \Gamma), L)\} \cup T) \rightarrow \\ (G, \{(\tau, \textbf{run}(\texttt{s}, \Gamma'), L)\} \cup \{(\tau_1, \textbf{run}(\texttt{s}_1, \Gamma_1), L_1)\} \cup T)\end{array}}$$

$$\frac{\Gamma(\texttt{b}) = o \qquad G(o) = (i, t, p) \qquad i{<}t{-}1 \qquad G' = G[o \mapsto (i{+}1, t, p)] \qquad L' = L[o \mapsto L(o){+}1]}{(G, \{(\tau, \textbf{run}(\texttt{wait}(\texttt{b}); \texttt{s}, \Gamma), L)\} \cup T) \rightarrow (G', \{(\tau, \textbf{wait}(o, \texttt{s}, \Gamma), L')\} \cup T)}$$

$$\frac{\Gamma(\texttt{b}) = o \qquad G(o) = (i, t, p) \qquad i{=}t{-}1 \qquad G' = G[o \mapsto (0, t, p{+}1)] \qquad L' = L[o \mapsto L(o){+}1]}{(G, \{(\tau, \textbf{run}(\texttt{wait}(\texttt{b}); \texttt{s}, \Gamma), L)\} \cup T) \rightarrow (G', \{(\tau, \textbf{wait}(o, \texttt{s}, \Gamma), L')\} \cup T)}$$

$$\frac{L(o) = G_p(o)}{(G, \{(\tau, \textbf{wait}(o, \texttt{s}, \Gamma), \Gamma), L)\} \cup T) \rightarrow (G, \{(\tau, \textbf{run}(\texttt{s}, \Gamma), L)\} \cup T)}$$

$$\frac{\Gamma(\texttt{b}) = o \qquad \Gamma(\texttt{m}) = a \qquad a{>}0 \qquad G(o) = (i, t, p) \qquad G' = G[o \mapsto (i, t + a, p)]}{(G, \{(\tau, \textbf{run}(\texttt{add}(\texttt{b}, \texttt{m}); \texttt{s}, \Gamma), L)\} \cup T) \rightarrow (G', \{(\tau, \textbf{run}(\texttt{s}, \Gamma), L)\} \cup T)}$$

$$\frac{\Gamma(\texttt{b}) = o \quad \Gamma(\texttt{m}) = a \quad G(o) = (i, t, p) \quad t{\geq}a{>}0 \quad t{-}a{>}i \quad G' = G[o \mapsto (i, t{-}a, p)]}{(G, \{(\tau, \textbf{run}(\texttt{remove}(\texttt{b}, \texttt{m}); \texttt{s}, \Gamma), L)\} \cup T) \rightarrow (G', \{(\tau, \textbf{run}(\texttt{s}, \Gamma), L)\} \cup T)}$$

$$\frac{\Gamma(\texttt{b}) = o \quad \Gamma(\texttt{m}) = a \quad G(o) = (i, t, p) \quad t{\geq}a{>}0 \quad t{-}a{\leq}i \qquad G' = G[o \mapsto (0, t{-}a, p{+}1)]}{(G, \{(\tau, \textbf{run}(\texttt{remove}(\texttt{b}, \texttt{m}); \texttt{s}, \Gamma), L)\} \cup T) \rightarrow (G', \{(\tau, \textbf{run}(\texttt{s}, \Gamma), L)\} \cup T)}$$

$$(G, \{(\tau, \textbf{run}(\texttt{skip}, \Gamma), L)\} \cup T) \rightarrow (G, \{(\tau, \textbf{dead}, L)\} \cup T)$$

Figure D-1: Small-step Operational Semantics of Programs with Barriers

Most of the rules in Fig. D-1 are straightforward. When forking a new child thread, the main thread passes the global phase to the child thread. The treatment of loops is similar to that of if-then-else and is omitted. When issuing a barrier wait, a thread transitions to a waiting state. The final thread issuing a barrier wait increments the global phase p by 1 and resets the counter i to 0. Threads transition back to a running state when all participants have issued a barrier wait, i.e. the global phase is equal to threads' local phases.

Lemma 11 (Correct Participation)**.** *Given a program with a barrier b and a set of procedures P^i together with their corresponding pre/post-conditions (Φ^i_{pr}/Φ^i_{po}), if our verifier derives a proof for every procedure P^i, i.e. $\{\Phi^i_{pr}\}P^i\{\Phi^i_{po}\}$ is valid, then there are exactly a predefined number of threads participating in b's wait operations.*

Proof. Our verification rules rely on bounded permissions to handle concurrent accesses to barrier b. Given n is the predefined number of participants, it follows from Lemma 2 that there are at most n threads concurrently operating on barrier b. In order to perform a wait on barrier b, threads must have unit permissions of barrier b. Additionally, adding and removing participants correspond to the addition and subtraction of the total number of participants t in operational semantics, i.e. $t_b + \sum_{(c_i,t_b,a_i)\in S_b} a_i = t$ where t_b is the original number of participants declared at b's creation point. Hence, there are exactly n threads participating in barrier b. □

Lemma 12 (Correct Phasing)**.** *Given a program with a barrier b and a set of procedures P^i together with their corresponding pre/post-conditions (Φ^i_{pr}/Φ^i_{po}), if our verifier derives a proof for every procedure P^i, i.e. $\{\Phi^i_{pr}\}P^i\{\Phi^i_{po}\}$ is valid, then threads participating in barrier b operate in compatible numbers of phases.*

Proof. The phase number used in our barrier specification corresponds to the local phase in the operational semantics. The final phase of b corresponds to the global phase of b after all participants have completed their execution. First, if a thread fully participates in barrier b (it does not drop out), then it ends up in a local phase which is equal to the global phase. Second, if a participant drops out, it ends up

in a local phase which is at most equal to the global phase. Third, if a thread does not fully participate in barrier b, does not drop out, and ends up in a phase which is not the final phase, it will be rejected by the db-consistency check (described in Section 5.2.3). Hence, all participants end up in compatible numbers of phases. □

Lemma 13 (Soundness of Verifying Barrier Synchronization)**.** *Given a program with a barrier b and a set of procedures P^i together with their corresponding pre/post-conditions (Φ^i_{pr}/Φ^i_{po}), if our verifier derives a proof for every procedure P^i, i.e. $\{\Phi^i_{pr}\}P^i\{\Phi^i_{po}\}$ is valid, then the program is correctly synchronized with respect to the barrier b.*

Proof. It directly follows from Lemma 11, Lemma 12, and Definition 22. □