

Verification of Concurrent Programs via Refinement Proofs

by

Süha Orhun Mutluergil

A Thesis Submitted to the
Graduate School of Engineering
in Partial Fulfillment of the Requirements for
the Degree of

Doctor of Philosophy

in

Computer Science and Engineering

Koç University

June, 2018

Koç University
Graduate School of Sciences and Engineering

This is to certify that I have examined this copy of a doctoral dissertation by

Süha Orhun Mutluergil

and have found that it is complete and satisfactory in all respects,
and that any and all revisions required by the final
examining committee have been made.

Committee Members:

Prof. Attila Gürsoy

Assoc. Prof. Constantin Enea (co-advisor)

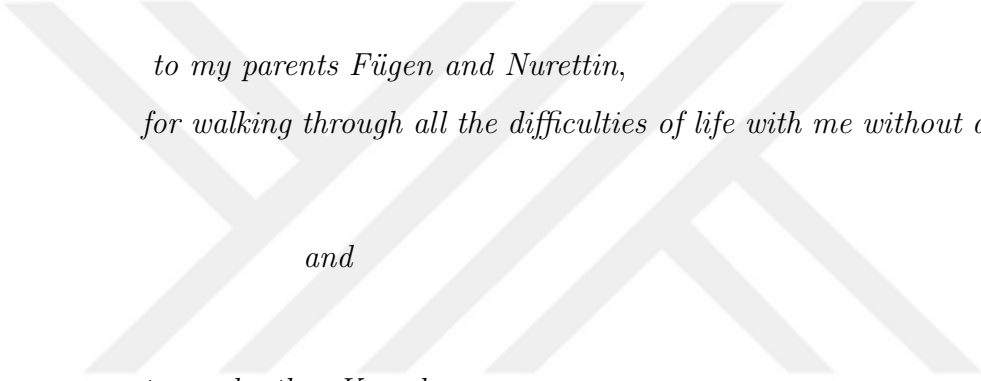
Prof. Ahmed Bouajjani

Assoc. Prof. Öznur Özkasap

Assist. Prof. Didem Unat

Assist. Prof. Tankut Barış Aktemur

Date: _____



*to my parents Fügen and Nurettin,
for walking through all the difficulties of life with me without any hesitation,*

and

*to my brother Kemal,
for bringing joy and hope to my life.*

ABSTRACT

Multi-core processors are preferred in many technological devices due to their increasing performance and reduced costs. They perform a range of tasks from helping people in their daily routines (i.e., via apps in cell-phones) to assist vital missions (i.e., flight control systems, surgery assistant devices, military databases and operating system kernels). To harness full potential of these devices, programmers need to develop efficient concurrent libraries.

However, writing a concurrent library is a difficult and error-prone task. To ensure correctness of the library, the programmer needs to consider all possible interleavings of the concurrent execution units and provide synchronization mechanisms whenever necessary to prevent undesired behaviors. Most of the time, testing is inconclusive and standard debugging tools are incapable of exposing bugs. Formal treatment is necessary, especially for mission critical software components.

In this thesis, we focus on ensuring safety properties of concurrent libraries i.e., something bad never happens or the program never reaches to a bad state. However, assertion checking for arbitrary programs is undecidable for the concurrent settings we consider. To overcome this undecidability barrier, researchers propose stronger correctness criteria that depend on the executions of the program instead of reachable states. Basically, refinement proofs are used to demonstrate that executions of both the specification program and the implementation show similar properties. Based on the choice of the *property* on executions, correctness proofs can be efficiently performed.

For instance, *linearizability* is the standard correctness criterion for concurrent data structures like stacks and queues. It allows us to establish observational refinement relation between the concurrent implementation and the atomic specification.

Linearizability can be shown by establishing forward or backward simulations between systems. In particular, finding a backward simulation relation is more difficult for concurrent data structures. In addition, existence of a backward or forward simulation is never guaranteed. As a part of this thesis, we show that contrary to common belief, many complex implementations including the Herlihy&Wing Queue and the Time-Stamped Stack can be proven correct by finding forward simulations. Our result on existence of forward simulations leads to simple and natural correctness proofs that are amenable to automation.

There are other ways of proving linearizability. Lipton’s reduction theory combined with program abstraction techniques is a strong tool for performing refinement proofs. Another part of the thesis describes a linearizability proof for the concurrent Chase-Lev Work-Stealing Queue (WSQ) implementation on Sequentially Consistent (SC) memory. The proof is mechanized and performed by using CIVL proof system. The proof consists of naturally structured refinement layers. The lowest level description is the WSQ implementation with the finest grained actions of which atomicity is guaranteed by the hardware. Through the upper layers, atomic blocks get coarser by using reduction, abstraction techniques and Owicki-Gries (OG) annotations. The highest layer consists of a simple atomic action for each queue operation that is tight enough to show linearizability.

The last part of the thesis presents a method for showing robustness of a program running under the Total Store Ordering (TSO) memory model, i.e., all of its TSO executions are equivalent to SC executions. This method utilizes SC reduction and abstraction techniques to address TSO robustness problem. In addition, we introduce an abstraction technique that allows us to obtain robust programs from non-robust programs by over-approximating both TSO and SC executions. This abstraction yields programs tight enough so that using existing reasoning methods for the SC semantics is sufficient for proving TSO invariants. We have evaluated these methods on a large set of benchmarks using CIVL proof system.

ÖZETÇE

Çok çekirdekli işlemciler, artan başarımları ve düşen fiyatları sebebiyle birçok teknolojik cihazda tercih edilmektedirler. Günlük işlerde insanlara yardım etmekten (cep telefonu uygulamaları ile olduğu gibi), hayati öneme haiz vazifelere (uçuş kontrol sistemleri, ameliyat yardım araçları, askeri veritabanları ve işletim sistemlerinde olduğu gibi) yardım etmeye kadar olan aralıkta birçok görev yerine getirirler. Bu cihazların tam gücünü kullanabilmek için, programcılar yüksek verimli, koşut-zamanlı yazılım kütüphaneleri geliştirmelidir.

Fakat, koşut-zamanlı kütüphane yazımı zor ve hataya açık bir iştir. Kütüphanenin doğruluğunu sağlamak için, programcı olası bütün binişimleri dikkate almalı ve gerekli durumlarda istenmeyen davranışları önlemek için eşleme yöntemlerine başvurmalıdır. Çoğu zaman test yöntemi doğruluğu sağlamak için sonuçsuz kalır ve standart hata ayıklama araçları hataları ortaya çıkartmakta yetersiz kalır. Özellikle yaşamsal yazılım parçaları için, biçimsel yaklaşım şarttır.

Bu tezde, koşut-zamanlı kütüphanelerin emniyet özelliklerini doğrulamaya, yani kötü bir şey asla olmaz veya program kötü bir hale asla erişemezi göstermeye odaklandık. Fakat, herhangi bir program için sav doğrulama problemi, bizim ilgilendiğimiz koşut-zamanlı ortamlar için *karar-verilemez* bir problem. Bu “*karar-verilemezlik*” engelini aşmak için, araştırmacılar hal-erişilebilirliği yerine programın yürütümlerine bağlı olan daha güçlü doğruluk kıstasları önermişlerdir. Temel olarak, geliştirme ispatları, hem belirtim programının hem de gerçekleştirme programının yürütümlerinin benzer özellikler gösterdiğini ispatlamakta kullanılır. Yürütümler üzerindeki “*özellik*” seçimine göre doğruluk ispatları etkin bir şekilde yapılabilir.

Mesela, *sıralanabilirlik*, yığıt ve kuyruk gibi koşut-zamanlı veri yapıları için standart doğruluk kıstasıdır. Bizim koşut-zamanlı gerçekleştirme ile atomik belirtim arasında

gözlensel geliştirme ilişkisi kurmamıza olanak sağlar. Sıralanabilirlik, sistemler arasında ileri veya geri simülasyon ilişkisi kurularak gösterilebilir. Özellikle koşul-zamanlı veri yapıları için, bir geri simülasyon ilişkisi bulmak daha zordur. Ek olarak, bir geri veya ileri simülasyon ilişkisinin varlığı hiçbir zaman kesin değildir. Bu tezin bir parçası olarak, genel kanının aksine, Herlihy&Wing Kuyruğu ve Zaman-Damgalı Yığıtı da içeren birçok karmaşık gerçeklemenin ileri simülasyon ilişkisi bularak doğruluğunun ispatlanabileceğini gösteriyoruz. İleri simülasyonların varlığına dair sonucumuz otomatikleştirilebilir, doğal ve basit ispatlara yol açıyor.

Sıralanabilirliği ispatlamak için başka yöntemler de mevcuttur. Soyutlama teknikleriyle birleştirilmiş Lipton'un indirgeme kuramı geliştirme ispatları için güçlü bir araçtır. Bu tezin başka bir kısmı, Ardışık Sıralı (SC) hafıza üzerinde çalışan Chase-Lev İş-Çalan Kuyruğu (WSQ) gerçeklemesinin bir sıralanabilirlik ispatını tasvir ediyor. İspat mekanize edilmiştir ve CIVL ispat sistemi ile yapılmıştır. İspat doğal bir şekilde yapılandırılmış geliştirme katmanlarından oluşmaktadır. En alt seviye belitleme, atomikliği donanım tarafından garanti edilmiş en ince taneli işlemler ile WSQ gerçeklemesidir. Üst katmanlara doğru, indirgeme, soyutlama ve Owicki-Gries (OG) notlarıyla atomik bloklar daha iri taneli hale gelir. En üst katman, her kuyruk yöntemi için, sıralanabilirliği göstermeye yetecek kadar sıkı, basit atomik işlemlerden müteşekkildir.

Tezin son bölümünü ise Tam Saklama Sırası (TSO) hafıza modeli altında çalışan programların *sağlamlığını*, yani bütün TSO yürütmelerinin SC yürütmelerine eşdeğer olduğunu, göstermek için bir yöntem sunuyor. Bu yöntem TSO sağlamlık sorununa hitap etmek için SC indirgeme ve soyutlama tekniklerinden faydalaniyor. Ek olarak, sağlam olmayan programların TSO ve SC yürütmelerini fazla yaklaşık tahmin ederek sağlam programlar elde edebilmek için bir soyutlama tekniği takdim ediyoruz. Bu soyutlama TSO değişmezlerini ispatlamak için var olan SC akıl yürütme yöntemlerini kullanmanın yeterli olacağı sıklıkta programlar hasıl ediyor. Bu yöntemleri geniş bir kıyaslama kümesinde CIVL ispat sistemini kullanarak değerlendirdik.

ACKNOWLEDGMENTS

First, I would like to express my gratitude to Dr. Constantin Enea and Dr. Ahmed Bouajjani. They have taught me how to identify, approach and solve theoretical problems. Their experience, guidance and comments play a significant role in all of the studies presented in this thesis. I would like to thank Dr. Serdar Taşiran and Dr. Attila Gürsoy for advising me during the first and second half of my studies, respectively. Their patience and support helped me to develop my academic skills and do research in a correct and meaningful way. Also, I would like to thank other members of the thesis committee, Dr. Öznur Özkasap, Dr. Didem Unat and Dr. Tankut Barış Aktemur, for their guidance and suggestions throughout the studies and their critical reading.

I am grateful to the Scientific and Technical Research Council of Turkey (TUBİTAK) for providing financial support for my studies under the National Doctoral Funding Programme 2211-E.

I would like to thank Dr. Shaz Qadeer for the support and help with using and understanding the CIVL proof assistant. His broad experience and command on theories over concurrent programs are carefully reflected on CIVL which makes CIVL an elegant tool not just for applying methods but also understanding the semantic models in a simple and easy way. I would like to thank Dr. Tayfun Elmas, a former member of the Koç - Microsoft Multicore Research Center, for the help and support with understanding QED, the ancestor of CIVL.

I would like to thank Dr. Gustavo Petri and other members of verification group at IRIF for the helpful discussions and contributions to this thesis. Also, I am grateful to Dr. Dan Grossman, Dr. Michael Ernst, Dr. Kivanç Muşlu and other members of the PLSE group at UW for their hospitality, lively discussions and broadening my

horizons during my visit to UW.

I am grateful to Dr. Tosun Terzioğlu, Dr. Albert Kohen Erkip, Dr. Kemal İnan and Dr. Alev Topuzoğlu for guiding me and helping me discover the exciting world of mathematics, theory and proofs. I will try to keep and spread the excitement and love of discovering things for the rest of my life.

I am thankful to all my friends from Koç University, Rustamzhon Melikov, Dr. Selim Bahadır, İsmail Kuru, Erkut Demirhan, Dr. Erdal Mutlu, Dr. Burcu Külahcioğlu Özkan, Maryam Dabaghchian, Dr. Hassan Salehe Matar, Can Bekar, Oğuz Doğan, Barış Ötüş, Aslıhan Çeliker and Ömer Kırnap, for their close friendship and sharing the long nights of studies at Koç with me. In addition, I would like to thank my close friends Selman Ergünay, Gökalp Erbeyoğlu and Ahmet Yöntem for their continuous support and friendship.

I would like to thank Emine Büyükdurmuş, Elif Tüysüz, Derya Ernur and Türkan Eryiğit for their help, patience and friendly assistance on administrative and financial issues.

Last but not least (in fact, the most), I am immensely grateful and in debt to my family, my father Nurettin, my mother Fügen and my brother Kemal for their limitless support and giving me courage to continue during the darkest hours of the night, enduring with me against all the difficulties, teaching me how to get up after falling down, their endless love, patience and good heart.

TABLE OF CONTENTS

List of Tables	xiv
List of Figures	xv
Nomenclature	xvii
Chapter 1: Introduction and the Context	1
1.1 Specifications	3
1.1.1 Safety and Liveness Properties	4
1.2 Program Refinements	5
1.2.1 Linearizability	6
1.3 Memory Models	8
1.3.1 Trace-Robustness	9
1.4 Organization and Contributions	10
Chapter 2: Proving Linearizability Using Forward Simulations	12
2.1 Introduction	12
2.2 Preliminaries	15
2.2.1 Libraries	15
2.2.2 Refinement and Linearizability	17
2.3 Refinement Proofs	18
2.4 Queues With Fixed Dequeue Linearization Points	20
2.4.1 Enqueue Methods With Non-Fixed Linearization Points	21
2.4.2 Abstract Queue Implementation	22
2.4.3 A Correctness Proof For Herlihy&Wing Queue	25

2.5	Stacks With Fixed Pop Commit Points	28
2.5.1	Pop Methods With Fixed Commit Points	28
2.5.2	Abstract Stack Implementation	31
2.5.3	A Correctness Proof For Time-Stamped Stack	35
2.6	Related Work	37
A	Libraries	39
B	Normal Forward/Backward Simulations	41
C	Proof of Theorem3	42
D	Proof of Theorem4	51
E	Proving the correctness of <i>TSS</i>	59
Chapter 3: A Correctness Proof of Chase-Lev WSQ by Abstraction and Reduction		72
3.1	Introduction	72
3.2	The Chase-Lev Work Stealing Queue Algorithm	73
3.3	Observations on the SC Executions of the Program	75
3.4	Path Splitting and Loop Peeling	77
3.5	Overview of Proof Methodology	80
3.6	Mechanized Proof Steps	83
3.7	Related Work	91
Chapter 4: Reasoning About TSO Programs Using Reduction and Abstraction		93
4.1	Introduction	93
4.2	Overview	95
4.3	TSO Robustness	100
4.4	A Reduction Theory for Checking Robustness	104
4.5	Abstractions and Verifying non-Robust Programs	109
4.6	Experimental Evaluation	112

4.7	Related Work	113
A	Properties of TSO Programs	116
B	A Hierarchy of TSO Consistency Models	121
Chapter 5: Conclusions and Future Work		125
Bibliography		127



LIST OF TABLES

4.1	TSO benchmark results	113
-----	---------------------------------	-----



LIST OF FIGURES

1.1	A simple two-threaded program	2
2.1	Herlihy & Wing Queue	21
2.2	Simulating queue histories with $AbsQ$	23
2.3	The transition relation of $AbsQ$	25
2.4	The Time-Stamped Stack	29
2.5	An execution of TSS	30
2.6	Simulating stack histories with $AbsS$	32
2.7	The transition relation of $AbsS$	34
2.8	The transition relation of $AbsQ_0$	43
2.9	The transition relation of $AbsS_0$	51
2.10	The flow diagram for the pop and push methods of the Time-Stamped Stack algorithm	60
2.11	Derivation rules for the push method of the TSS.	61
2.12	Derivation rules for the pop method of the TSS	62
3.1	Chase-Lev Work Stealing Queue Algorithm	74
3.2	Initial abstractions on steal method	77
3.3	Initial abstractions on take method	79
3.4	Proof layers for the mechanized proof of steal and put methods	84
3.5	Proof layers for the mechanized proof of take methods	85
4.1	An example message passing program and a sample trace	95
4.2	An example store buffering program.	97
4.3	Syntax of TSO programs	101

4.4	TSO semantics	102
4.5	An example robust program that needs a read abstraction to be write atomic	111
4.6	Old and new <i>so</i> relations example	116
4.7	An example store buffering program and an acyclic TSO trace	118



NOMENCLATURE

CIVL	Concurrent Intermediate Verification Language (tool)
FIFO	First-in First-out (Queue)
FOL	First-Order Logic
LIFO	Last-in First-out (Stack)
LTS	Labeled Transition System
OG	Owicki-Gries Method
SC	Sequential Consistency / Sequentially Consistent
ST	Set Theory
TSO	Total-Store Ordering
WSQ	Work-Stealing Queue

Chapter 1

INTRODUCTION AND THE CONTEXT

Moore's law states that the number of transistors in processors doubles approximately every two years. A rough interpretation of it is that the performance of the processors increase exponentially with time. Moore's law has predicted the improvements in the hardware correctly for the last three decades. However, the pace of advancements began to slow down in the past few years due to technical challenges and physical limitations.

To overcome this bottleneck, hardware manufacturers has began to produce multi-core processors. Multi-core processors become mainstream and they are utilized in our daily-life devices like PCs, tablets and smart phones. Performance of the software running on multi-core machines can still improve tremendously if they are running concurrently. Hence, many platforms and frameworks allow programmers to write concurrent programs to harness full power of these multi-core processors.

However, developing concurrent programs is more difficult and error-prone than developing sequential programs¹. This difficulty stems from two main sources. First, concurrent programs use shared memory locations during computations. Unexpected interleavings of concurrent execution units might corrupt shared data and result in incorrect computations. To prevent this, the programmer needs to take into consideration all possible interleavings and use synchronization mechanisms to prevent undesired interleavings. The programmer needs to be very careful while establishing synchronization and consider both performance and correctness during the develop-

¹Actually, static verification problem is undecidable [61] for both sequential and concurrent programs. Although there is no difference in theory, reasoning on concurrent programs have more practical difficulties.

Shared: F1 = F2 = 0			
P_0		P_1	
Local: lf2 = 1		Local: lf1 = 1	
1	F1 = 1;	6	F2 := 1;
2	while (lf2 == 1)	7	while (lf1 == 1)
3	lf2 := F2;	8	lf1 := F1;
4	<i>//Critical Section</i>	9	<i>//Critical Section</i>
5	F1 := 0;	10	F2 := 0;

Figure 1.1: A simple two threaded program that aims to provide mutual exclusion. **F1** and **F2** are shared integers. **lf1** and **lf2** are local integer variables of P_2 and P_1 , respectively.

ment.

Second, hardware, language or the platform that provides concurrency has a complex structure. Semantics of the underlying framework can be very complicated and the specification documents can be vague and difficult to understand.

To explain these problems in a clear way, consider the simple C-like program in Figure 1.1. In this program, there are two concurrent execution units (threads): P_0 and P_1 . Assume that each numbered line can be executed atomically. Shared variables **F1** and **F2** are used as synchronization variables and initialized as 0. Local variables **lf2** of P_0 and **lf1** of P_1 are initialized as 1. Suppose that the programmer wants to achieve mutual exclusion for critical sections at lines 4 and 9. For this purpose, P_0 first sets **F1** flag and then iteratively reads **F2** until it is lowered by P_1 . Then, it enters its critical section and lowers **F1** when it finishes its job. P_1 works similarly on the second flag **F2**.

If critical sections at lines 4 and 9 do not depend on some input parameters or outside configurations, testing helps little for ensuring correctness. There is a single initial configuration that needs to be tested. However, all possible interleavings of P_0 and P_1 must be considered to guarantee correctness. So, the programmer's only option with testing is to run the same program numerous times hoping that all possible

interleavings are covered during these runs. If there is a subtle bug that manifests itself in few executions, the programmer has a little chance to catch it.

Similar arguments prevail if the programmer decides to use a traditional debugger (i.e., one that allows stepwise monitoring of the program and provides breakpoints). The debugger is not aware or in control of the thread interleavings. Encountering a problematic interleaving is a matter of chance. Moreover, there is a class of bugs, called *heisenbugs*, that disappear when the program is monitored. So, if the program has a heisenbug, the programmer cannot see it with debugging tools.

Second, semantics of the concurrency model could be unclear or vague. Some compilers and architectures might reorder program statements or delay some of them for performance reasons. If the programmer is not fully aware of these optimizations, she might encounter some unexpected results. For our toy program in Figure 1.1, if the write operations at Lines 1 and 6 can be delayed beyond later operations of same threads, P_0 and P_1 could satisfy the conditions at lines 2 and 7 simultaneously and enter their critical sections together.

To overcome these difficulties, the programmer needs to verify the program using formal methods.

1.1 Specifications

In order to ensure that a program works correctly, a specification that describes what it is supposed to do must be written. We want to be precise and clear about the meaning of specifications. So, mathematics and logic are the first natural choices for describing them. In particular, First-Order Logic (FOL) and Set Theory (ST) are the basic tools that we use to express and reason about specifications.

There is a variety of statements that can be accepted as a program specification such as “This program runs under 5 seconds for all possible inputs”, “It works without crashing on the following configurations”, “Server response time must be less than 3 seconds for every query”, “This method never reaches a state in which both of the flags are raised” or “This program produces the same outputs with the other program

for all valid inputs”. In the scope of this thesis, we are not interested in performance related properties like the first and third sentences and vague sentences like the second one. We are interested in properties that can be described by FOL and ST like the last two statements.

Note that the specification we are looking for the toy example in Figure 1.1 can be described in FOL as: $\forall s \in S. \neg(pc_0 = 4 \wedge pc_1 = 9)$ where pc_0 and pc_1 are program counters of threads P_0 and P_1 , respectively, and S is the set of possible valuations of all variables (states) of this program. We would like to use following stronger version of this formula as the specification from now on: $\forall s \in S. \neg(lf1 = 1 \wedge lf2 = 1 \wedge pc_0 = 4 \wedge pc_1 = 9)$. It is actually equivalent to the original formula since $pc_0 = 4 \implies lf2 = 1$ and $pc_1 = 9 \implies lf1 = 1$.

1.1.1 Safety and Liveness Properties

Let us call properties we can describe by FOL and ST as logical properties. We can divide logical properties into two main categories: safety and liveness properties. Informally, safety properties assert that something bad never happens and liveness properties assert that something good eventually happens. The specification we have for the toy problem is a safety property since it asserts that the program cannot reach a state in which two variables contain these particular values. In general, mutual exclusion conditions can be described as safety properties. Similarly, program invariants and assertions on program variables and positions of threads are considered to be safety properties.

On the other hand, liveness properties are described over executions instead of states. For the toy example above, we can add to the specification that in all executions of the program, one of the threads eventually enters its critical section. This condition is known as deadlock freedom i.e., it is never the case that all threads are blocked at some point and none can take a step. Deadlock freedom is an important property that must be assured for concurrent programs that need synchronization. However, it is fundamentally different than assertions and program invariants. Therefore, formal

techniques for checking assertions and deadlock freedom are quite different. However, livelock freedom and starvation freedom are also liveness properties and they can be checked with similar methods.

In the scope of this thesis we are interested only in safety properties of programs. We can describe safety properties in other way as: “Is there an execution of the program that visits a state violating an assertion of the specification?” Unfortunately, in all of the settings we consider in this thesis, assertion violation problem is undecidable [52, 51]. One way of getting rid of this undecidability barrier is to require manual input and guidance from the user to complete the proof. Another way is to come up with stronger correctness notions that will imply the desired safety properties but that are decidable and can be computed efficiently. In general, stronger notions are incomplete in the sense that a subset of correct programs can be verified using them. Combination of requiring user guidance and stronger correctness notions yields efficient and useful verification mechanisms.

Usually, these stronger correctness notions require reasoning on executions of the program running on different semantic models. Similarly, if the program specification is another program running on another semantic model, relating executions of the implementation and the specification programs at some abstract level could be easier and more helpful for proving correctness than directly proving that they reach the same set of states. In both cases, the programmer needs to carry a refinement proof to show correctness of the program.

1.2 Program Refinements

Let P_1 and P_2 be two programs. They may be the same programs or they might be running on different semantic models. Both programs produce executions when they run. Basically, executions are sequences of program states linearly ordered according to a sequence of transitions (actions) coming from the operations of threads. Most of the time, we are not interested in the entire execution. We might want to focus on particular parts of the program state or consider only orderings of particular states

in the execution. Let us call the graph obtained by projecting the executions on the parts we are interested in as *traces*. Then, we say that P_1 refines P_2 , or P_2 abstracts P_1 , if and only if every trace of P_1 is also a trace of P_2 . In other words, for every execution of P_1 , there exists an execution of P_2 with the same trace. As a convention, we call P_1 as the implementation and P_2 as the specification.

Showing refinement of two programs is a common method for proving correctness. Let us consider the toy program in Figure 1.1 again. This figure represents the implementation. We can set the specification as the same program but each of the threads execute their statements atomically. To ease the reasoning process, we can keep track of only the ordering of actions generated by the first and the last statements of critical sections as traces. Then, showing refinement between the implementation and the specification will imply mutual exclusion of the implementation.

This is an example use of refinement proofs for a single program. This idea can be extended to get *linearizability* [36] which is a widely-accepted correctness criterion for concurrent libraries.

1.2.1 Linearizability

Consider concurrent libraries that implement a set of fixed methods. We assume that each method begins with a *call* statement and ends with a *return* statement. These statements are just for marking the borders of the methods in the executions and do not have any visible effects on program states. Then, a concurrent implementation P_1 is said to be linearizable with respect to the sequential specification P_2 with atomic method bodies, if we can replace actions of a method of P_1 between its call and return points with a single action executing the atomic body of the same method from P_2 , for every execution of P_1 . Usually, P_2 has the same method bodies of P_1 that executes atomically, like our toy example described previously. Then, linearizability implies that we can assume that each of this library's methods execute atomically between its call and return points.

Proving linearizability easily reduces to a refinement proof by taking the implemen-

tation and the specification as they are and defining traces as projection of executions on call and return actions.

One way of doing linearizability proofs (or similar refinement proofs) is to find a mapping from states reachable by P_1 to states reachable by P_2 such that call and return actions of P_1 maps to call and return actions of P_2 , respectively, only one action of every method of P_1 maps to the atomic single action of the same method of P_2 and every other actions of P_1 maps to stuttering (idle) actions of P_2 . Finding such a mapping, one can construct an execution of P_2 from an execution of P_1 in a forward or backward manners depending on the mapping such that both executions have the same traces.

As we will elaborate more later on Chapter 2, finding a relation that constructs the corresponding execution of P_2 from initial state in a forward manner (therefore called a forward simulation relation) is more preferable. Forward simulations are easier to come up by and they are amenable to automation. However, existence of a forward simulation is not always guaranteed. In Chapter 2, we give a result on existence of forward simulation relations for concurrent stack and queue implementations and show correctness of complicated and well-known concurrent stack and queue implementations by finding forward simulation relations.

Another way of doing linearizability proofs is to use *abstraction* and *reduction* [44] in an interleaving manner. These techniques are re-writing rules for the program such that granularity of atomic actions increase after each application.

Basically, abstraction replaces each statement with a new statement that allows more executions possible and more states reachable. This transformation is valid because every execution and reachable state of the original program is preserved in the abstract program and if the original program has a failing state or undesirable execution, then the abstract program contains it as well.

On the other hand, reduction composes two atomic statements into a single atomic statement if either the first one is a right mover or the second one is a left mover. We say that action α of a thread t commutes right of action β of another thread

u that follows α immediately in an execution π , if π can be *simulated* by another execution in which first u executes β followed by t executing α . If actions α obtained by executing a statement R commutes right of all actions of other threads in every execution of the program, then R is a right mover. Left mover can be defined in a dual manner.

By applying reduction and abstraction iteratively on concurrent libraries, one might make the method bodies atomic and show linearizability of the library. In Chapter 3, we provide an automated linearizability proof for a complicated WSQ implementation by using reduction and abstraction. Apart from being machine checked, this proof provides insight for the correctness intuition of the implementation.

1.3 Memory Models

Up to this point we have not provided a semantic model for the concurrent programs. Reason for this is that there is no single universal semantic model for describing them. Depending on the underlying processor architecture, programming language or platform, behaviors and reachable states of programs differ.

Sequential Consistency (SC) [41] is one of the semantic models for describing executions of concurrent programs. SC imposes two conditions over executions:

- All of the actions in an execution appear in some linear order.
- Actions from the same thread appear in the order specified by the program (program order).

SC is a simple and elegant model. Therefore, it is assumed implicitly by programmers and programming languages most of the time. We have had this assumption until now, as well. Linearizability is defined for SC executions of the programs and might be even not meaningful for some models we consider later on.

Most probably, the programmer of the example program in Figure 1.1 assumes SC implicitly since it is correct under this assumption. Arguing its correctness under SC is simple. We would like to use following stronger formula as the specification from

now on: $\forall s \in S. \neg(lf1 = 0 \wedge lf2 = 0 \wedge pc_0 = 4 \wedge pc_1 = 9)$. It is actually equivalent to the original formula since $\forall s \in S. pc_0 = 4 \implies lf2 = 0$ and $\forall s \in S. pc_1 = 9 \implies lf1 = 0$. Assume that, without loss of generality, $lf2 = 0$ holds when $pc_0 = 4$ and $pc_1 = 9$ at some state, in an execution. It means that Line 3 is executed before Line 6. Since Line 1 is executed before Line 3 and Line 6 is executed before Line 8 due to the program order, Line 1 must be executed before Line 8 in this execution. Hence, $lf1 = 1$ must hold at this state.

However, this program might not be correct under other semantic models. Consider Total-Store Ordering (TSO) model that describes behaviors of concurrent programs running on x86-Intel processor architecture. According to its operational description, each thread keeps a FIFO buffer. Statements that write on shared variables insert the new value and the variable address to this buffer. Statements that read from shared variables first check the local buffer. If the variable address is in the buffer, the latest value for this variable (the one closest to the tail of the queue) is returned as a result of the read action. Otherwise, the value in the shared memory is returned. The buffers are drained to the shared memory non-deterministically. For our example program, If the values written by the statements at Lines 1 and 6 are not drained until Lines 4 and 9, respectively, then we can reach a state in which $pc_0 = 4 \wedge pc_1 = 9$.

1.3.1 Trace-Robustness

TSO model allows more behaviors for programs than SC. Hence, reasoning on correctness of TSO programs is harder than SC. Checking correctness of assertions for arbitrary TSO and SC programs are both undecidable problems [52, 51]. Therefore, researchers have proposed stronger correctness criteria (consistency models) for TSO programs that can be checked effectively.

Trace-robustness is one of the consistency models that can be checked effectively and can be used for verifying non-trivial TSO programs. In this model, traces are orderings of read and write accesses on shared variables in an execution. Then, a

program is trace-robust if for all TSO executions of this program there exists an SC execution with the same trace. Equal traces imply equal end states for both TSO and SC execution. Hence, trace-robustness imply that every state reachable by a TSO execution is also reachable by an SC execution (state-robustness). Thus, if a program is trace-robust, verifying the program for SC executions guarantees its correctness for TSO executions in terms of safety properties.

Trace-robustness can be shown by performing refinement proofs as well. This time, the implementation is a program under TSO semantics and the specification is the same program under SC semantics. We consider the same traces that keeps ordering of read and write operations. Then, showing that the TSO version of the program refines the SC version implies trace-robustness. In Chapter 4, we propose a method for checking trace-robustness based on reduction and abstraction. In addition, we introduced a novel abstraction technique that can be used for verifying non-trace-robust TSO programs.

1.4 Organization and Contributions

This thesis consists of three studies on verification of concurrent libraries by refinement proofs. Each chapter describes a separate study. The rest of thesis is organized as follows:

- Chapter 2 describes a study on linearizability proofs [17]. Its main result is a theorem that ensures existence of forward simulation relations for concurrent stack and queue implementations. It introduces new stack and queue specifications for which forward simulations exist. We show applicability by proving correctness of complicated Herlihy& Wing Queue and Time-Stamped Stack implementations by finding forward simulations between the implementations and the new specifications. To the best of our knowledge, this study gives first formal proof for correctness of the Time-Stamped Stack.
- Chapter 3 introduces a correctness (linearizability) proof for a complicated WSQ

implementation [53]. Proof is done by using reduction and abstraction in a sequence of refinement proof layers. Each layer abstracts some part of the library or increases the granularity of atomic blocks and gives intuition about correctness of the implementation. In addition, the proof is performed by using CIVL proof system. To the best of our knowledge, this work provides first mechanized proof for this implementation.

- Last chapter provides techniques for verifying TSO programs [18]. First, we propose a method for showing trace-robustness of TSO programs based on abstraction and reduction. Our method requires reasoning on only SC executions of the program and can be implemented on any tool that allows abstraction and reduction reasoning. We extend notion of trace-robustness in harmony with reduction definitions such that our extended consistency criterion includes programs that are not trace-robust but state-robust. Second, we propose a method for verifying TSO programs that are not even state-robust. We introduce an abstraction mechanism that adds more behaviors to both SC and TSO programs and that could make programs trace-robust. Then, SC reasoning tools can be used for verifying weaker safety properties on the abstract program. This technique enables us to verify WSQ implementation of Chapter 3 which is not even state-robust.

Chapter 2

PROVING LINEARIZABILITY USING FORWARD SIMULATIONS

2.1 Introduction

Programming efficient concurrent implementations of atomic collections, e.g., stacks and queues, is error prone. To minimize synchronization overhead between concurrent method invocations, implementors avoid blocking operations like lock acquisition, allowing methods to execute concurrently. However, concurrency risks unintended inter-operation interference, and risks conformance to atomic reference implementations. Conformance is formally captured by (*observational*) *refinement*, which assures that all behaviors of programs using these efficient implementations would also be possible were the atomic reference implementations used instead.

Observational refinement can be formalized as a trace inclusion problem, and the latter can itself be reduced to an invariant checking problem, but this requires in general introducing history and prophecy variables [1]. Alternatively, verifying refinement requires in general establishing a forward simulation *and* a backward simulation [47]. While simulations are natural concepts, backward reasoning, corresponding to the use of prophecy variables, is in general hard and complex for programs manipulating data structures. Therefore, a crucial issue is to understand the limits of forward reasoning in establishing refinement. More precisely, an important question is to determine for which concurrent abstract data structures, and for which classes of implementations, it is possible to carry out a refinement proof using only forward simulations.

To get rid of backward simulations (or prophecy variables) while preserving completeness w.r.t. refinement, it is necessary to have reference implementations that

are *deterministic*. Interestingly, determinism allows also to simplify the forward simulation checking problem. Indeed, in this case, this problem can be reduced to an invariant checking problem. Basically, the simulation relation can be seen as an invariant of the system composed of the two compared programs. Therefore, existing methods and tools for invariant checking can be leveraged in this context.

But, in order to determine precisely what is meant by determinism, an important point is to fix the alphabet of observable events along computations. Typically, to reason about refinement between two library implementations, the only observable events are the calls and returns corresponding to the method invocations along computations. This means that only the external interface of the library is considered to compare behaviors, and nothing else from the implementations is exposed. Unfortunately, it can be shown that in this case, it is impossible to have deterministic atomic reference implementations for common data structures such as stacks and queues (see, e.g., [57]). Then, an important question is what is the necessary amount of information that should be exposed by the implementations to overcome this problem ?

One approach addressing this question is based on linearizability [36] and its correspondence with refinement [27, 16]. Linearizability of a computation (of some implementation) means that each of the method invocations can be seen as happening at some point, called *linearization point*, occurring somewhere between the call and return events of that invocation. The obtained sequence of linearization points along the computation should define a sequence of operations that is possible in the atomic reference implementation. Proving the existence of such sequences of linearization points, for all the computations of a concurrent library, is a complex problem [9, 14, 30]. However, proving linearizability becomes less complex when linearization points are fixed for each method, i.e., associated with the execution of a designated statement in its source code [14]. In this case, we can consider that libraries expose in addition to calls and returns, events signaling linearization points. By extending this way the alphabet of observable events, it becomes straightforward to

define *deterministic* atomic reference implementations. Therefore, proving linearizability can be carried out using forward simulations when linearization points are fixed, e.g., [64, 10, 63, 4]. Unfortunately, this approach is not applicable to efficient implementations such as the LCRQ queue [49] (based on the principle of the Herlihy&Wing queue [36]), and the Time-Stamped Stack [24]. The proofs of linearizability of these implementations are highly nontrivial, very involved, and hard to read, understand and automatize. Therefore, the crucial question we address is what is precisely the kind of information that is necessary to expose in order to obtain deterministic atomic reference implementations for such data structures, allowing to derive simple and natural linearizability proofs for such complex implementations, based on forward simulations, that are amenable to automation ?

We observe that the main difficulty in reasoning about these implementations is that, linearization points of enqueue/push operations occurring along some given computation, depend in general on the linearization points of dequeue/pop operations that occur arbitrarily far in the future. Therefore, since linearization points for enqueue/push operations cannot be determined in advance, the information that could be fixed and exposed can concern only the dequeue/pop operations.

One first idea is to consider that linearization points are fixed for dequeue/pop methods and only for these methods. We show that under the assumption that implementations expose linearizations points for these methods, it is possible to define deterministic atomic reference implementations for both queues and stacks. We show that this is indeed useful by providing a simple proof of the Herlihy&Wing queue (based on establishing a forward simulation) that can be carried out as an invariant checking proof.

However, in the case of Time-Stamped Stack, fixing linearization points of pop operations is actually too restrictive. Nevertheless, we show that our approach can be generalized to handle this case. The key idea is to reason about what we call *commit points*, and that correspond roughly speaking to the last point a method accesses to the shared data structure during its execution. We prove that by exposing commit points

(instead of linearization points) for pop methods, we can still provide deterministic reference implementations. We show that using this approach leads to a quite simple proof of the Time-Stamped Stack, based on forward simulations.

2.2 Preliminaries

We formalize several abstraction relations between libraries using a simple yet universal model of computation, namely labeled transition systems (LTS). This model captures shared-memory programs with an arbitrary number of threads, abstracting away the details of any particular programming system irrelevant to our development.

A *labeled transition system* (LTS) $A = (Q, \Sigma, s_0, \delta)$ over the possibly-infinite alphabet Σ is a possibly-infinite set Q of states with initial state $s_0 \in Q$, and a transition relation $\delta \subseteq Q \times \Sigma \times Q$. The i th symbol of a sequence $\tau \in \Sigma^*$ is denoted τ_i , and the empty sequence is denoted by ϵ . An *execution* of A is an alternating sequence of states and transition labels (called also actions) $\rho = s_0, e_0, s_1 \dots e_{k-1}, s_k$ for some $k > 0$ such that $\delta(s_i, e_i, s_{i+1})$ for each i such that $0 \leq i < k$. We write $s_i \xrightarrow{e_i \dots e_{j-1}}_A s_j$ as shorthand for the subsequence $s_i, e_i, \dots, s_{j-1}, e_{j-1}, s_j$ of ρ , for any $0 \leq i \leq j < k$ (in particular $s_i \xrightarrow{\epsilon} s_i$). The projection $\tau|_{\Gamma}$ of a sequence τ is the maximum subsequence of τ over alphabet Γ . This notation is extended to sets of sequences as usual. A *trace* of A is the projection $\rho|_{\Sigma}$ of an execution ρ of A . The set of executions, resp., traces, of an LTS A is denoted by $E(A)$, resp., $Tr(A)$. An LTS is *deterministic* if for any state s and any sequence $\tau \in \Sigma^*$, there is at most one state s' such that $s \xrightarrow{\tau} s'$. More generally, for an alphabet $\Gamma \subseteq \Sigma$, an LTS is Γ -*deterministic* if for any state s and any sequence $\tau \in \Gamma^*$, there is at most one state s' such that $s \xrightarrow{\tau'} s'$ and τ is a subsequence of τ' .

2.2.1 Libraries

Programs interact with libraries by calling named library *methods*, which receive *parameter values* and yield *return values* upon completion. We fix arbitrary sets $\mathbb{M}$ and $\mathbb{V}$ of method names and parameter/return values. We fix an arbitrary set $\mathbb{O}$ of

operation identifiers, and for given sets $\mathbb{M}$ and $\mathbb{V}$ of methods and values, we fix the sets

$$C = \{inv(m, d, k) : m \in \mathbb{M}, d \in \mathbb{V}, k \in \mathbb{O}\} \text{ and } R = \{ret(m, d, k) : m \in \mathbb{M}, d \in \mathbb{V}, k \in \mathbb{O}\}$$

of *call actions* and *return actions*; each call action $inv(m, d, k)$ combines a method $m \in \mathbb{M}$ and value $d \in \mathbb{V}$ with an *operation identifier* $k \in \mathbb{O}$. Operation identifiers are used to pair call and return actions. We may omit the second field from a call/return action a for methods that have no inputs or return values. For notational convenience, we take $\mathbb{O} = \mathbb{N}$ for the rest of this chapter.

A *library* is an LTS over alphabet Σ such that $C \cup R \subseteq \Sigma$. We assume that the traces of a library satisfy standard well-formedness properties, e.g., return actions correspond to previous call actions, which for lack of space are delegated to Appendix A. An operation k is called *completed* in a trace τ when $ret(m, d, k)$ occurs in τ , for some m and d . Otherwise, it is called *pending*.

The projection of a library trace over $C \cup R$ is called a *history*. The set of histories of a library L is denoted by $H(L)$. Since libraries only dictate methods executions between their respective calls and returns, for any history they admit, they must also admit histories with weaker inter-operation ordering, in which calls may happen earlier, and/or returns later. A history h_1 is *weaker* than a history h_2 , written $h_1 \sqsubseteq h_2$, iff there exists a history h'_1 obtained from h_1 by appending return actions, and deleting call actions, s.t.: h_2 is a permutation of h'_1 that preserves the order between return and call actions, i.e., if a given return action occurs before a given call action in h'_1 , then the same holds in h_2 .

A library L is called *atomic* when there exists a set S of sequential histories such that $H(L)$ contains every weakening of a history in S . Atomic libraries are often considered as specifications for concurrent objects. Libraries can be made atomic by guarding their methods bodies with global lock acquisitions.

A library L is called a *queue implementation* when $\mathbb{M} = \{enq, deq\}$ (enq is the method that enqueues a value and deq is the method removing a value) and $\mathbb{V} =$

$\mathbb{N} \cup \{\text{EMPTY}\}$ where EMPTY is the value returned by *deq* when the queue is empty. Similarly, a library L is called a *stack implementation* when $\mathbb{M} = \{\text{push}, \text{pop}\}$ and $\mathbb{V} = \mathbb{N} \cup \{\text{EMPTY}\}$. For queue and stack implementations, we assume that the same value is never added twice, i.e., for every trace τ of such a library and every two call actions $\text{inv}(m, d_1, k_1)$ and $\text{inv}(m, d_2, k_2)$ where $m \in \{\text{enq}, \text{push}\}$ we have that $d_1 \neq d_2$. As shown in several works [4, 15], this assumption is without loss of generality for libraries that are data independent, i.e., their behaviors are not influenced by the values added to the collection, which is always the case in practice. On a technical note, this assumption is used to define (Γ -)deterministic abstract implementations of stacks and queues in Section 2.4 and Section 2.5.

2.2.2 Refinement and Linearizability

Conformance of a library L_1 to a specification given as an “abstract” library L_2 is formally captured by (*observational*) *refinement*. Informally, we say L_1 refines L_2 iff every computation of every program using L_1 would also be possible were L_2 used instead. We assume that a program can interact with the library only through call and return actions, and thus refinement can be defined as history set inclusion. Refinement is equivalent to the *linearizability* criterion [36] when L_2 is an atomic library [27, 16].

Definition 1. A library L_1 refines another library L_2 iff $H(L_1) \subseteq H(L_2)$.

Linearizability [36] requires that every history of a concurrent library L_1 can be “linearized” to a sequential history admitted by a library L_2 used as a specification. Formally, a sequential history h_2 with only complete operations is called a *linearization* of a history h_1 when $h_1 \sqsubseteq h_2$. A history h_1 is *linearizable* w.r.t. a library L_2 iff there exists a linearization h_2 of h_1 such that $h_2 \in H(L_2)$. A library L_1 is *linearizable* w.r.t. L_2 , written $L_1 \sqsubseteq L_2$, iff each history $h_1 \in H(L_1)$ is linearizable w.r.t. L_2 .

Theorem 1 ([27, 16]). $L_1 \sqsubseteq L_2$ iff L_1 refines L_2 , if L_2 is atomic.

In the rest of this chapter, we discuss methods for proving refinement (and thus, linearizability) focusing mainly on queue and stack implementations.

2.3 Refinement Proofs

Library refinement is the instance of a more general notion of refinement between LTSs which for some alphabet Γ of *observable actions* is defined as the inclusion of sets of traces projected on Γ . Library refinement corresponds to the case $\Gamma = C \cup R$. Typically, Γ -refinement between two LTSs A and B is proved using *simulation relations* which roughly, require that B can mimic every step of A using a (possibly empty) sequence of steps. Mainly, there are two kinds of simulation relations, forward or backward, depending on whether the preservation of steps is proved starting from a similar state forward or backward. It has been shown that Γ -refinement is equivalent to the existence of *backward simulations*, modulo the addition of history variables that record events in the implementation, and to the existence of *forward simulations* provided that the right-hand side LTS B is Γ -deterministic [1, 47]. We focus on proofs based on forward simulations because they are easier to automatize.

In general, forward simulations are *not* a complete proof method for library refinement because libraries are not $C \cup R$ -deterministic (the same sequence of call/return actions can lead to different states depending on the interleaving of the internal actions). However, there are classes of atomic libraries, e.g., libraries with “fixed linearization points” (defined later in this section), for which it is possible to identify a larger alphabet Γ of observable actions (including call/return actions), and implementations that are Γ -deterministic. For queues and stacks, Section 2.4 and Section 2.5 define other such classes of implementations that cover all the implementations that we are aware of.

Let $L_1 = (Q_1, \Sigma, s_0^1, \delta_1)$ and $L_2 = (Q_2, \Sigma, s_0^2, \delta_2)$ be two libraries over Σ_1 and Σ_2 , resp., such that $C \cup R \subseteq \Sigma_1 \cap \Sigma_2$. Also, let Γ be a set of actions s.t. $C \cup R \subseteq \Gamma \subseteq \Sigma_1 \cap \Sigma_2$.

Definition 2. *The library L_1 Γ -refines L_2 iff $Tr(L_1)|\Gamma \subseteq Tr(L_2)|\Gamma$.*

Notice that Γ -refinement implies refinement for any Γ as in Definition 2.

We define a notion of *forward simulation* that can be used to prove Γ -refinement (a dual notion of *backward simulation* is defined in Appendix B). For a relation

$R \subseteq A \times B$, $R[X]$ is the set of elements related by R to elements of X , i.e., $R[X] = \{y : \exists x \in X. R(x, y)\}$.

Definition 3. A relation $fs \subseteq Q_1 \times Q_2$ is called a Γ -forward simulation from L_1 to L_2 iff $fs[s_0^1] = \{s_0^2\}$ and:

- If $(s, \gamma, s') \in \delta_1$, for some $\gamma \in \Gamma$, and $u \in fs[s]$, then there exists $u' \in fs[s']$ such that $u \xrightarrow{\sigma} u'$, $\sigma_i = \gamma$, for some i , and $\sigma_j \in \Sigma_2 \setminus \Gamma$, for each $j \neq i$.
- If $(s, e, s') \in \delta_1$, for some $e \in \Sigma_1 \setminus \Gamma$ and $u \in fs[s]$, then there exists $u' \in fs[s']$ such that $u \xrightarrow{\sigma} u'$ and $\sigma \in (\Sigma_2 \setminus \Gamma)^*$.

A Γ -forward simulation requires that every step of L_1 corresponds to a sequence of steps of L_2 . To imply Γ -refinement, every step of L_1 labeled by an observable action $\gamma \in \Gamma$ should be simulated by a sequence of steps of L_2 where exactly one transition is labeled by γ and all the other transitions are labeled by non-observable actions.

The following shows the soundness and the completeness of Γ -forward simulations (when L_2 is Γ -deterministic). It is an instantiation of previous results [1, 47].

Theorem 2. L_1 Γ -refines L_2 when there is a Γ -forward simulation from L_1 to L_2 . Moreover, if L_1 Γ -refines L_2 and L_2 is Γ -deterministic, then there is a Γ -forward simulation from L_1 to L_2 .

The linearization of a concurrent history can be also defined in terms of *linearization points*. Informally, a linearization point of an operation in an execution is a point in time where the operation is conceptually effectuated; given the linearization points of each operation, the linearization of a concurrent history is the sequential history which takes operations in order of their linearization points. For some libraries, the linearization points correspond to a fixed set of actions. For instance, in the case of atomic libraries where method bodies are guarded with a global-lock acquisition, the linearization point of every method invocation corresponds to the execution of the body. When the linearization points are fixed, we assume that the library is an LTS over an alphabet that includes actions $lin(m, d, k)$ with $m \in \mathbb{M}$, $d \in \mathbb{V}$ and $k \in \mathbb{O}$.

The action $lin(m, d, k)$ represents the linearization point of the operation k returning value d . Let Lin denote the set of such actions. The projection of a library trace over $C \cup R \cup Lin$ is called an *extended history*. A trace or extended history is called *Lin-complete* when every completed operation has a linearization point, i.e., each return action $ret(m, d, k)$ is preceded by an action $lin(m, d, k)$. A library L over alphabet Σ is called *with fixed linearization points* iff $C \cup R \cup Lin \subseteq \Sigma$ and every trace $\tau \in Tr(L)$ is *Lin-complete*.

Proving the correctness of an implementation L_1 of a concurrent object such as a queue or a stack with fixed linearization points reduces to proving that L_1 is a $(C \cup R \cup Lin)$ -refinement of an abstract implementation L_2 of the same object where method bodies are guarded with a global-lock acquisition. Since the abstract implementation is usually $(C \cup R \cup Lin)$ -deterministic, by Theorem 2, proving $(C \cup R \cup Lin)$ -refinement is equivalent to finding a $(C \cup R \cup Lin)$ -forward simulation from L_1 to L_2 .

Section 2.4 and Section 2.5 extend this result to queue and stack implementations where the linearization point of the methods *adding* values to the collection is *not* fixed.

2.4 Queues With Fixed Dequeue Linearization Points

The typical abstract implementation of a concurrent queue, denoted as $AbsQ_0$, maintains a sequence of values, the enqueue adds a value atomically to the beginning of the sequence, and the dequeue removes a value from the end of the sequence (if any, otherwise it returns **EMPTY**). Both methods have a fixed linearization point when the update of the sequence happens. For some queue implementations, e.g., the Herlihy&Wing Queue [36] (*HWQ* for short), there exists no forward simulation to $AbsQ_0$ although they are a refinement of $AbsQ_0$. The main reason is that the enqueue methods don't have a *fixed* linearization point. In this section, we propose a new abstract implementation for queues, denoted as $AbsQ$, which roughly maintains a *partially-ordered set* of values instead of a sequence. We show that there exists a forward simulation from any correct queue implementation where only the *dequeue* methods

```

void enq(int x){
    i = back++;
    items[i] = x;
}
int deq() {
    while (1) {
        range = back - 1;
        for (int i = 0; i <= range; i++){
            x = swap(items[i], null);
            if ( x != null ) return x;
        }
    }
}

```

Figure 2.1: Herlihy & Wing Queue. We assume that every statement is atomic.

have fixed linearization points (the enqueue methods are unconstrained) to *AbsQ*. This covers all the queue implementations that we are aware of, in particular *HWQ*, Baskets Queue [37], LCRQ [49], or Time-Stamped Queue [24] (where the enqueues don't have fixed linearization points). We also describe a forward simulation from *HWQ* to *AbsQ*.

2.4.1 Enqueue Methods With Non-Fixed Linearization Points

We describe *HWQ* where the linearization points of the enqueue methods are not fixed. The shared state consists of an array `items` storing the values in the queue and a counter `back` storing the index of the first unused position in `items`. Initially, all the positions in the array are `null` and `back` is 0. An enqueue method starts by reserving a position in `items` (`i` stores the index of this position and `back` is incremented so the same position can't be used by other enqueues) and then, stores the input value `x` at this position. The dequeue method traverses the array `items` starting from the beginning and atomically swaps `null` with the encountered value. If the value is not `null`, then the dequeue returns that value. If it reaches the end of the array, then it restarts.

The linearization points of the enqueues are not fixed, they depend on dequeues executing in the future. Consider the following trace with two concurrent enqueues

($i(k)$ represents the value of i in operation k): $inv(enq, x, 1)$, $inv(enq, y, 2)$, $i(1) = \text{bck}++$, $i(2) = \text{bck}++$, $items[i(2)] = y$. Assuming that the linearization point corresponds to the assignment of i , the history of this trace should be linearized to $inv(enq, x, 1)$, $ret(enq, 1)$, $inv(enq, y, 2)$, $ret(enq, 2)$. However, a dequeue executing until completion after this trace will return y (only position 1 is filled in the array $items$) which is not consistent with this linearization. On the other hand, assuming that enqueues should be linearized at the assignment of $items[i]$ and extending the trace with $items[i(1)] = x$ and a completed dequeue that in this case returns x , leads to the incorrect linearization: $inv(enq, y, 2)$, $ret(enq, 2)$, $inv(enq, x, 1)$, $ret(enq, 1)$, $inv(deq, 3)$, $ret(deq, x, 3)$.

The dequeue method has a fixed linearization point which corresponds to an execution of `swap` returning a non-null value. This action alone contributes to the effect of that value being removed from the queue. Every concurrent history can be linearized to a sequential history where dequeues occur in the order of their linearization points in the concurrent history. This claim is formally proved in Section 2.4.3.

Since the linearization points of the enqueues are not fixed, there exists no forward simulation from HWQ to $AbsQ_0$. In the following, we describe the abstract implementation $AbsQ$ for which such a forward simulation does exist.

2.4.2 Abstract Queue Implementation

Informally, $AbsQ$ records the happens-before order between enqueue operations for which the added value has not been removed by a dequeue operation. The linearization point of a dequeue operation with return value $d \neq \text{EMPTY}$ is enabled only if the happens-before stored in the current state contains a minimal enqueue that adds the value d . The effect of the linearization point is that the minimal enqueue is removed from the current state and the return value is recorded in the library state. When the return value is `EMPTY`, the linearization point of a dequeue is enabled only if the current state stores only pending enqueues (the dequeue overlaps with all the enqueue operations stored in the current state and it can be linearized before all of them). The

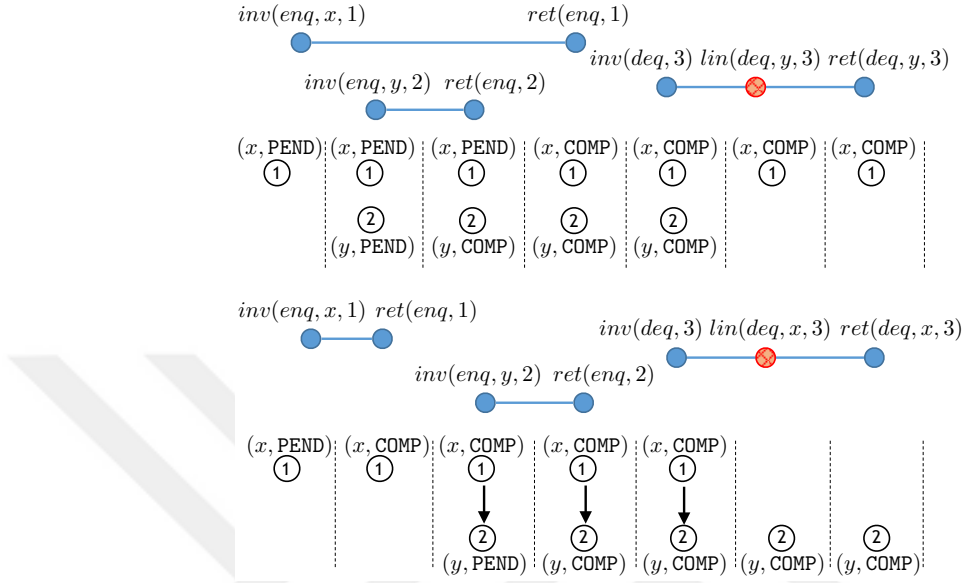


Figure 2.2: Simulating queue histories with $AbsQ$. The order between actions is from left to right.

return of a dequeue is enabled only if the returned value matches the one fixed at the linearization point.

Figure 2.2 pictures two executions of $AbsQ$ for two extended histories (that include dequeue linearization points). The state of $AbsQ$ after each action is pictured as a graph below the action. The nodes of this graph represent enqueue operations and the edges happens-before constraints. Each node is labeled by a value (the input of the enqueue) and a flag **PEND** or **COMP** showing whether the operation is pending or completed. For instance, in the case of the first history, the dequeue linearization point $lin(deq, y, 3)$ is enabled because the current happens-before contains a *minimal* enqueue operation with input y . Note that a linearization point $lin(deq, x, 3)$ is also enabled at this state.

Formally, the states of $AbsQ$ are tuples $\langle O, <, \ell, rv, cp \rangle$ where $O \subseteq \mathbb{O}$ is a set of operation identifiers, $< \subseteq O \times O$ is a strict partial order, $\ell : O \rightarrow \mathbb{V} \times \{\text{PEND}, \text{COMP}\}$ labels every identifier with a value and a pending/completed flag (the flag is used to track the happens-before order), $rv : \mathbb{O} \rightarrow \mathbb{V}$ records the return value of a dequeue fixed at its

linearization point ($\rightarrow$ denotes a partial function), and $cp : \mathbb{Q} \rightarrow \{A_1, A_2, R_1, R_2, R_3\}$ records the control point of every enqueue (A_1, A_2) or dequeue operation (R_1, R_2, R_3). All the components are $\emptyset$ in the initial state, and the transition relation $\rightarrow$ is defined in Fig. 2.3. The alphabet of $AbsQ$ contains call/return actions and dequeue linearization points, denoted by $lin(deq, d, k)$. $Lin(deq)$ is the set of all actions $lin(deq, d, k)$.

Concerning enqueue operations, the rule CALL-ENQ orders the invoked operation after all the completed enqueues in the current state, and the rules RET-ENQ1/RET-ENQ2 flip the corresponding flag from PEND to COMP provided that the operation is still present in the current state. For dequeue operations, CALL-DEQ only increments the control point and RET-DEQ checks whether the return value is the same as the one fixed at the linearization point. The linearization point rule LIN-DEQ1 corresponds to the case of a non-empty queue, showing that $lin(deq, d, k)$ is enabled only if d has been added by an enqueue which is minimal in the current happens-before. When enabled, it removes the enqueue adding d from the state. The linearization point rule LIN-DEQ2 corresponds to the case of dequeue operations linearized with an EMPTY return value.

The following result states that the library $AbsQ$ has exactly the same set of histories as the standard abstract library $AbsQ_0$ (see Appendix C for a proof).

Theorem 3. *$AbsQ$ is a refinement of $AbsQ_0$ and vice-versa.*

A trace of a queue implementation is called *Lin(deq)-complete* when every completed dequeue has a linearization point, i.e., each return action $ret(deq, d, k)$ is preceded by an action $lin(deq, d, k)$. A queue implementation L over alphabet Σ , such that $C \cup R \cup Lin(deq) \subseteq \Sigma$, is called *with fixed dequeue linearization points* when every trace $\tau \in Tr(L)$ is *Lin(deq)-complete*.

The following result shows that $C \cup R \cup Lin(deq)$ -forward simulations are a sound and complete proof method for showing the correctness of a queue implementation with fixed dequeue linearization points (up to the correctness of the linearization points). It is obtained from Theorem 3 and Theorem 2 using the fact that the alphabet of $AbsQ$ is exactly $C \cup R \cup Lin(deq)$ and $AbsQ$ is deterministic.

$$\begin{array}{c}
\text{CALL-ENQ} \\
\frac{k \notin \text{dom}(cp) \quad d \neq \text{EMPTY}}{O, <, \ell, rv, cp \xrightarrow{\text{inv}(\text{enq}, d, k)} O \cup \{k\}, < \cup \text{COMP}(O) \times \{k\}, \ell[k \mapsto (d, \text{PEND})], rv, cp[k \mapsto A_1]} \\
\\
\text{CALL-DEQ} \qquad \text{RET-DEQ} \\
\frac{k \notin \text{dom}(cp)}{O, <, \ell, rv, cp \xrightarrow{\text{inv}(\text{deq}, k)} O, <, \ell, rv, cp[k \mapsto R_1]} \quad \frac{cp(k) = R_2 \quad rv(k) = d}{O, <, \ell, rv, cp \xrightarrow{\text{ret}(\text{deq}, d, k)} O, <, \ell, rv, cp[k \mapsto R_3]} \\
\\
\text{RET-ENQ1} \qquad \text{RET-ENQ2} \\
\frac{cp(k) = A_1 \quad k \in O \quad \ell(k) = (d, \text{PEND})}{O, <, \ell, rv, cp \xrightarrow{\text{ret}(\text{enq}, k)} O, <, \ell[k \mapsto (d, \text{COMP})], rv, cp[k \mapsto A_2]} \quad \frac{cp(k) = A_1 \quad k \notin O}{O, <, \ell, rv, cp \xrightarrow{\text{ret}(\text{enq}, k)} O, <, \ell, rv, cp[k \mapsto A_2]} \\
\\
\text{LIN-DEQ1} \\
\frac{cp(k) = R_1 \quad d \neq \text{EMPTY} \quad k' \in \min(O) \quad \ell_1(k') = d}{O, <, \ell, rv, cp \xrightarrow{\text{lin}(\text{deq}, d, k)} O \setminus \{k'\}, <\uparrow k', \ell, rv[k \mapsto d], cp[k \mapsto R_2]} \\
\\
\text{LIN-DEQ2} \\
\frac{cp(k) = R_1 \quad \forall o \in O. \ell_2(o) = \text{PEND}}{O, <, \ell, rv, cp \xrightarrow{\text{lin}(\text{deq}, \text{EMPTY}, k)} O, <, \ell, rv[k \mapsto \text{EMPTY}], cp[k \mapsto R_2]}
\end{array}$$

Figure 2.3: The transition relation of AbsQ . We use the following notations: $\ell_i(k)$ denotes the projection of $\ell(k)$ over the i -th component, for each $i \in \{1, 2\}$, $\text{COMP}(O) = \{k \in O : \ell_2(k) = \text{COMP}\}$, $f[x \mapsto y]$ is the function g such that $g(z) = f(z)$ for all $z \neq x$ in the domain of f , and $g(x) = y$, $\min(O)$ is the set of elements of O which are minimal in the order relation $<$, and $<\uparrow k$ denotes the relation $<$ where all the pairs containing k have been removed.

Corollary 1. *A queue implementation L with fixed dequeue linearization points is a $C \cup R \cup \text{Lin}(\text{deq})$ -refinement of AbsQ_0 iff there exists a $C \cup R \cup \text{Lin}(\text{deq})$ -forward simulation from L to AbsQ .*

2.4.3 A Correctness Proof For Herlihy&Wing Queue

We describe a forward simulation fs_1 from HWQ to AbsQ . A HWQ state is related by fs_1 to an AbsQ state that consists of all the enqueue operations for which the input is still present in the array **items** and all the pending enqueue operations that have at most reserved an array position, ordered by a relation $<$ satisfying the following:

- (a) pending enqueues are maximal, i.e., for every two enqueues k and k' such that

k' is pending, we have that $k' \not\prec k$,

- (b) $<$ is consistent with the order in which positions of `items` have been reserved, i.e., for every two enqueues k and k' such that $\mathbf{i}(k) < \mathbf{i}(k')$, we have that $k' \not\prec k$,
- (c) an enqueue which has reserved a position i can't be ordered before another enqueue that has reserved a position $j \geq i$ when the position i has been “observed” by a non-linearized dequeue that may “observe” j in the current array traversal, i.e., for every two enqueues k and k' , and a dequeue k_d , such that

$$\mathbf{x}(k_d) = \text{null} \wedge \mathbf{i}(k') \leq \text{range}(k_d) \wedge \mathbf{i}(k) \leq \mathbf{i}(k_d) \leq \mathbf{i}(k') \wedge (\mathbf{i}(k) = \mathbf{i}(k_d) \Rightarrow k_d @ \text{if-inc}) \quad (2.1)$$

we have that $k \not\prec k'$. The predicate $k_d @ \text{if-inc}$ holds when the dequeue k_d is at a control point after a `swap` returning `null` and before the increment of `i`.

An enqueue is labeled by (d, PEND) where d is the input value if it's pending and by (d, COMP) , otherwise. Also, for every dequeue operation k such that $\mathbf{x}(k) = d \neq \text{null}$, we have that $rv(k) = d$.

We show that fs_1 is indeed a $C \cup R \cup \text{Lin}(deg)$ -forward simulation. Let s and t be states of HWQ and $AbsQ$, respectively, such that $(s, t) \in fs_1$. We omit discussing the trivial case of transitions labeled by `call` and `return` actions which are simulated by similar transitions of $AbsQ$ (for the return a dequeue operation k , we use the equality between the local variable $\mathbf{x}(k)$ in s and the component $rv(k)$ in t).

We show that each internal step of an enqueue or dequeue, except the execution of `swap` returning a non-null value in dequeue (which represents its linearization point), is simulated by an *empty* sequence of $AbsQ$ transitions, i.e., for every state s' obtained through one of these steps, if $(s, t) \in fs_1$, then $(s', t) \in fs_1$ for each $AbsQ$ state t . Essentially, this consists in proving the following property, called *monotonicity*: the set of possible orders $<$ associated by fs_1 to s' doesn't exclude any order $<$ associated to s .

Concerning enqueues, let s' be the state obtained from s when a pending enqueue k reserves an array position. This enqueue must be maximal in both t and any state

t' related to s' (since it's pending). Moreover, there is no dequeue that can “observe” this position before restarting the array traversal. Therefore, item (c) in the definition of $<$ doesn't constrain the order between k and some other enqueue neither in s nor in s' . Since this transition doesn't affect the constraints on the order between enqueues different from k (their local variables remain unchanged), monotonicity holds. This property is trivially satisfied by the second step of enqueue which doesn't affect i .

To prove monotonicity in the case of dequeue internal steps different from its linearization point, it is important to track the non-trivial instantiations of item (c) in the definition of $<$ over the two states s and s' , i.e., the triples (k, k', k_d) for which (2.1) holds. Instantiations that are enabled only in s' may in principle lead to a violation of monotonicity (since they restrict the orders $<$ associated to s'). For the two steps that begin an array traversal, i.e., reading the index of the last used position and setting i to 0, there exist no such new instantiations in s' because the value of i is either not set or 0. The same is true for the increment of i in a dequeue k_d since the predicate $k_d@if-inc$ holds in state s . The execution of **swap** returning **null** in a dequeue k_d enables new instantiations (k, k', k_d) in s' , thus adding potentially new constraints $k \not\leq k'$. We show that these instantiations are however vacuous because k must be pending in s and thus maximal in every order $<$ associated by fs_1 to s . Let k and k' be two enqueues such that together with the dequeue k_d they satisfy the property (2.1) in s' but not in s . We write $i_s(k)$ for the value of the variable i of operation k in state s . We have that $i_{s'}(k) = i_{s'}(k_d) \leq i_{s'}(k')$ and $items[i_{s'}(k_d)] = \mathbf{null}$. The latter implies that the enqueue k didn't executed the second statement (since the position it reserved is still **null**) and it is pending in s . The step that checks that the value returned by **swap** is **null** doesn't modify the variables in property (2.1) and also, it doesn't change the valuation of the predicate $@if-inc$.

Finally, we show that the linearization point of a dequeue k of HWQ , i.e., an execution of **swap** returning a non-null value d , from state s and leading to a state s' is simulated by a transition labeled by $lin(deg, d, k)$ of $AbsQ$ from state t . By the

definition of HWQ , there is a unique enqueue k_e which filled the position updated by k , i.e., $i_s(k_e) = i_s(k)$ and $x_{s'}(k) = x_s(k_e)$. We show that k_e is minimal in the order $<$ of t which implies that $lin(deq, d, k)$ is enabled in t . Thus, instantiating item (c) in the definition of $<$ with $k' = k_e$ and $k_d = k$ we get that every enqueue that reserved a position smaller than the one of k_e can't be ordered before k_e in the order $<$. Also, applying item (b) with $k = k_e$ we get the same for every enqueue that reserved a bigger position. An enqueue that didn't reserved a position is by definition maximal in $<$ and therefore, not a predecessor of k_e . Then, the state t' obtained from t through a $lin(deq, d, k)$ transition is related to s' because (1) the value added by k_e is not anymore present in `items` which implies that k_e doesn't occur in any $AbsQ$ state related to s' , and (2) the value of $x(k)$ is set to $d \neq \text{null}$ which implies that $rv(k)$ is set to d in every $AbsQ$ state related to s' .

2.5 Stacks With Fixed Pop Commit Points

While the abstract queue in Section 2.4 can be adapted to stacks (the linearization point $lin(pop, d, k)$ with $d \neq \text{EMPTY}$ is enabled when k is added by a push which is maximal in the happens-before order stored in the state), it can't simulate (through forward simulations) existing stack implementations like the Time-Stamped Stack [24] (*TSS*, for short) where the linearization points of the pop operations are not fixed. Exploiting particular properties of the stack semantics, we refine the ideas used in $AbsQ$ and define a new abstract implementation for stacks, denoted as $AbsS$, which is able to simulate such implementations. Forward simulations to $AbsS$ are complete for proving the correctness of stack implementations provided that the point in time where the return value of a pop operation is determined, called *commit point*, corresponds to a fixed action.

2.5.1 Pop Methods With Fixed Commit Points

We explain the meaning of the commit points on a simplified version of the Time-Stamped Stack [24] (*TSS*, for short) given in Figure 2.4. This implementation main-

```

struct Node{
    int data;
    int ts;
    Node* next;
    boolean taken;
};
Node* pools[maxThreads];
int TS = 0;

void push(int x) {
    Node* n = new Node(x, MAXINT,
                        null, false);
    n->next = pools[myTID];
    pools[myTID] = n;
    int i = TS++;
    n->ts = i;
}

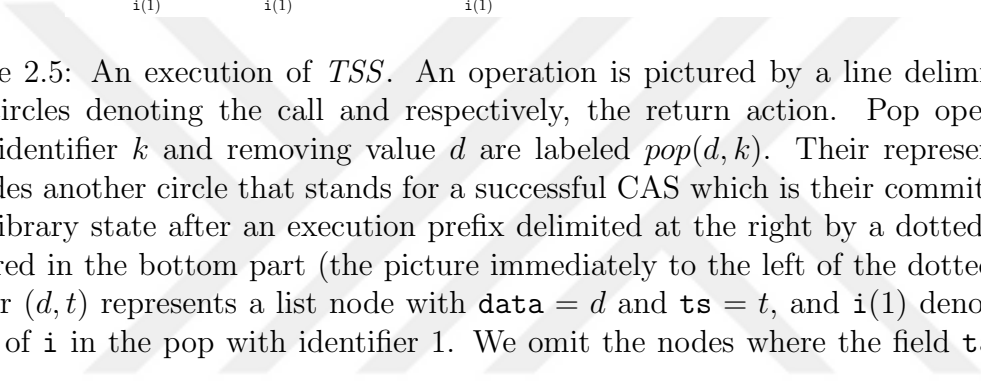
int pop() {
    boolean success = false;
    int maxTS = 1;
    Node* youngest = null;
    while ( !success ) {
        maxTS = 1; youngest = null;
        for(int i=0; i<maxThreads; i++){
            Node* n = pools[i];
            while (n->taken && n->next != n)
                n = n->next;
            if(maxTS < n->ts) {
                maxTS = n->ts; youngest = n;
            }
        }
        if (youngest != null)
            success=CAS(youngest->taken,
                        false, true);
    }
    return youngest->data;
}

```

Figure 2.4: The Time-Stamped Stack. We assume that every statement is atomic.

tains an array of singly-linked lists, one for each thread, where list nodes contain a data value (field **data**), a timestamp (field **ts**), the next pointer (field **next**), and a boolean flag indicating whether the node represents a value removed from the stack (field **taken**). Initially, each list contains a sentinel dummy node pointing to itself with timestamp -1 and the flag **taken** set to **false**.

Pushing a value to the stack proceeds in several steps: adding a node with maximal timestamp in the list associated to the thread executing the push (given by the special variable **myTID**), asking for a new timestamp (given by the shared variable **TS**), and updating the timestamp of the added node. Popping a value from the stack consists in traversing all the lists, finding the first element which doesn't represent a removed value (i.e., **taken** is **false**) in each list, and selecting the element with the maximal timestamp. A compare-and-swap (CAS) is used to set the **taken** flag of this element to **true**. The procedure restarts if the CAS fails.



The push operations don't have a fixed linearization point because adding a node to a list and updating its timestamp are not executed in a single atomic step. The nodes can be added in an order which is not consistent with the order between the timestamps assigned later in the execution. Also, the value added by a push that just added an element to a list can be popped before the value added by a completed push (since it has a maximal timestamp). The same holds for pop operations: The only reasonable choice for a linearization point is a successful CAS (that results in updating the field `taken`). Fig. 2.5 pictures an execution showing that this action doesn't correspond to a linearization point, i.e., an execution for which the pop operations in every correct linearization are not ordered according to the order between successful CASs. In every correct linearization of that execution, the pop operation removing x is ordered before the one removing z although they perform a successful CAS in the opposite order.

An interesting property of the successful CASs in pop operations is that they fix the return value, i.e., the return value is `youngest->data` where `youngest` is the node

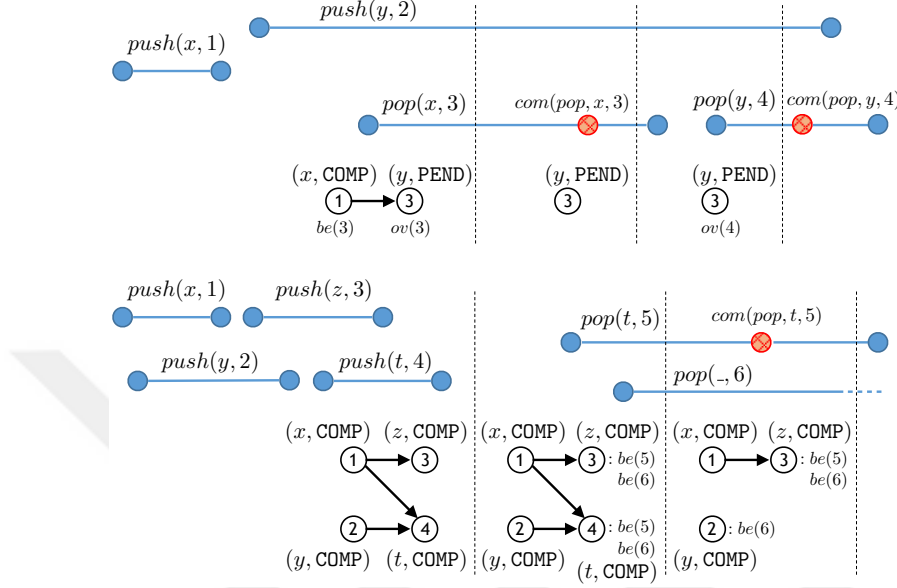
updated by the CAS. We call such actions *commit points*. More generally, commit points are actions that access shared variables, from which every control-flow path leads to the return control point and contains no more accesses to the shared memory (i.e., after a commit point, the return value is computed using only local variables).

When the commit points of pop operations are fixed to particular implementation actions (e.g., a successful CAS) we assume that the library is an LTS over an alphabet that contains actions $com(pop, d, k)$ with $d \in \mathbb{V}$ and $k \in \mathbb{O}$ (denoting the commit point of the pop with identifier k and returning d). Let $Com(pop)$ be the set of such actions.

2.5.2 Abstract Stack Implementation

We define an abstract stack $AbsS$ over alphabet $C \cup R \cup Com(pop)$ that essentially, similarly to $AbsQ$, maintains the happens-before order of the pushes whose value has not been yet removed. Pops are treated differently since the commit points are not necessarily linearization points, intuitively, a pop can be linearized before its commit. Each pop operation starts by taking a snapshot of the greatest completed push operations in the happens-before order, and continuously tracks the push operations which are overlapping with it. The commit point $com(pop, d, k)$ with $d \neq \text{EMPTY}$ is enabled only if d was added by one of the push operations in the initial snapshot, or by a push happening earlier when all the values from the initial snapshot have been removed, or by one of the push operations that overlaps with pop k . The commit point $com(pop, \text{EMPTY}, k)$ is enabled only if all the values added by push operations ending before k started have been removed. The effect of the commit points is explained below through examples.

Figure 2.6 pictures two executions of $AbsS$ for two extended histories (that include pop commit points). For readability, we give the state of $AbsS$ only after several execution prefixes delimited at the right by a dotted line. We focus on pop operations – the effect of push calls and returns is similar to enqueue calls and returns in $AbsQ$. Let us first consider the history on the top part. The first state we give is reached after the call of pop with identifier 3. This shows the effect of a pop invocation: the

Figure 2.6: Simulating stack histories with *AbsS*.

greatest completed pushes according to the current happens-before (here, the push with identifier 1) are marked as $be(3)$ (from “before” operation 3), and the pending pushes are marked as $ov(3)$ (from “overlapping” with operation 3). As a side remark, any other push operation that starts after $pop\ 3$ would be also marked as $ov(3)$. The commit point $com(pop, x, 3)$ (pictured with a red circle) is enabled because x was added by a push marked as $be(3)$. The effect of the commit point is that push 1 is removed from the state (the execution on the bottom shows a more complicated case). For the second pop, the commit point $com(pop, y, 4)$ is enabled because y was added by a push marked as $ov(4)$. The execution on the bottom shows an example where the marking $be(k)$ for some pop k is updated at commit points. The pushes 3 and 4 are marked as $be(5)$ and $be(6)$ when the pops 5 and 6 start. Then, $com(pop, t, 5)$ is enabled since t was added by $push(t, 4)$ which is marked as $be(5)$. Besides removing $push(t, 4)$, the commit point produces a state where a pop committing later, e.g., pop 6, can remove y which was added by a predecessor of $push(t, 4)$ in the happens-before (y could become the top of the stack when t is removed). This history is valid because $push(y, 2)$ can be linearized after $push(x, 1)$ and $push(z, 3)$. Thus, push 2, a

predecessor of the push which is removed, is marked as $be(6)$. Push 1 which is also a predecessor of the removed push is not marked as $be(6)$ because it happens before another push, i.e., push 3, which is already marked as $be(6)$ (the value added by push 3 should be removed before the value added by push 1 could become the top of the stack).

Formally, the states of $AbsS$ are tuples $\langle O, <, \ell, rv, cp, be, ov \rangle$ where $<$ is a strict partial order over the set O of operation identifiers, $\ell : O \rightarrow \mathbb{V} \times \{\text{PEND}, \text{COMP}\}$ labels every identifier in O with a value and a pending/completed flag, $rv : \mathbb{O} \rightarrow \mathbb{V}$ records the return value of a pending pop fixed at its commit point, $cp : \mathbb{O} \rightarrow \{A_1, A_2, R_1, R_2, R_3\}$ records the control point of every push (A_1, A_2) or pop operation (R_1, R_2, R_3) , $be : \mathbb{O} \rightarrow 2^O$ records the greatest completed push operations before a pop started or happening earlier provided that the values of all the push happening later have been removed, and $ov : \mathbb{O} \rightarrow 2^O$ records push operations overlapping with a pop. All the components are $\emptyset$ in the initial state, and the transition relation $\rightarrow$ is defined in Fig. 2.7.

The transition rules which don't correspond to commit point actions are similar to those for $AbsQ$. The rule COM-POP1 for $com(pop, d, k)$ is enabled only if there exists a push k' which added value d and which belongs to $be(k)$ or $ov(k)$. When enabled, the push k' is removed from the set O (and the order $<$) and for every other pop k_1 such that k' belongs to $be(k_1)$, k' is replaced in $be(k_1)$ by its predecessors which are followed exclusively by pushes overlapping with k_1 (these predecessors become maximal closed pushes once k' is removed). Also, $rv(k)$ is set to d . The rule COM-POP1 for $com(pop, \text{EMPTY}, k)$ is enabled only if $be(k)$ is empty (i.e., all the values added by pushes ending before k , if any, have been removed). Then, $rv(k)$ is set to **EMPTY**.

Let $AbsS_0$ be the standard abstract implementation of a stack (where elements are stored in a sequence; push, resp., pop operations add, resp., remove, an element from the beginning of the sequence in one atomic step). For $\mathbb{M} = \{push, pop\}$, the alphabet of $AbsS_0$ is $C \cup R \cup Lin$. The following result states that the library $AbsS$ has exactly the same set of histories as $AbsS_0$ (see Appendix D for a proof).

Theorem 4. *$AbsS$ is a refinement of $AbsS_0$ and vice-versa.*

$$\begin{array}{c}
\text{CALL-PUSH} \\
\hline
\frac{k \notin \text{dom}(cp) \quad d \neq \text{EMPTY} \quad \forall k'. \text{ov}'(k') = \text{ov}(k') \cup \{k\}}{O, <, \ell, rv, cp, be, ov \xrightarrow{\text{inv}(\text{push}, d, k)} O \cup \{k\}, < \cup \text{COMP}(O) \times \{k\}, \ell[k \mapsto (d, \text{PEND})], rv, cp[k \mapsto A_1], be, \text{ov}'} \\
\\
\text{CALL-POP} \\
\hline
\frac{k \notin \text{dom}(cp)}{O, <, \ell, rv, cp, be, ov \xrightarrow{\text{inv}(\text{pop}, k)} O, <, \ell, rv, cp[k \mapsto R_1], be[k \mapsto \text{maxCo}(O)], \text{ov}[k \mapsto \text{PEND}(O)]} \\
\\
\text{RET-POP} \\
\hline
\frac{cp(k) = R_2 \quad rv(k) = d}{O, <, \ell, rv, cp, be, ov \xrightarrow{\text{ret}(\text{pop}, d, k)} O, <, \ell, rv, cp[k \mapsto R_3], be, \text{ov}} \\
\\
\text{RET-PUSH1} \\
\hline
\frac{cp(k) = A_1 \quad k \in O \quad \ell(k) = (d, \text{PEND})}{O, <, \ell, rv, cp, be, ov \xrightarrow{\text{ret}(\text{push}, k)} O, <, \ell[k \mapsto (d, \text{COMP})], rv, cp[k \mapsto A_2], be, \text{ov}} \\
\\
\text{RET-PUSH2} \\
\hline
\frac{cp(k) = A_1 \quad k \notin O}{O, <, \ell, rv, cp, be, ov \xrightarrow{\text{ret}(\text{push}, k)} O, <, \ell, rv, cp[k \mapsto A_2], be, \text{ov}} \\
\\
\text{COM-POP1} \\
\hline
\frac{\begin{array}{l} cp(k) = R_1 \quad d \neq \text{EMPTY} \quad k' \in be(k) \cup \text{ov}(k) \quad \ell_1(k') = d \quad \forall k_1. k' \notin be(k_1) \Rightarrow be'(k_1) = be(k_1) \\ \forall k_1. k' \in be(k_1) \Rightarrow be'(k_1) = (be(k_1) \setminus \{k'\}) \cup \{k_2 : k_2 \in \text{pred}_<(k') \wedge \forall k_3. (k_2 \in \text{pred}_<(k_3) \wedge k_3 \neq k') \Rightarrow k_3 \in \text{ov}(k_1) \} \end{array}}{O, <, \ell, rv, cp, be, ov \xrightarrow{\text{com}(\text{pop}, d, k)} O \setminus \{k'\}, < \uparrow k', \ell, rv[k \mapsto d], cp[k \mapsto R_2], be', \text{ov}} \\
\\
\text{COM-POP2} \\
\hline
\frac{cp(k) = R_1 \quad be(k) = \emptyset}{O, <, \ell, rv, cp, be, ov \xrightarrow{\text{com}(\text{pop}, \text{EMPTY}, k)} O, <, \ell, rv[k \mapsto \text{EMPTY}], cp[k \mapsto R_2], be, \text{ov}}
\end{array}$$

Figure 2.7: The transition relation of $\text{Abs}S$. We use the following notions: $\text{maxCo}(O)$ is the set of greatest operations in O (w.r.t. $<$) which are completed, i.e., $\text{maxCo}(O) = \{k \in O : \ell_2(k) = \text{COMP}, \forall k' \in O. k' < k \vee \ell_2(k') = \text{PEND}\}$, $\text{PEND}(O) = \{k \in O : \ell_2(k) = \text{PEND}\}$, and $\text{pred}_<(k')$ is the set of immediate predecessors of k' according to $<$, i.e., $\text{pred}_<(k') = \{k \in O : k < k' \wedge \forall k'' \in O. k'' > k' \vee k'' < k\}$.

A trace of a stack implementation is called *Com(pop)-complete* when every completed pop has a commit point, i.e., each return $\text{ret}(\text{pop}, d, k)$ is preceded by an action $\text{com}(\text{pop}, d, k)$. A stack implementation L over Σ , such that $C \cup R \cup \text{Com}(\text{pop}) \subseteq \Sigma$, is called *with fixed pop commit points* when every trace $\tau \in \text{Tr}(L)$ is *Com(pop)-complete*.

As a consequence of Theorem 2, $C \cup R \cup \text{Com}(\text{pop})$ -forward simulations are a sound and complete proof method for showing the correctness of a stack implementation with

fixed pop commit points (up to the correctness of the commit points).

Corollary 2. *A stack L with fixed pop commit points is a $C \cup R \cup Com(pop)$ -refinement of $AbsS$ iff there is a $C \cup R \cup Com(pop)$ -forward simulation from L to $AbsS$.*

Linearization points can also be seen as commit points and thus the following holds.

Corollary 3. *A stack implementation L with fixed pop linearization points where transition labels $lin(pop, d, k)$ are substituted with $com(pop, d, k)$ is a $C \cup R \cup Com(pop)$ -refinement of $AbsS_0$ iff there is a $C \cup R \cup Com(pop)$ -forward simulation from L to $AbsS$.*

2.5.3 A Correctness Proof For Time-Stamped Stack

We describe a forward simulation fs_2 from TSS to $AbsS$. Except for the constraints on the components be and ov of a $AbsS$ state, it is similar to the simulation fs_1 from HWQ to $AbsQ$. Thus, the $AbsS$ states $t = \langle O, <, \ell, rv, cp, be, ov \rangle$ associated by fs_2 to a TSS state s satisfy the following. The set O consists of all the identifiers of pushes in s which didn't added yet a node to `pools` or for which the input is still present in `pools` (i.e., the node created by the push has `taken` set to `false`). A push k is labeled by $(d, PEND)$ where d is the input value if it's pending and by $(d, COMP)$, otherwise.

To describe the order relation $<$ we consider the following notations: $\mathbf{ts}_s(k)$, resp., $\mathbf{TID}_s(k)$, denotes the timestamp of the node created by the push k in state s (the `ts` field of this node), resp., the id of the thread executing k . By an abuse of terminology, we call $\mathbf{ts}_s(k)$ the timestamp of k in state s . Also, $k \rightsquigarrow_s k'$ when intuitively, a traversal of `pools` would encounter the node created by k before the one created by k' . More precisely, $k \rightsquigarrow_s k'$ when $\mathbf{TID}_s(k) < \mathbf{TID}_s(k')$, or $\mathbf{TID}_s(k) = \mathbf{TID}_s(k')$ and the node created by k' is reachable from the one created by k in the list pointed to by `pools`[$\mathbf{TID}_s(k)$]. The order relation $<$ satisfies the following: (1) pending pushes are maximal, (2) $<$ is consistent with the order between node timestamps, i.e., $\mathbf{ts}_s(k) \leq \mathbf{ts}_s(k')$ implies $k' \not< k$, and (3) $<$ includes the order between pushes executed in the same thread, i.e., $\mathbf{TID}_s(k) = \mathbf{TID}_s(k')$ and $\mathbf{ts}_s(k) < \mathbf{ts}_s(k')$ implies $k < k'$.

The components *be* and *ov* satisfy the following constraints (their domain is the set of identifiers of pending pops):

- a pop k with **youngest** $\neq$ **null** that reached a node with timestamp τ (its variable **n** points to this node) overlaps with every push that created a node with a timestamp bigger than τ and which occurs in **pools** before the node reached by k , i.e., **youngest** _{s} (k) $\neq$ **null**, **n** _{s} (k) = **n** _{s} (k_1), $k_2 \rightsquigarrow_s k_1$, **n** _{s} (k_2)->**taken** = **false**, and **ts** _{s} (k_2) $\geq$ **ts** _{s} (k_1) implies $k_2 \in ov(k)$, for each k, k_1, k_2
- a pop k with **youngest** = **null** overlaps with every push that created a node which occurs in **pools** before the node reached by k , i.e., **youngest** _{s} (k) = **null**, **n** _{s} (k) = **n** _{s} (k_1), $k_2 \rightsquigarrow_s k_1$, and **n** _{s} (k_2)->**taken** = **false** implies $k_2 \in ov(k)$, for each k, k_1, k_2
- if the variable **youngest** of a pop k points to a node which is not taken, then this node was created by a push in $be(k) \cup ov(k)$ or the node currently reached by k is followed in **pools** by another node which was created by a push in $be(k) \cup ov(k)$, i.e., **youngest** _{s} (k) = **n** _{s} (k_1), **n** _{s} (k_1)->**taken** = **false**, and **n** _{s} (k) = **n** _{s} (k_2) implies $k_1 \in be(k) \cup ov(k)$ or that there exists $k_3 \in O$ such that **ts** _{s} (k_3) $>$ **ts** _{s} (k_1), $k_3 \in be(k) \cup ov(k)$, and either $k_2 \rightsquigarrow_s k_3$ or **n** _{s} (k_2) = **n** _{s} (k_3) and k is traversing the last list in the array **pools**, for each k, k_1, k_2

There are some more constraints on *be* and *ov* that can be seen as invariants of *AbsS*, i.e., $be(k)$ and $ov(k)$ don't contain predecessors of pushes from $be(k)$ (for each k, k_1, k_2 , $k_1 < k_2$ and $k_2 \in be(k)$ implies $k_1 \notin be(k) \cup ov(k)$). They can be found in Appendix E.

Finally, for every pop operation k such that **success**(k) = **true**, we have that $rv(k) = \text{youngest}(k)\text{->data}$.

The proof that fs_2 is indeed a forward simulation from *TSS* to *AbsS* follows the same lines as the one given for the Herlihy&Wing Queue. It can be found in Appendix E.

2.6 Related Work

Many techniques for linearizability verification, e.g., [64, 10, 63, 4], are based on forward simulation arguments, and typically only work for libraries where the linearization point of every invocation of a method m is fixed to a particular statement in the code of m . The works in [62, 23, 25, 65] deal with *external* linearization points where the action of an operation k can be the linearization point of a concurrently executing operation k' . We say that the linearization point of k' is external. This situation arises in read-only methods like the **contains** method of an optimistic set [54], libraries based on the elimination back-off scheme, e.g., [33], or flat combining [32, 29]. In these implementations, an operation can do an update on the shared state that becomes the linearization point of a concurrent read-only method (e.g., a **contains** returning **true** may be linearized when an **add** method adds a new value to the shared state) or an operation may update the data structure on behalf of other concurrently executing operations (whose updates are published in the shared state). In all these cases, every linearization point can still be associated syntactically to a statement in the code of a method and doesn't depend on operations executed in the future (unlike *HWQ* and *TSS*). However, identifying the set of operations for which such a statement is a linearization point can only be done by looking at the whole program state (the local states of all the active operations). This poses a problem in the context of compositional reasoning (where auxiliary variables are required), but still admits a forward simulation argument. For manual proofs, such implementations with external linearization points can still be defined as LTSs that produce *Lin*-complete traces and thus still fall in the class of implementations for which forward simulations are enough for proving refinement. These proof methods are not complete and they are not able to deal with implementations like *HWQ* or *TSS*.

There also exist linearizability proof techniques based on backward simulations or alternatively, prophecy variables, e.g., [57, 43]. These works can deal with implementations where the linearization points are not fixed, but the proofs are conceptually more complex and less amenable to automation.

The works in [34, 15] propose reductions of linearizability to assertion checking where the idea is to define finite-state automata that recognize violations of concurrent queues and stacks. These automata are simple enough in the case of queues and there is a proof of *HWQ* based on this reduction [34]. However, in the case of stacks, the automata become much more complicated and we are not aware of a proof for an implementation such as *TSS* which is based on this reduction.



A Libraries

Programs interact with libraries by calling named library *methods*, which receive *parameter values* and yield *return values* upon completion. We fix arbitrary sets $\mathbb{M}$ and $\mathbb{V}$ of method names and parameter/return values.

We fix an arbitrary set $\mathbb{O}$ of operation identifiers, and for given sets $\mathbb{M}$ and $\mathbb{V}$ of methods and values, we fix the sets

$$C = \{inv(m, d, k) : m \in \mathbb{M}, d \in \mathbb{V}, k \in \mathbb{O}\}, \text{ and}$$

$$R = \{ret(m, d, k) : m \in \mathbb{M}, d \in \mathbb{V}, k \in \mathbb{O}\}$$

of *call actions* and *return actions*; each call action $inv(m, d, k)$ combines a method $m \in \mathbb{M}$ and value $d \in \mathbb{V}$ with an *operation identifier* $k \in \mathbb{O}$. Operation identifiers are used to pair call and return actions. We assume every set of words is closed under isomorphic renaming of operation identifiers. We denote the operation identifier of a call/return action a by $op(a)$. Call and return actions $c \in C$ and $r \in R$ are *matching*, written $c \vdash r$, when $op(c) = op(r)$. We may omit the second field from a call/return action a for methods that have no inputs (e.g., the pop method of a stack) or return values (e.g., the push method of a stack). A word $\tau \in \Sigma^*$ over alphabet Σ , such that $(C \cup R) \subseteq \Sigma$, is *well formed* when:

- Each return is preceded by a matching call:
 $\tau_j \in R$ implies $\tau_i \vdash \tau_j$ for some $i < j$.
- Each operation identifier is used in at most one call/return:
 $op(\tau_i) = op(\tau_j)$ and $i < j$ implies $\tau_i \vdash \tau_j$.

We say that the well-formed word $\tau \in \Sigma^*$ is *sequential* when

- Operations do not overlap:
 $\tau_i, \tau_k \in C$ and $i < k$ implies $\tau_i \vdash \tau_j$ for some $i < j < k$.

Well-formed words represent traces of a library. We assume every set of well-formed words is closed under isomorphic renaming of operation identifiers. For notational

convenience, we take $\mathbb{O} = \mathbb{N}$ for the rest of this chapter. When the value of a certain field in a call/return action is not important we use the placeholder $_$, e.g., $inv(m, _, k)$ instead of $inv(m, d, k)$ when the input d can take any value.

An operation k is called *completed* in a well-formed trace τ when $ret(m, d, k)$ occurs in τ , for some m and d . Otherwise, it is called *pending*.

Libraries dictate the execution of methods between their call and return points. Accordingly, a library cannot prevent a method from being called, though it can decide not to return. Furthermore, any library action performed in the interval between call and return points can also be performed should the call have been made earlier, and/or the return made later. A library thus allows any sequence of invocations to its methods made by *some* program.

Definition 4. A library L is an LTS over alphabet Σ such that $C \cup R \subseteq \Sigma$ and each trace $\tau \in Tr(L)$ is well formed, and

- Call actions $c \in C$ cannot be disabled:
 $\tau \cdot \tau' \in Tr(L)$ implies $\tau \cdot c \cdot \tau' \in Tr(L)$ if $\tau \cdot c \cdot \tau'$ is well formed.
- Call actions $c \in C$ cannot disable other actions:
 $\tau \cdot a \cdot c \cdot \tau' \in Tr(L)$ implies $\tau \cdot c \cdot a \cdot \tau' \in Tr(L)$.
- Return actions $r \in R$ cannot enable other actions:
 $\tau \cdot r \cdot a \cdot \tau' \in Tr(L)$ implies $\tau \cdot a \cdot r \cdot \tau' \in Tr(L)$.

Note that even a library that implements *atomic methods*, e.g., by guarding method bodies with a global-lock acquisition, admits executions in which method calls and returns overlap. For simplicity, Definition 4 assumes that every thread performs a single operation. The extension to multiple operations per thread is straightforward, e.g. the closure rules must assume that the actions a and c belong to different threads.

B Normal Forward/Backward Simulations

We define a class of forward/backward simulations, called *normal simulations*, that are used in the proofs in Appendix C and Appendix D.

Definition 5. Let $L_1 = (Q_1, \Sigma, s_0^1, \delta_1)$ and $L_2 = (Q_2, \Sigma, s_0^2, \delta_2)$ be two libraries over alphabets Σ_1 and Σ_2 , respectively, such that $C \cup R \subseteq \Sigma_1 \cap \Sigma_2$, and Γ a set of actions such that $C \cup R \subseteq \Gamma \subseteq \Sigma_1 \cap \Sigma_2$. A relation $fs \subseteq Q_1 \times Q_2$ is called a normal Γ -forward simulation from L_1 to L_2 iff the following holds:

- (i) $fs[s_0^1] = \{s_0^2\}$
- (ii-a) If $(s, c, s') \in \delta_1$, for some $c \in C$, and $u \in fs[s]$, then there exists $u' \in fs[s']$ such that $u \xrightarrow{\sigma} u'$, $\sigma_0 = c$, and $\sigma_i \in \Sigma_2 \setminus \Gamma$, for each $0 < i < |\sigma|$.
- (ii-b) If $(s, r, s') \in \delta_1$, for some $r \in R$, and $u \in fs[s]$, then there exists $u' \in fs[s']$ such that $u \xrightarrow{\sigma} u'$, $\sigma_{|\sigma|-1} = r$, and $\sigma_i \in \Sigma_2 \setminus \Gamma$, for each $0 \leq i < |\sigma| - 1$.
- (ii-c) If $(s, \gamma, s') \in \delta_1$, for some $\gamma \in \Gamma \setminus (C \cup R)$, and $u \in fs[s]$, then there exists $u' \in fs[s']$ such that $\delta_2(u, \gamma, u')$.
- (ii-d) If $(s, e, s') \in \delta_1$, for some $e \in \Sigma_1 \setminus \Gamma$ and $u \in fs[s]$, then there exists $u' \in fs[s']$ such that $u \xrightarrow{\sigma} u'$ and $\sigma \in (\Sigma_2 \setminus \Gamma)^*$.

With normal Γ -forward simulations, a step of L_1 labeled by a call, resp., return, action is simulated by a sequence of steps of L_2 that start, resp., end, with the same action, and a step of L_1 labeled by another observable action should be matched by a step of L_2 labeled by the same action. The rest of the transitions in L_1 are matched to a possibly empty sequence of transitions of L_2 with arbitrary labels.

A dual notion of forward simulation is the backward simulation:

Definition 6. Let $L_1 = (Q_1, \Sigma, s_0^1, \delta_1)$ and $L_2 = (Q_2, \Sigma, s_0^2, \delta_2)$ be two libraries over a common alphabet Σ , and $\Gamma \subseteq \Sigma$ a set of actions such that $(C \cup R) \subseteq \Gamma$. A relation

$bs \subseteq Q_1 \times Q_2$ is called a normal Γ -backward simulation from L_1 to L_2 iff the following holds:

$$(i) \ bs[s_0^1] = \{s_0^2\}$$

(ii-a) If $(s, c, s') \in \delta_1$, for some $c \in C$, and $u' \in bs[s']$, then there exists $u \in bs[s]$ such that $u \xrightarrow{\sigma} u'$, $\sigma_0 = c$, and $\sigma_i \in \Sigma \setminus \Gamma$, for each $0 < i < |\sigma|$.

(ii-b) If $(s, r, s') \in \delta_1$, for some $r \in R$, and $u' \in bs[s']$, then there exists $u \in bs[s]$ such that $u \xrightarrow{\sigma} u'$, $\sigma_{|\sigma|-1} = r$, and $\sigma_i \in \Sigma \setminus \Gamma$, for each $0 \leq i < |\sigma| - 1$.

(ii-c) If $(s, \gamma, s') \in \delta_1$, for some $\gamma \in \Gamma \setminus (C \cup R)$, and $u' \in bs[s']$, then there exists $u \in bs[s]$ such that $\delta_2(u, \gamma, u')$

(ii-d) If $(s, e, s') \in \delta_1$ for some $e \in \Sigma \setminus \Gamma$ and $u' \in bs[s']$, then there exists $u \in bs[s]$ such that $u \xrightarrow{\sigma} u'$ and $\sigma \in (\Sigma_2 \setminus \Gamma)^*$.

C Proof of Theorem 3

We show that $AbsQ$ and $AbsQ_0$ refine each other. We start by giving a formal definition of the standard reference implementation $AbsQ_0$. Thus, the states of $AbsQ_0$ are tuples $\langle \sigma, in^0, rv^0, cp^0 \rangle$ where $\sigma \in \mathbb{V}^*$ is a sequence of values, $in^0 : \mathbb{O} \rightarrow \mathbb{V}$ records the input value of an enqueue, $rv^0 : \mathbb{O} \rightarrow \mathbb{V}$ records the return value of a dequeue fixed at its linearization point ($\rightarrow$ denotes a partial function), and $cp^0 : \mathbb{O} \rightarrow \{A_1, A, A_2, R_1, R_2, R_3\}$ records the control point of every enqueue (A_1, A, A_2) or dequeue operation (R_1, R_2, R_3). All the components are $\emptyset$ in the initial state, and the transition relation $\rightarrow$ is defined in Fig. 2.8. The alphabet of $AbsQ$ contains call/return actions and enqueue/dequeue linearization points.

To prove that $AbsQ$ is a refinement of $AbsQ_0$ we define a normal $C \cup R \cup Lin(deg)$ -backward simulation (i.e, a backward simulation as in Definition 6) from $AbsQ$ to $AbsQ_0$. The reverse is shown using a normal $C \cup R \cup Lin(deg)$ -forward simulation (i.e, a forward simulation as in Definition 5).

$$\begin{array}{c}
\text{CALL-ENQ} \qquad \qquad \qquad \text{LIN-ENQ} \\
\frac{k \notin \text{dom}(cp^0) \quad d \neq \text{EMPTY}}{\sigma, in^0, rv^0, cp^0 \xrightarrow{\text{inv}(\text{enq}, d, k)} \sigma, in^0[k \mapsto d], rv^0, cp^0[k \mapsto A_1]} \quad \frac{cp^0(k) = A_1}{\sigma, in^0, rv^0, cp^0 \xrightarrow{\text{lin}(\text{enq}, d, k)} d \cdot \sigma, in^0, rv^0, cp^0[k \mapsto A]} \\
\\
\text{RET-ENQ} \\
\frac{cp^0(k) = A}{\sigma, in^0, rv^0, cp^0 \xrightarrow{\text{ret}(\text{enq}, k)} \sigma, in^0, rv^0, cp^0[k \mapsto A_2]} \\
\\
\text{CALL-DEQ} \qquad \qquad \qquad \text{LIN-DEQ1} \\
\frac{k \notin \text{dom}(cp^0)}{\sigma, in^0, rv^0, cp^0 \xrightarrow{\text{inv}(\text{deq}, k)} \sigma, in^0, rv^0, cp^0[k \mapsto R_1]} \quad \frac{cp^0(k) = R_1 \quad \sigma = \sigma' \cdot d}{\sigma, in^0, rv^0, cp^0 \xrightarrow{\text{lin}(\text{deq}, d, k)} \sigma', in^0, rv^0[k \mapsto d], cp^0[k \mapsto R_2]} \\
\\
\text{LIN-DEQ2} \\
\frac{cp^0(k) = R_1 \quad \sigma = \epsilon}{\sigma, in^0, rv^0, cp^0 \xrightarrow{\text{lin}(\text{deq}, \text{EMPTY}, k)} \sigma, in^0, rv^0[k \mapsto \text{EMPTY}], cp^0[k \mapsto R_2]} \\
\\
\text{RET-DEQ} \\
\frac{cp^0(k) = R_2 \quad rv^0(k) = d}{\sigma, in^0, rv^0, cp^0 \xrightarrow{\text{ret}(\text{deq}, d, k)} \sigma, in^0, rv^0, cp^0[k \mapsto R_3]}
\end{array}$$

Figure 2.8: The transition relation of $AbsQ_0$.

Lemma 1. $AbsQ$ is a refinement of $AbsQ_0$.

Proof. We define a normal $C \cup R \cup \text{Lin}(\text{deq})$ -backward simulation bs from $AbsQ$ to $AbsQ_0$ as follows. Given an $AbsQ$ state $s = \langle O, <, \ell, rv, cp \rangle$ and an $AbsQ_0$ state $t = \langle \sigma, in^0, rv^0, cp^0 \rangle$ we have that $(s, t) \in bs$ iff the following hold:

- the sequence σ is a linearization of a partial order $(D, \prec)$ where D contains values labeling elements of O and all the values corresponding to completed enqueues, i.e., $\ell_1(\text{COMP}(O)) \subseteq D \subseteq \ell_1(O)$ ordered according to the happens-before order between the enqueues that added them, i.e., $d_1 \prec d_2$ iff there exists k_1, k_2 such that $\ell_1(k_1) = d_1$, $\ell_1(k_2) = d_2$, and $k_1 < k_2$.
- the return values fixed at dequeue linearization points are the same, i.e., for every k , $rv(k) = rv^0(k)$,
- every dequeue is at the same control point in both s and t , i.e., for every k and

$i \in \{1, 2, 3\}$, $cp(k) = R_i$ iff $cp^0(k) = R_i$,

- every pending enqueue has the same input value in both s and t , i.e., for every k , $\ell_1(k) = in^0(k)$,
- a pending enqueue from O has been linearized whenever its value is contained in σ , i.e., for every k , $cp^0(k) = A$ if $\ell_1(k) \in D$ and $\ell_2(k) = \text{PEND}$,
- a pending enqueue from O hasn't been linearized whenever its value is not in σ , i.e., for every k , $cp^0(k) = A_1$ iff $\ell_1(k) \notin D$ and $\ell_2(k) = \text{PEND}$,
- a pending enqueue which is not in O has been linearized, i.e., for every k , $cp^0(k) = A$ if $k \notin O$ and $cp(k) = A_1$,
- an enqueue is completed in s whenever it is completed in t , i.e., for every k , $cp(k) = A_2$ iff $cp^0(k) = A_2$,

For the conditions described above, if we fix the set D and σ_t , then the state t related to s becomes unique. We use this fact in the proof. In some places, we only give D , σ_t and s without explicitly defining t or show that there exists t with the given σ_t that is related to s by just finding a D such that σ_t is a linearization of $(D, \prec)$ where $\prec$ is induced from $<_s$.

or σ_t and not describing t explicitly.

In the following, we show that indeed bs is a normal $C \cup R \cup Lin(deg)$ -backward simulation from $AbsQ$ to $AbsQ_0$.

$$\langle i \rangle \quad bs[s_0^{AbsQ}] = \{s_0^{AbsQ_0}\}.$$

CALL-ENQ Let $s \xrightarrow{inv(enq, d, k)}_{AbsQ} s'$ and $t' \in bs[s']$. Either $k \in D_{t'}$ or not.

First consider the former case. We know that $\ell_{s'}(k) = (d, \text{PEND})$ and k is maximal in s' . Hence $\sigma_{t'} = \rho \circ \langle d \rangle \circ \pi$ where π contains linearization of pending elements in $O_{s'}$. Then, pick $\sigma_t = \rho$. We can find such a $t \in bs[s]$ with σ_t . Let

$(D, \prec)$ be the partial order that is used while constructing $\sigma_{t'}$ from O_s and $<_s$. We can find $(D', \prec')$ for relating s to t such that D' does not contain the values of pending elements that formed π suffix of $s_{t'}$ and d coming from linearization of $k \in O_{s'}$.

One can also see that $t \xrightarrow{\alpha}_{AbsQ_0} t'$ where $\alpha = inv(enq, d, k), lin(enq, d, k), lin(enq, d_1, k_1), \dots, lin(enq, d_j, k_j)$ such that $\pi = d_1, \dots, d_j$ and $k_1, \dots, k_j \in O_{s'}$ are the pending elements that are linearized to form π . Note that α obeys the definition of normal backward simulation definition.

For the second case, pick t such that $\sigma_t = \sigma_{t'}$. We can find a t with σ_t related to s by bs using the same $(D, \prec)$ partial order that is used while relating s' to t' . $\ell_1(\text{COMP}(O_s)) \subseteq D$ holds because $\text{COMP}(O_s) = \text{COMP}(O_{s'})$.

CALL-DEQ Let $s \xrightarrow{inv(deq, d, k)}_{AbsQ} s'$ and $t' \in bs[s']$. Pick t such that it is equal to t' in every field except that $k \notin dom(cp_t^0)$. Then, $t \in bs[s]$ and $t \xrightarrow{inv(deq, d, k)}_{AbsQ_0} t'$.

LIN-DEQ1 Let $s \xrightarrow{lin(deq, d, k)}_{AbsQ} s'$, $t' \in bs[s']$ and $d \neq \text{EMPTY}$. We pick t such that $\sigma_t = \langle d \rangle \circ \sigma_{t'}$. We first show that $t \in bs[s]$. Let $(D, \prec)$ be the partial order that is linearized to obtain $\sigma_{t'}$ and $k' \in O_s$ be the element such that $\ell_{s_1}(k') = d$. We know that k' is minimal in $<_s$ due to the premise of the rule LIN-DEQ1. Hence, we can obtain $(D', \prec')$ such that $D' = D \cup \{\ell_{s_1}(k')\}$ and σ_t is a linearization of it.

In addition, $t \xrightarrow{lin(deq, d, k)}_{AbsQ_0} t'$. The action $lin(deq, d, k)$ is enabled in state t since d is the minimum element of σ_t . Note that the transition relating t to t' obeys the definition of normal forward simulation.

LIN-DEQ2 Let $s \xrightarrow{lin(deq, \text{EMPTY}, k)}_{AbsQ} s'$ and $t' \in bs[s']$. We pick $(D, \prec)$ for relating s to t such that $D = \emptyset$. Such a D is a valid choice since all the elements O_s are pending. Then, $\sigma_t = \langle \rangle$ is the only linearization of $(D, \prec)$. Hence, $lin(deq, \text{EMPTY}, k)$ action is enabled in $AbsQ_0$ and $t \xrightarrow{lin(deq, \text{EMPTY}, k)}_{AbsQ_0} t'$ holds.

RET-ENQ1 Let $s \xrightarrow{ret(enq,k)}_{AbsQ} s'$, $\ell_s(k) = (d, \text{PEND})$ and $t' \in bs[s']$. Assume $(D, \prec)$ be the partial order of which linearization is $\sigma_{t'}$. Pick $D' = D$. Then, $\ell_1(\text{COMP}(O_s)) \subseteq D \subseteq \ell_1(O_s)$ holds since $\text{COMP}(O_s) = \text{COMP}(O_{s'}) \setminus \{k\}$ and $k \in \text{PEND}(O_s)$. Construct $t \in bs[s]$ such that $\sigma_t = \sigma_{t'}$ is obtained by linearizing the partial order $(D', \prec')$. Then, $t \xrightarrow{ret(enq,k)}_{AbsQ_0} t'$ holds and it is a valid action with respect to normal backward-simulation relation definition.

RET-ENQ2 Let $s \xrightarrow{ret(enq,k)}_{AbsQ} s'$, $k \notin O_s$ and $t' \in bs[s']$. Since $O_s = O_{s'}$ and $\text{COMP}(O_s) = \text{COMP}(O_{s'})$, we can pick $D' = D$ where $(D, \prec)$ is the strict partial order such that $\sigma_{t'}$ is its linearization. Construct $t \in bs[s]$ such that $\sigma_t = \sigma_{t'}$ is obtained by linearizing the partial order $(D', \prec')$. Then, $t \xrightarrow{ret(enq,k)}_{AbsQ_0} t'$ holds and it is a valid action with respect to normal backward-simulation relation definition.

RET-DEQ Let $s \xrightarrow{ret(deq,d,k)}_{AbsQ} s'$ and $t' \in bs[s']$. Assume $(D, prec)$ is the partial order of which linearization is $\sigma_{t'}$. Construct $t \in bs[s]$ such that $\sigma_t = \sigma_{t'}$ and $(D, \prec)$ is the partial order σ_t is obtained from. $\text{COMP}(O_s) \subseteq D \subseteq O_s$ since $\text{COMP}(O_s) = \text{COMP}(O_{s'})$ and $O_s = O_{s'}$. Then, $t \xrightarrow{ret(deq,d,k)}_{AbsQ_0} t'$ holds. We have $rv_s(k) = rv_t^0(k)$ since $t \in bs[s]$. Hence the $ret(deq, d, k)$ is enabled in t . Moreover, $ret(deq, d, k)$ is a valid transition with respect to the normal backward simulation relation definition.

□

Lemma 2. $AbsQ_0$ is a refinement of $AbsQ$.

Proof. We define a normal $C \cup R \cup Lin(deq)$ -forward simulation fs from $AbsQ_0$ to $AbsQ$ as follows. Given $AbsQ_0$ state $t = \langle \sigma, in^0, rv^0, cp^0 \rangle$ and an $AbsQ$ state $s = \langle O, <, \ell, rv, cp \rangle$ we have that $(t, s) \in fs$ iff the following hold:

- the sequence σ is a linearization of a partial order $(D, \prec)$ where D contains values labeling elements of O and all the values corresponding to completed

enqueues, i.e., $\ell_1(\text{COMP}(O)) \subseteq D \subseteq \ell_1(O)$ ordered according to the happens-before order between the enqueues that added them, i.e., $d_1 \prec d_2$ iff there exists k_1, k_2 such that $\ell_1(k_1) = d_1$, $\ell_1(k_2) = d_2$, and $k_1 < k_2$.

- every dequeue is at the same control point in both s and t , i.e., for every k and $i \in \{1, 2, 3\}$, $cp(k) = R_i$ iff $cp^0(k) = R_i$,
- every enqueue is pending in s whenever it is pending in t , i.e., for every k , $cp(k) = A_1$ iff $cp^0(k) \in \{A_1, A\}$,
- every enqueue is completed in s whenever it is completed in t , i.e., for every k , $cp(k) = A_2$ iff $cp^0(k) = A_2$,
- every pending enqueue which is not linearized or whose value is present in σ is a member of O , i.e., for every k ,

$k \in O \wedge \ell(k) = (d, \text{PEND})$ iff

$$(cp^0(k) = A_1 \wedge in^0(k) = d) \vee (\exists i. \sigma_i = d \wedge cp^0(k) = A \wedge in^0(k) = d)$$

- every completed enqueue whose value is present in σ is a member of O , i.e., for every k ,

$$k \in O \wedge \ell(k) = (d, \text{COMP}) \text{ iff } \exists i. \sigma_i = d \wedge cp^0(k) = A_2 \wedge in^0(k) = d$$

- pending enqueues are maximal in $<$, i.e., for every k and k' , $k \not\prec k'$ if $\ell_2(k) = \text{PEND}$,
- the return values fixed at dequeue linearization points are the same, i.e., for every k , $rv(k) = rv^0(k)$.

In the following, we show that indeed fs is a normal $C \cup R \cup \text{Lin}(deq)$ -forward simulation from $AbsQ_0$ to $AbsQ$.

$$\langle i \rangle \quad fs[s_0^{AbsQ_0}] = \{s_0^{AbsQ}\}$$

CALL-ENQ Let $t \xrightarrow{inv(enq,d,k)}_{AbsQ_0} t'$ and $s \in fs[t]$. Then, $inv(enq,d,k)$ is an enabled action in $AbsQ$ since premise of CALL-ENQ holds in t and $s \in fs[t]$. Obtain s' such that $s \xrightarrow{inv(enq,d,k)}_{AbsQ} s'$. Note that s' is unique since $AbsQ$ is deterministic with respect to $C \cup R \cup Lin(deq)$.

Next, we show that $s' \in fs[t']$. Let $(D, \prec)$ be the partial order used while relating t to s . Same partial order can be used while relating $\sigma_{s'}$ to t' since $COMP(O_s) = COMP(O_{s'})$, $O_s \subseteq O_{s'}$ and $<_s \subseteq <_{s'}$. The only change we have in control point fields after the actions is that $cp_{s'}^0(k) = A_1$ and $cp_{t'}(k) = A_1$ which satisfies the conditions on fs . Moreover k is a maximal pending node in t' as required by the fs conditions. Consequently, $s' \in fs[t']$.

CALL-DEQ Let $t \xrightarrow{inv(deq,k)}_{AbsQ_0} t'$ and $s \in fs[t]$. Then, $inv(deq,k)$ is an enabled action in $AbsQ$ since premise of CALL-DEQ holds in t and $s \in fs[t]$. Obtain s' such that $s \xrightarrow{inv(deq,k)}_{AbsQ} s'$. Note that s' is unique since $AbsQ$ is deterministic with respect to $C \cup R \cup Lin(deq)$.

Next, we show that $s' \in fs[t']$. Since $\sigma_s = \sigma_{s'}$, $O_s = O_{s'}$ and $COMP(O_s) = COMP(O_{s'})$, we can pick same $(D, \prec)$ partial order in s' and show that $\sigma_{t'}$ is a linearization of it. The only change in control points after the transitions is that $cp_{t'}^0(k) = cp_{s'}(k) = R_1$ which does not violate any condition in fs . Consequently, $s' \in fs[t']$.

LIN-ENQ Let $t \xrightarrow{lin(enq,d,k)}_{AbsQ_0} t'$ and $s \in fs[t]$. Then, pick $s' = s$ such that $s \xrightarrow{\epsilon}_{AbsQ} s'$. Note that ϵ is a valid transition with respect to the normal forward simulation relation definition. We show that $s \in fs[t']$. If $(D, \prec)$ is the partial order in s of which one linearization is σ_t , we pick $D' = D \cup \{k\} \subseteq O_s$. $(D', \prec')$ can be linearized to $\sigma_{t'}$ since k is a maximal pending node and can be linearized at the end. Moreover, the only change in control point $cp_{t'}^0(k) = A$ which does not violate the fs conditions.

LIN-DEQ1 Let $t \xrightarrow{lin(deq,d,k)}_{AbsQ_0} t'$, $d \neq \text{EMPTY}$ and $s \in fs[t]$. Then, $lin(deq,d,k)$ is an

enabled action in $AbsQ$. There must exist $d \in D \subseteq \ell_1(O_s)$ such that $\ell_{s1}(k') = d$ and k' is minimal in D (since d is linearized as the minimum element in σ_t according to premise of LIN-DEQ1 of $AbsQ$). Obtain s' such that $s \xrightarrow{lin(deq,d,k)}_{AbsQ} s'$. Note that s' is unique since $AbsQ$ is deterministic with respect to $C \cup R \cup Lin(deq)$.

Next, we show that $s' \in fs[t']$. Let $(D, \prec)$ be the partial order used while relating t to s such that σ_t is a linearization of this partial order. Since we have shown that k' is minimal in that partial order, $\sigma_{t'}$ is a linearization of $(D', \prec')$ where $D' = D \setminus \{\ell_1(k')\}$. Note that $\ell_1(\mathbf{COMP}(O_{s'})) \subseteq D' \subseteq \ell_1(O_{s'})$ holds. The only change in control points is that $cp_{t'}^0(k) = cp_{s'}(k) = R_2$ which does not violate the conditions for relating t' to s' . Note that the fifth condition of fs still holds for k' while relating t' to s' . After transitions $rv_{t'}^0(k) = rv_{s'}(k) = d$ and the last condition on fs is preserved.

LIN-DEQ2 Let $t \xrightarrow{lin(deq, \mathbf{EMPTY}, k)}_{AbsQ_0} t'$ and $s \in fs[t]$. Then, $lin(deq, d, k)$ is an enabled action in $AbsQ$. If $\mathbf{COMP}(O_t) \neq \emptyset$, then D use for linearization cannot be $\emptyset$ $\sigma_t = \langle \rangle$ cannot be a linearization of $(D, \prec)$. Obtain s' such that $s \xrightarrow{lin(deq, \mathbf{EMPTY}, k)}_{AbsQ} s'$. Note that s' is unique since $AbsQ$ is deterministic with respect to $C \cup R \cup Lin(deq)$.

Next, we show that $s' \in fs[t']$. Let $(D, \prec)$ be the partial order used while relating t to s such that $\sigma_t = \langle \rangle$ is a linearization of this partial order. We can use the same $(D, \prec)$ for relating t' to s' because σ field is the same for both s , s' ; and O , $<$, ℓ fields are same for both t and t' . The only change in control points is that $cp_{t'}^0(k) = cp_{s'}(k) = R_2$ which does not violate the conditions for relating t' to s' . After transitions $rv_{t'}^0(k) = rv_{s'}(k) = d$ and the last condition on fs is preserved.

RET-ENQ Let $t \xrightarrow{ret(enq, k)}_{AbsQ_0} t'$ and $s \in fs[t]$. Then, there are two cases assuming data independence: (i) $in_t^0(k) = d$ and $\exists i. \sigma_t(i) = d$ (ii) or not.

First, consider the former case. Then, RET-ENQ1 rule of $AbsQ$ is applicable. Its precondition holds since fifth condition of fs holds while relating t to s . Apply this rule ($ret(enq, k)$) to obtain s' . Note that s' is unique since $AbsQ$ is deterministic with respect to $C \cup R \cup Lin(deq)$ and it is a valid action according to the normal forward-simulation relation definition.

Next, we show that $s' \in fs[t']$. Since $\exists i. \sigma_t(i) = d$, we know that $d \in D$ where $(D, \prec)$ is the partial order satisfying first condition of fs while relating t to s , and $k \in O_s$ takes part in the linearization i.e., $\ell_{t1}(k) \in D$. We can use the same partial order $(D, \prec)$ for relating t' to s' such that it satisfies the first condition of fs . The only change in control points is that $cp_{t'}^0(k) = cp_{s'}(k) = A_2$ which does not violate the conditions for relating t' to s' . Note that the sixth condition of fs also continue to hold for k for the post-states.

Second, consider the latter case: $in_t^0(k) = d$, but $\forall i. \sigma_t(i) \neq d$. Since $(t, s) \in fs$, $k \notin O_s$ by the fifth and sixth conditions. Hence, the pre-condition of RET-ENQ2 is satisfied by t . Apply this rule ($ret(enq, k)$) to obtain s' . Note that s' is unique since $AbsQ$ is deterministic with respect to $C \cup R \cup Lin(deq)$ and it is a valid action according to the normal forward-simulation relation definition.

Next, we show that $s' \in fs[t']$. For satisfying the first condition, one can use the same $(D, \prec)$ partial order that is used for relating pre-states since σ fields of t, t' and $O, <, \ell$ fields of s and s' are the same. The only change in control points is that $cp_{t'}^0(k) = cp_{s'}(k) = A_2$ which does not violate the conditions for relating t' to s' .

RET-DEQ Let $t \xrightarrow{ret(deq, d, k)}_{AbsQ_0} t'$ and $s \in fs[t]$. Then, $ret(deq, d, k)$ is an enabled action in $AbsQ$ due to premise RET-DEQ of $AbsQ_0$ and the last condition on fs (since $(t, s) \in fs$). Obtain s' such that $s \xrightarrow{ret(deq, d, k)}_{AbsQ} s'$. Note that s' is unique since $AbsQ$ is deterministic with respect to $C \cup R \cup Lin(deq)$.

We see that $s' \in fs[t']$. Pre-states are equal to the post-states with the only exception in the control points such that $cp_{t'}^0(k) = cp_{s'}(k) = R_3$. All the condi-

$$\begin{array}{c}
\text{CALL-PUSH} \qquad \qquad \qquad \text{LIN-PUSH} \\
\frac{k \notin \text{dom}(cp^0) \quad d \neq \text{EMPTY}}{\sigma, in^0, rv^0, cp^0 \xrightarrow{\text{inv}(\text{push}, d, k)} \sigma, in^0[k \mapsto d], rv^0, cp^0[k \mapsto A_1]} \quad \frac{cp^0(k) = A_1}{\sigma, in^0, rv^0, cp^0 \xrightarrow{\text{lin}(\text{push}, d, k)} d \cdot \sigma, in^0, rv^0, cp^0[k \mapsto A]} \\
\\
\text{RET-PUSH} \\
\frac{cp^0(k) = A}{\sigma, in^0, rv^0, cp^0 \xrightarrow{\text{ret}(\text{push}, k)} \sigma, in^0, rv^0, cp^0[k \mapsto A_2]} \\
\\
\text{CALL-POP} \qquad \qquad \qquad \text{LIN-POP1} \\
\frac{k \notin \text{dom}(cp^0)}{\sigma, in^0, rv^0, cp^0 \xrightarrow{\text{inv}(\text{pop}, k)} \sigma, in^0, rv^0, cp^0[k \mapsto R_1]} \quad \frac{cp^0(k) = R_1 \quad \sigma = d \cdot \sigma'}{\sigma, in^0, rv^0, cp^0 \xrightarrow{\text{lin}(\text{pop}, d, k)} \sigma', in^0, rv^0[k \mapsto d], cp^0[k \mapsto R_2]} \\
\\
\text{LIN-POP2} \qquad \qquad \qquad \text{RET-POP} \\
\frac{cp^0(k) = R_1 \quad \sigma = \epsilon}{\sigma, in^0, rv^0, cp^0 \xrightarrow{\text{lin}(\text{pop}, \text{EMPTY}, k)} \sigma, in^0, rv^0[k \mapsto \text{EMPTY}], cp^0[k \mapsto R_2]} \quad \frac{cp^0(k) = R_2 \quad rv^0(k) = d}{\sigma, in^0, rv^0, cp^0 \xrightarrow{\text{ret}(\text{pop}, d, k)} \sigma, in^0, rv^0, cp^0[k \mapsto R_3]}
\end{array}$$

Figure 2.9: The transition relation of $AbsS_0$.

tions except the third one continues to hold in the post states since they hold in the pre-states. The third rule regarding the control points of dequeues also continue to hold since changes in the control point of k does not violate it.

□

D Proof of Theorem 4

We show that $AbsS$ and $AbsS_0$ refine each other. The standard reference implementation $AbsS_0$ is defined exactly as the one for queues, $AbsQ_0$, except that pop linearization points extract values from the beginning of the sequence stored in the state.

Thus, the states of $AbsS_0$ are tuples $\langle \sigma, in^0, rv^0, cp^0 \rangle$ where $\sigma \in \mathbb{V}^*$ is a sequence of values, $in^0 : \mathbb{O} \rightarrow \mathbb{V}$ records the input value of a push, $rv^0 : \mathbb{O} \rightarrow \mathbb{V}$ records the return value of a pop fixed at its linearization point ($\rightarrow$ denotes a partial function), and $cp^0 : \mathbb{O} \rightarrow \{A_1, A, A_2, R_1, R_2, R_3\}$ records the control point of every push (A_1, A, A_2) or pop operation (R_1, R_2, R_3). All the components are $\emptyset$ in the initial state, and the

transition relation $\rightarrow$ is defined in Fig. 2.9. The alphabet of $AbsS$ contains call/return actions and push/pop linearization points.

To prove that $AbsS$ is a refinement of $AbsS_0$ we define a normal $C \cup R$ -backward simulation (i.e, a backward simulation as in Definition 6) from $AbsS$ to $AbsS_0$. The reverse is shown using a normal $C \cup R$ -forward simulation (i.e, a forward simulation as in Definition 5).

Lemma 3. *$AbsS$ is a refinement of $AbsS_0$.*

Proof. We define a normal $C \cup R$ -backward simulation bs from $AbsS$ to $AbsS_0$ as follows. Given an $AbsS$ state $s = \langle O, <, \ell, rv, cp \rangle$ and an $AbsS_0$ state $t = \langle \sigma, in^0, rv^0, cp^0 \rangle$ we have that $(s, t) \in bs$ iff the following hold:

- if a pop has committed or respectively, it has returned in s , then it had been linearized or respectively, it has returned in t , i.e., for every k , if $cp(k) \in \{R_2, R_3\}$ then $cp^0(k) = cp(k)$,
- a push is completed in s whenever the same is true in t , i.e., for every k , $cp(k) = A_2$ iff $cp^0(k) = A_2$,
- a push is pending in s iff either it is a non-linearized pending push in t or its linearization point has been executed, i.e., for every k , $cp(k) = A_1$ iff $cp^0(k) = A_1$ and $in^0(k)$ doesn't occur in σ , or $cp(k) = A$,
- if a pop didn't commit in s then it is pending in t and it may have been linearized, i.e., for every k , if $cp(k) = R_1$ then $cp^0(k) \in \{R_1, R_2\}$,
- there exists a partial injective function $g : \{k : cp(k) = R_1\} \rightarrow O$ which associates uncommitted pops to pushes in O such that:
 - for every k , $k \in dom(g)$ iff $g(k) \in be(k) \cup ov(k)$
 - the sequence σ is the mirror of a linearization of a partial order $(D, \prec)$ where D contains values labeling elements of O except for those in the

range of g , and all the values corresponding to completed pushes which are not in the range of g , i.e., $\ell_1(\text{COMP}(O) \setminus \text{range}(g)) \subseteq D \subseteq \ell_1(O \setminus \text{range}(g))$ ordered according to the happens-before order between the pushes that added them, i.e., $d_1 \prec d_2$ iff there exists k_1, k_2 such that $\ell_1(k_1) = d_1$, $\ell_1(k_2) = d_2$, and $k_1 < k_2$

- every pop in the domain of g has been linearized, i.e., for every k , $k \in \text{dom}(g)$ implies $cp^0(k) = R_2$,
 - every pop which is not in the domain of g hasn't been linearized, i.e., for every k , $k \notin \text{dom}(g)$ implies $cp^0(k) = R_1$,
 - every push in the range of g has been linearized, i.e., for every k , $k \in \text{range}(g)$ implies $cp^0(k) = A$,
 - a pending enqueue from O has been linearized when its value is contained in σ , i.e., for every k , if $\ell_1(k) \in D$ and $\ell_2(k) = \text{PEND}$, then $cp^0(k) = A$.
- the return values fixed at pop commit points are the same, i.e., for every k , if $rv(k)$ is defined, then $rv(k) = rv^0(k)$,
 - every pending push has the same input value in both s and t , i.e., for every k , $\ell_1(k) = in^0(k)$,

In the following, we show that indeed bs is a normal $C \cup R$ -backward simulation from $AbsS$ to $AbsS_0$:

- Let $s \xrightarrow{inv(push, d, k)} s'$ be a transition in $AbsS$ and $(s', t') \in bs$. We consider two cases depending on whether the value d occurs on a position i in the sequence σ of t' or not. If it occurs, let t be a $AbsS_0$ state where essentially, the component σ is the prefix of the sequence σ of t' that contains the first $i - 1$ positions (except for some set of pushes that will be defined hereafter, all operations are at the same control point). Let τ be the following $AbsS_0$ trace:

$$\tau = inv(push, d, k), \quad lin(push, d, k), \quad lin(push, d_{i+1}, k_{i+1}), \dots, lin(push, d_{n-1}, k_{n-1})$$

where d_j is the value on position j in the sequence σ of t' and n is the length of this sequence (we assume that positions are indexed starting from 0). Let $k_i = k$. For every k_j with $i \leq j \leq n-1$, we must have that $cp(k_j) = A$ in t' . We take $cp(k_j) = A_1$ in t for every $j \geq i+1$ and cp undefined for k_i . We have that $t \xrightarrow{\tau} t'$ is a valid sequence of transitions of $AbsS_0$ and $(s, t) \in bs$ (the latter can be proved by taking the same function g used in establishing that $(s', t') \in bs$). Now, assume that the value d is not in the sequence σ of t' . We consider an $AbsS_0$ state t where the component σ is the same as the one in t' . There are two sub-cases depending on whether there exists a pending pop k' such that $g(k') = k$ when establishing that $(s', t') \in bs$. If it exists, the operations are at the same control point in both t and t' except for the push k for which $cp(k)$ is undefined in t , and the the pop k' for which we take $cp(k') = R_0$ in t . We have that

$$t \xrightarrow{inv(push,d,k), lin(push,d,k) lin(pop,d,k')} t'$$

in $AbsS_0$. If there exists no such pop k' , it can be easily seen that there exists t such that $t \xrightarrow{inv(push,d,k)} t'$ and $(s, t) \in bs$.

- Let $s \xrightarrow{inv(pop,k)} s'$ be a transition in $AbsS$ and $(s', t') \in bs$. We consider two cases depending on whether in the function g used to relate s' to t' we have that $k \in dom(g)$. In other words, either the newly invoked pop operation k did not linearize yet ($k \notin dom(g)$) or it linearizes and removes an element inserted by a linearized push ($k \in dom(g)$). The second case also splits into two sub-cases: The value removed by pop k is inserted by a push $k' = g(k)$ that is still pending or the push has returned. We will look at all three cases separately. The easiest one is the first case. There exists some t where essentially the component σ is the same as the one in t' , such that $t \xrightarrow{inv(pop,k)} t'$ and $(s, t) \in bs$.

For the first sub-case of the second case, we take an $AbsS_0$ state t where $\sigma_t = d \cdot \sigma_{t'}$ (we use σ_t to denote the component σ in t). It must happen that $cp_{t'}^0(k) = R_2$. The operations are at the same control point in both t and t' , except for

k in which case cp^0 is undefined. We have that $t \xrightarrow{inv(pop,k), lin(pop,d,k')} t'$ and $(s, t) \in bs$. The latter holds because essentially, k' is a maximal node in s' (since it is pending).

For the second sub-case, we define an $AbsS_0$ state t where the sequence σ is the minimal prefix of $\sigma_{t'}$ that includes the value d added by k' . Let i be the index of this value in $\sigma_{t'}$ and k_j with $i < j$ the identifiers of the pushes that added the values following d in $\sigma_{t'}$. Let τ be the following $AbsS_0$ trace:

$$\tau = inv(pop, k), lin(pop, d, k), lin(push, d_{i+1}, k_{i+1}), \dots, lin(push, d_{n-1}, k_{n-1})$$

where d_j is the value on position j in the sequence $\sigma_{t'}$ and n is the length of this sequence. We have that $t \xrightarrow{\tau} t'$ is a valid sequence of transitions of $AbsS_0$ and $(s, t) \in bs$. The latter relies on the fact that k' is a greatest completed push in s and all pushes k_j with $j > i$ are pending in s .

- Let $s \xrightarrow{com(pop,d,k)} s'$ be a transition in $AbsS$ and $(s', t') \in bs$. When this transition results in removing a greatest completed push or a pending push in s , then there exists an $AbsS_0$ state t such that $t \xrightarrow{\tau} t'$ is a valid sequence of $AbsS_0$ transitions and $(s, t) \in bs$, for some t and τ defined as in the second case of $inv(pop, k)$. When it removes a completed push which is followed by other completed pushes (in the happens-before in s), then we pick $t = t'$. We have that $t \xrightarrow{\epsilon} t'$ and $(s, t) \in bs$ (for the latter we must choose a function g such that $g(k) = k'$ where k' is the push removed by the $AbsS$ transition).
- Let $s \xrightarrow{ret(push,k)} s'$ be a transition in $AbsS$ and $(s', t') \in bs$. We consider two cases depending on whether the happens-before in s contains push k . If it contains push k , there are two sub-cases: (1) if its input is present in $\sigma_{t'}$ then there exists an $AbsS_0$ state t such that $t \xrightarrow{ret(push,k)} t'$ is a valid sequence of $AbsS_0$ transitions and $(s, t) \in bs$, and (2) otherwise, we take a state t where essentially, $\sigma_t = d \cdot \sigma_{t'}$ for which we have that $t \xrightarrow{lin(pop,d,k), ret(push,k)} t'$ and $(s, t) \in bs$. If the

happens-before in s doesn't contain push k , then there exists an $AbsS_0$ state t such that $t \xrightarrow{ret(push,k)} t'$.

- The case of pop returns $ret(pop, k)$ is trivial. Such transitions are simulated by $ret(pop, k)$ transitions of $AbsS_0$.

□

Lemma 4. $AbsS_0$ is a refinement of $AbsS$.

Proof. We define a normal CUR -forward simulation fs from $AbsS_0$ to $AbsS$ as follows. Given an $AbsS_0$ state $t = \langle \sigma, in^0, rv^0, cp^0 \rangle$ and an $AbsS$ state $s = \langle O, <, \ell, rv, cp \rangle$ we have that $(t, s) \in fs$ iff the following hold:

1. every pop is at the same control point in both t and s , i.e., for every k and $i \in \{1, 2, 3\}$, $cp^0(k) = R_i$ iff $cp(k) = R_i$,
2. a push has been invoked in t whenever it has been invoked in s , i.e., for every k , $cp^0(k) = A_1$ iff $cp(k) = A_1$,
3. a push which is linearized in t has been invoked in s , i.e., for every k , if $cp^0(k) = A$ then $cp(k) = A_0$,
4. a push is completed in t iff the same holds in s , i.e., for every k , $cp^0(k) = A_2$ iff $cp(k) = A_2$,
5. the pair (O, ℓ) in s satisfies the following:
 - for every k , if $in^0(k) = d$, $cp^0(k) \in \{A_1, A\}$, and d occurs in σ , then $k \in O$ and $\ell(k) = (d, \text{PEND})$,
 - for every k , if $in^0(k) = d$, $cp^0(k) = A_2$, and d occurs in σ , then $k \in O$ and $\ell(k) = (d, \text{COMP})$,
6. every pending push in O is overlapping with every non-linearized pop, i.e., for every k , if $cp^0(k) = R_1$ then $\{k' : k' \in O \wedge \ell_2(k') = \text{PEND}\} \subseteq ov(k)$.

7. every completed push is either overlapping or was the greatest completed push before a non-linearized pop started, i.e., for every k , if $cp^0(k) = R_1$, then $\text{COMP}(O) \subseteq ov(k) \cup be(k)$,
8. for every push that overlaps with a pop k or was maximal in $<$ when k started, its successors are overlapping with k , i.e., $k_1 \in be(k) \cup ov(k)$ and $k_1 < k_2$ implies $k_2 \in ov(k)$ for each k, k_1, k_2
9. predecessors of pushes in $be(k)$ for a given pop k are neither overlapping with k nor in $be(k)$, i.e., $k_1 < k_2$ and $k_2 \in be(k)$ implies $k_1 \notin ov(k) \cup be(k)$ for each k, k_1, k_2
10. pending pushes are maximal in $<$, for every k and k' , $k \not< k'$ if $\ell_2(k) = \text{PEND}$,
11. the sequence σ is the mirror of a linearization of a partial order $(D, \prec)$ where D contains values labeling elements of O and all the values corresponding to completed pushes, i.e., $\ell_1(\text{COMP}(O)) \subseteq D \subseteq \ell_1(O)$ ordered according to the happens-before order between the pushes that added them, i.e., $d_1 \prec d_2$ iff there exists k_1, k_2 such that $\ell_1(k_1) = d_1$, $\ell_1(k_2) = d_2$, and $k_1 < k_2$.
12. the return values fixed at pop linearization/commit points are the same, i.e., for every k , $rv(k) = rv^0(k)$,
13. every pending push has the same input value in both s and t , i.e., for every k , $\ell_1(k) = in^0(k)$,

In the following, we show that indeed fs is a normal $C \cup R$ -backward simulation from $AbsS_0$ to $AbsS$:

- Let $t \xrightarrow{inv(push,d,k)} t'$ be a transition in $AbsS_0$ and $(t, s) \in fs$. We have that $(t', s') \in fs$ where $s \xrightarrow{inv(push,d,k)} s'$ (recall that $AbsS$ is deterministic). Since the push k is non-linearized in t' , the component σ of both t and t' are the same and $\ell_2(k) = \text{PEND}$ in s' . Then, the component σ in $AbsS_0$ states related by fs

to s' is allowed to exclude values added by pushes in s' which are labeled as pending. The effect of $inv(push, d, k)$ in $AbsS$ implies that k overlaps with all pending pops.

- Let $t \xrightarrow{inv(pop, k)} t'$ be a transition in $AbsS_0$ and $(t, s) \in fs$. We have that $(t', s') \in fs$ where $s \xrightarrow{inv(pop, k)} s'$. The only difference between s and s' is that the components $be(k)$ and $ov(k)$ in s' contain the greatest completed pushes in s and the pending pushes in s , respectively (these components were undefined in s). The relation fs doesn't exclude this particular choice for $be(k)$ and $ov(k)$ when applied to t' and s' .
- Let $t \xrightarrow{lin(push, d, k)} t'$ be a transition in $AbsS_0$ and $(t, s) \in fs$. We have that $(t', s) \in fs$, i.e., the $AbsS_0$ transition is simulated by an empty sequence of $AbsS$ transitions, because essentially the component σ of t' still corresponds to a linearization of the pushes in s according to item 11 in the definition of fs . The sequence σ in t' contains the value added by the push k at the end, but this is allowed by fs since k is labeled as pending in s .
- Let $t \xrightarrow{lin(pop, d, k)} t'$ be a transition in $AbsS_0$ and $(t, s) \in fs$. We have that $(t', s') \in fs$ where $s \xrightarrow{com(pop, d, k)} s'$. The transition labeled by $com(pop, d, k)$ is enabled in $AbsS_0$ because d was the first value in the sequence σ of t . Indeed, this implies that d was added by a push k' which is maximal in the happens-before stored in s . This clearly implies that $k' \in be(k) \cup ov(k)$. In addition, the sequence σ in t' does correspond to a linearization of the pushes in s' (which don't contain k anymore) because σ in t had this property with respect to s and σ in t' is obtained by deleting the first value in the sequence σ of t .
- Let $t \xrightarrow{ret(push, k)} t'$ be a transition in $AbsS_0$ and $(t, s) \in fs$. We have that $(t', s') \in fs$ where $s \xrightarrow{ret(push, k)} s'$. There are two cases depending on whether the value added by k is still present in the sequence σ of t . If it is not, then the push k doesn't occur in the happens-before from s , and the only effect of these

two transitions is changing the control point of k . Therefore, $(t', s') \in fs$ clearly holds. When this value is still present, the effect of $ret(push, k)$ in $AbsS$ is changing the flag of push k from **PEND** to **COMP**. Since the order between pushes doesn't change, we have that $(t', s') \in fs$.

- Let $t \xrightarrow{ret(pop, d, k)} t'$ be a transition in $AbsS_0$ and $(t, s) \in fs$. We have that $(t', s') \in fs$ where $s \xrightarrow{ret(pop, d, k)} s'$. This case is obvious, the only change between s and s' being the control point of k .

□

E Proving the correctness of TSS

The LTS corresponding to the description of TSS given in Fig. 2.4 is defined as usual. The control points and transition labels we use in the following proof are pictured in Fig. 2.10. To simplify the proof, we take the initializations of some local variables together as atomic.

States of the TS-Stack contains the global variables and local variables as fields. Global variables are just elements of their domains and local variables are maps from operation identifiers to their domains. We say $i_q(k)$ for referencing the value of local variable i of operation k in state q . There is only one special local variable called $myTID$. Its value is unique to each pending operation in a state i.e., concurrent operations cannot have the same $myTID$ value. TS-Stack states also contains sets $O_a, O_r \in \mathbb{O}$ which are operation identifier sets of push and pops respectively, and the control point function cp which is a map from operation identifiers to the control points set that are presented in the flow diagram Figure 2.10. Transition relation of the TS-Stack is presented in Figure 2.11 (push rules) and Figure 2.12 (pop rules). Next, we show that the linearizability of TS Stack.

Lemma 5. *TSS is a $C \cup R \cup Com(pop)$ -refinement of AbsS.*

Proof. We show that the relation fs_2 defined in Section 2.5.3 is a $C \cup R \cup Com(pop)$ -forward simulation from TSS to $AbsS$. For readability, we recall the definition of

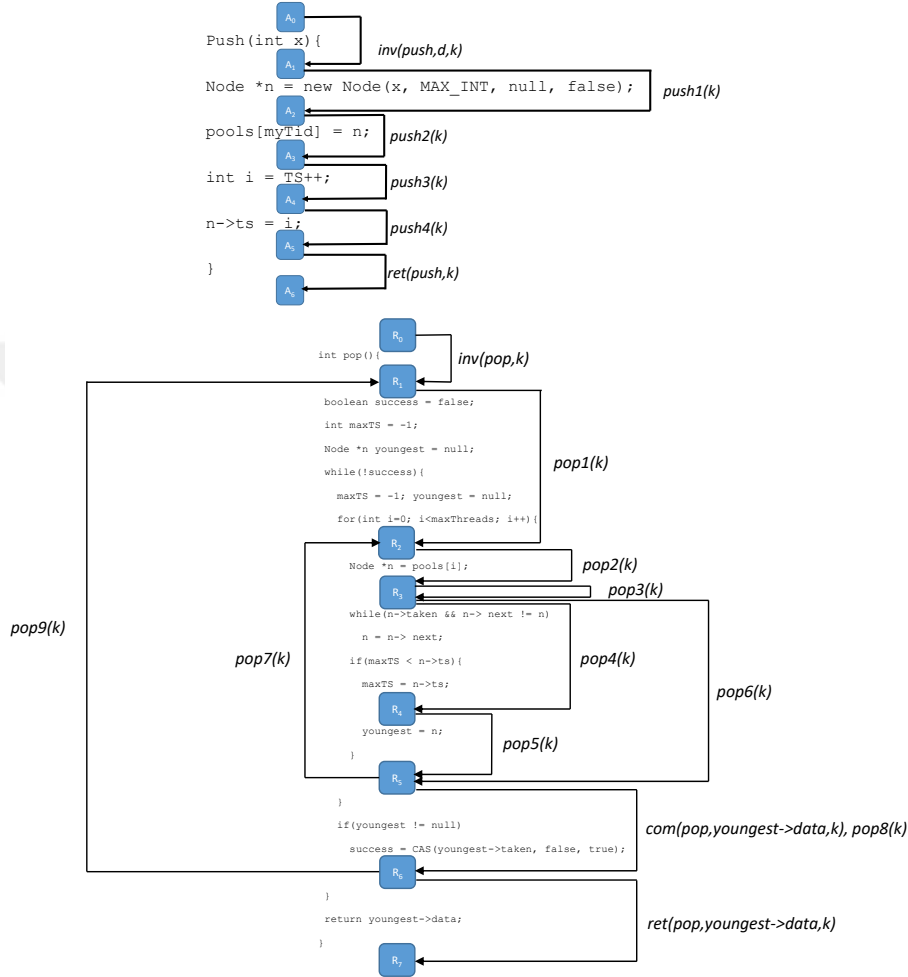


Figure 2.10: The flow diagram for the pop and push methods of the Time-Stamped Stack algorithm. The blue points show the control points roughly and the arrows show the possible transitions.

fs_2 .

Let us make some clarifications before defining the relation. In order not to confuse nodes in TS Stack and nodes in $AbsS$, we call nodes of $AbsS$ as vertices from now on. We also define ordering relation (called traverse order) among the operations in a state of TS . It basically reflects the traverse order of pop operations. For two push operations $m, n \in O_a$ is state s we say that $m <_s^{tr} n$ iff either $myTid(m) < myTid(n)$ or $myTid(m) = myTid(n)$ and $n_s(n)$ is reachable from $n_s(m)$ using next pointers. $\geq^{tr}$ is obtained from $<^{tr}$ in the usual way.

$$\begin{array}{c}
\text{CALL-PUSH} \\
\frac{k \notin \text{dom}(cp) \quad d \neq \text{null}}{\dots, O_a, x, cp, \dots \xrightarrow{\text{inv}(\text{push}, d, k)} \dots, O_a \cup \{k\}, x[k \mapsto d], cp[k \mapsto A_1], \dots} \\
\\
\text{PUSH1} \\
\frac{cp(k) = A_1 \quad *n' = (x(k), \text{MAX_INT}, \text{null}, \text{false})}{\dots, n, cp, \dots \xrightarrow{\text{push1}(k)} \dots, n[k \mapsto n'], cp[k \mapsto A_2], \dots} \\
\\
\text{PUSH2} \\
\frac{cp(k) = A_2}{\dots, pools, cp, \dots \xrightarrow{\text{push2}(k)} \dots, pools[\text{myTid}(k) \mapsto n(k)], cp[k \mapsto A_3], \dots} \\
\\
\text{PUSH3} \\
\frac{cp(k) = A_3}{\dots, i, TS, cp, \dots \xrightarrow{\text{push3}(k)} \dots, i[k \mapsto TS], TS + 1, cp[k \mapsto A_4], \dots} \\
\\
\text{PUSH4} \\
\frac{cp(k) = A_4 \quad n'(k) = n(k)[ts \mapsto i(k)] \quad \forall k'. cp(k') = A_6 \implies i(k') < i(k)}{\dots, n, cp, \dots \xrightarrow{\text{push4}(k)} \dots, n[k \mapsto n'(k)], cp[k \mapsto A_5], \dots} \\
\\
\text{RET-PUSH} \\
\frac{cp(k) = A_5}{\dots, cp, \dots \xrightarrow{\text{ret}(\text{push}, k)} \dots, cp[k \mapsto A_6], \dots}
\end{array}$$

Figure 2.11: The push derivation rules of *TSS*. We only mention the state components that are modified. Unmentioned state components have the names in the algorithm in the prestate. $*n = (a, b, c, d)$ is shorthand for $n \rightarrow data = a$, $n \rightarrow ts = b$, ... $n' = n[ts \mapsto expr]$ is short for $n' \rightarrow ts = expr$ and all the other fields of n and n' are the same.

The relation $fs_2 \subseteq Q_C \rightarrow Q_{AbsS}$ contains (s, t) iff the following are satisfied:

Nodes $k \in O_t$ iff k is a push operation in s ($k \in O_a$) such that either it has not inserted its node to the pool yet ($cp_s(k) = A_i$ and $i < 3$) or its node is not taken by a pop ($cp_s(k) = A_i$, $i \geq 3$ and $n_s(k) \rightarrow taken = false$).

Pend/Comp A vertex $k \in O_t$ is pending ($\ell_t(k) = (d, \text{PEND})$) iff k satisfies the previous condition, $x_s(k) = d$ and it is not completed in s ($cp_s(k) = A_i$ and $i < 6$). Similarly, this vertex is completed ($\ell_t(k) = (d, \text{COMP})$) iff k satisfies the previous condition, $x_s(k) = d$ and it is completed in s ($cp_s(k) = A_6$). Pending vertices are maximal

CALL-POP $\frac{k \notin \text{dom}(cp)}{O_r, cp \xrightarrow{\text{inv}(\text{pop}, k)} O_r \cup \{k\}, cp[k \mapsto R_1]}$	RET-POP $\frac{cp(k) = R_6 \quad \text{suc}(k) = \text{false} \quad d = \text{yst}(k) \rightarrow \text{data}}{cp \xrightarrow{\text{ret}(\text{pop}, d, k)} cp[k \mapsto R_7]}$
POP1 $\frac{cp(k) = R_1 \quad \text{maxThreads} > 0}{\text{suc}, \text{yst}, mTS, i, cp \xrightarrow{\text{pop1}(k)} \text{suc}[k \mapsto \text{false}], \text{yst}[k \mapsto \text{null}], mTS[k \mapsto -1], i[k \mapsto 0], cp[k \mapsto R_2]}$	
POP2 $\frac{cp(k) = R_2 \quad 0 \leq i(k) < \text{maxThreads}}{n, cp \xrightarrow{\text{pop2}(k)} n[k \mapsto \text{pools}(i(k))], cp[k \mapsto R_3]}$	POP5 $\frac{cp(k) = R_4}{\text{yst}, cp \xrightarrow{\text{pop5}(k)} \text{yst}[k \mapsto n(k)], cp[k \mapsto R_5]}$
POP3 $\frac{cp(k) = R_3 \quad n(k) \neq \text{null} \quad n(k) \rightarrow \text{tkn} = \text{true} \quad n(k) \rightarrow \text{next} \neq n(k)}{n \xrightarrow{\text{pop3}(k)} n[k \mapsto n(k) \rightarrow \text{next}]}$	POP9 $\frac{cp(k) = R_6 \quad \text{suc}(k) = \text{false}}{cp \xrightarrow{\text{pop9}(k)} cp[k \mapsto R_1]}$
POP4 $\frac{cp(k) = R_3 \quad n(k) \neq \text{null} \quad n(k) \rightarrow \text{tkn} = \text{false} \quad n(k) \rightarrow \text{ts} > \text{maxTS}(k)}{\text{maxTS}, cp \xrightarrow{\text{pop4}(k)} \text{maxTS}[k \mapsto n(k) \rightarrow \text{ts}], cp[k \mapsto R_4]}$	
POP6 $\frac{cp(k) = R_3 \quad n(k) \neq \text{null} \quad n(k) \rightarrow \text{tkn} = \text{false} \quad n(k) \rightarrow \text{ts} \leq \text{maxTS}(k)}{cp \xrightarrow{\text{pop6}(k)} cp[k \mapsto R_5]}$	
POP7 $\frac{cp(k) = R_5 \quad i(k) < \text{maxThreads} - 1}{i, cp \xrightarrow{\text{pop7}(k)} i[k \mapsto i(k) + 1], cp[k \mapsto R_2]}$	POP8 $\frac{cp(k) = R_5 \quad \text{yst}(k) = \text{null} \vee (\text{yst}(k) \neq \text{null} \wedge \text{yst} \rightarrow \text{tkn})}{\text{suc}, cp \xrightarrow{\text{pop7}(k)} \text{suc}[k \mapsto \text{false}], cp[k \mapsto R_6]}$
COM-POP $\frac{cp(k) = R_5 \quad \text{yst}(k) \neq \text{null} \quad \text{yst}(k) = m \quad d = m \rightarrow \text{data} \quad m \rightarrow \text{tkn} = \text{false} \quad m' = m[\text{tkn} \mapsto \text{true}]}{\text{suc}, \text{yst}, cp \xrightarrow{\text{com}(\text{pop}, d, k)} \text{suc}[k \mapsto \text{true}], \text{yst}[k \mapsto m'], cp[k \mapsto R_6]}$	

Figure 2.12: Pop derivation rules for the *TSS*. We only mention state components that are modified. $n' = n[\text{taken} \mapsto \text{expr}]$ is short for $n' \rightarrow \text{taken} = \text{expr}$ and all the other fields of n and n' are the same. We abbreviate **youngest**, **success** and **taken** fields as **yst**, **suc** and **tkn**, respectively, for space reasons.

with respect to $<_t$ i.e., if $k \in O_t$ is a pending vertex, then for all $k' \in O_t$ $k \not\prec_t k'$.

TSOrder If a node has a smaller timestamp than the other node in s , the operations that inserted them cannot be ordered reversely in t . More formally, let $k, k' \in O_t$ s.t. $n_s(k) \rightarrow \text{ts} \leq n_s(k') \rightarrow \text{ts}$. Then, $k' \not\prec_t k$.

TidOrder Order among the nodes inserted by the same threads in s must be preserved among the operations that inserted them in t . Let $k, k' \in O_t$ s.t. $myTid_s(k) = myTid_s(k')$ and $n_s(k) \rightarrow ts < n_s(k') \rightarrow ts$. Then, $k <_t k'$.

Frontiers Every maximally closed or pending vertex can be removed by a pending pop. More formally, let $k \in O_t$ such that $\ell_t(k) = (_, \text{PEND})$. Then, for all pops p , $k \in ov_t(p)$. In the other case, let $k \in O_t$ such that $\ell_t(k) = (_, \text{COMP})$ and for all other $k' \in O_t$ such that $k <_t k'$, we know $\ell_t(k') = (_, \text{PEND})$. Then, for all pop operations p , $k \in be_t(p)$ or $k \in ov_t(p)$.

MaximalOV If a push $k \in O_t$ is a candidate to be removed by a pop p , then every other push k' invoked after k is a candidate to be removed by p since k is concurrent with p . More formally, let $k, k' \in O_t$ such that $k <_t k'$ and there exists a pop p such that $k \in be_t(p)$ or $k \in ov_t(p)$. Then, $k' \in ov_t(p)$.

MinimalBE If a push $k \in O_t$ has finished before the pop p is invoked and yet k is a candidate to be removed by p , other pushes completed before k can not be candidates to be removed by p at that state. More formally, let $k, k' \in O_t$ such that $k <_t k'$ and there exists a pop p such that $k' \in be_t(p)$. Then, neither $k \in be_t(p)$ nor $k \in ov_t(b)$.

ReverseFrontiers If all immediate followers $k' \in O_t$ of a push $k \in O_t$ are concurrent with pop p , then k is either concurrent or maximally closed with respect to p . More formally, let $k \in O_t$ and for all $k' \in O_t$ such that $k \in pred_{<_t}(k')$, $k' \in ov_t(p)$, where p is a pop operation. Then, $k \in ov_t(p) \cup be_t(p)$.

FixReturn If a pop p is after its commit point action in s , then the rv value of this operation in t is fixed to $youngest_s(p) \rightarrow data$. More formally, Let p be the pop operation such that $cp_s(p) = R_6$ and $success_s(p) = \text{true}$. Then, $rv_t(p) = youngest_s(p) \rightarrow data$.

TraverseBefore If a pop operation p is currently visiting node n , it has non-null node y as the *youngest* and there is a non-null not taken node m coming before n in the traverse order with a greater timestamp than y , then the operation that inserts m must be concurrent with p . More formally, assume $youngest_s(p) = y$ and $y \neq \mathbf{null}$. Let $k \in O_t$ such that $n_s(k) \neq \mathbf{null}$, $n_s(k) \rightarrow taken = \mathbf{false}$, $n_s(k) <_s^{tr} n_s(p)$ and $n_s(k) \rightarrow ts \geq y \rightarrow ts$. Then, $k \in ov_t(p)$.

TraverseBeforeNull If a pop operation p is currently visiting node n , and its *youngest* field is $\mathbf{null}$, then every other node m coming before n in the traverse order must be concurrent with p . More formally, let $youngest_s(p) = \mathbf{null}$ and assume there exists an operation $k \in O_t$ such that $n_s(k) \neq \mathbf{null}$, $n_s(k) \rightarrow taken = \mathbf{false}$ and $n_s(k) <_s^{tr} n_s(p)$. Then, $k \in ov_t(p)$.

TraverseAfter If a pop operation p is currently visiting node n that is not null and its youngest element m is not null and still not taken in state s , then either m is a candidate to be removed by p in t or there exists a later node m' than n such that m' is a candidate in t and it has a bigger timestamp than n . More formally, assume that there exists $k, k' \in O_t$ such that $youngest_s(p) \rightarrow taken \neq \mathbf{false}$, $youngest_s(p) = n_s(k)$ and $n_s(k') = n_s(p)$. Then, either $k \in ov_t(p) \vee k \in be_t(p)$ or there exists $k'' \in O_t$ s.t. $n_s(k'') \rightarrow ts > n_s(k) \rightarrow ts$ and $k'' \in ov_t(p) \vee k'' \in be_t(p)$ and either $k' <_s^{tr} k''$ or $n_s(p) = n_s(k'') \wedge cp_s(p) = R_j \wedge j < 5$.

Next, we will show that fs_2 is really a $C \cup R \cup Com(pop)$ -forward simulation relation. Except the trivial base case, we case-split on the transition rules. We first assume $s \xrightarrow{\alpha}_{TSS} s'$ and $t \in fs_2[s]$. Then, we find corresponding transition $\alpha' \in \Sigma_{AbsS}$ obeying the $C \cup R \cup Com(pop)$ -forward simulation relation conditions and obtain t' such that $t \xrightarrow{\alpha'}_{AbsS} st$ and $t' \in fs_2[s']$.

We observe that if $\alpha \in C \cup R \cup Com(pop)$, then the corresponding rule in $AbsS$ is $\alpha' = \alpha$. Otherwise, $\alpha' = \epsilon$.

Let the following describe α : $\psi \triangleright s \xrightarrow{\alpha}_{TSS} s'$ where ψ is the precondition (guard) that needs to be satisfied for enabling α and $\psi' \triangleright t \xrightarrow{\alpha'}_{AbsS} t'$ describe the α' if $\alpha' \neq \epsilon$

(equivalently $\alpha' = \alpha$).

For the cases $\alpha' = \alpha$, we first need to show α' is enabled in state t i.e., t satisfies ψ' . If this can not be directly obtained from the information that s satisfies ψ and using one or two obvious conditions on fs_2 (since $t \in fs_2[s]$), we show the derivation in the proof. Then, t' is obtained in a unique way since $AbsS$ is deterministic on its alphabet $\Sigma_{AbsS} = C \cup R \cup Com(pop)$. The, only other thing to show is $t' \in fs_2[t']$. We show this by proving that t' does not violate any of the conditions of the fs_2 described above. Suppose conditions on fs_2 are of the form $\forall \bar{k}. guard_{s,t}(\bar{k}) \triangleright \phi_{s,t}(\bar{k})$ where the $\bar{k}$ is a vector of operation identifiers and ϕ defined on states s and t must hold if the guard defined on s and t holds. We say that a vector $\bar{k}_1$ is a new instantiation of the condition if $\bar{k}_1$ does not satisfy the $guard_{s,t}$ while relating pre-states, but it satisfies $guard_{s',t'}$ while relating post-states.

We only explain why the new instantiations due to the difference between s' and s or the difference between t' and t do not violate the conditions. We skip the instances that we assumed while relating s to t .

For the cases in which $\alpha' = \epsilon$, we have $t' = t$ and the only thing to show is $t \in fs_2[s']$. Again, we only explain why the new instantiations due to the difference between s' and s do not violate the conditions.

In the following, we show that fs_2 is a $C \cup R \cup Com(pop)$ -forward simulation relation.

$$\text{INIT } fs_2[q_{0TSS}] = \{q_{0AbsS}\}$$

CALL-PUSH The same derivation rule of TSS is applied to t to obtain t' . The premise of the rule is satisfied by t trivially in the sense explained before. The new vertex k is added to the O_t such that k is maximal, pending and every completed vertex is ordered before k in t' . Moreover, k is overlapping with every pending pop. To see that $t' \in fs_2[s']$ we observe the following: *Nodes* condition is preserved because $k \in O_{t'}$. Since the newly added vertex k is maximal and pending in t' , *Pend/Comp* condition is preserved. *Frontiers* and *MaximalOV* conditions are

not violated since k is added to $ov(p)$ set for every pending pop operation p .

PUSH1 We have $t' = t$ and show $t \in fs_2[s']$. *Nodes* and *Pend/Comp* conditions are still satisfied since k remains to be a pending vertex. *TSTOrder* is still preserved. Timestamp of $n_{s'}(k)$ is maximal and every other nodes of push operations with maximal timestamp in s' are pending vertices in t . Hence there can be no ordering between those pushes and k in t that can violate *TSTOrder*. Moreover, k is maximal in t which means that it cannot be ordered before another push k' of which node has a lower timestamp. *TidOrder* is also satisfied. Since k is ordered after every completed push in t and every other push by the same thread is completed, ordering required by the *TidOrder* is present.

PUSH2 We have $t' = t$ and show $t \in fs_2[s']$. *Nodes* and *Pend/Comp* conditions are still satisfied since k remains to be a pending vertex. One can also see that the *TraverseBefore* condition is preserved. Let the pop p visiting node m and $n_{s'}(k) <_{s'}^{tr} m$. Since k and p are both pending in s and $t \in fs_2[s]$, $k \in ov_t(p)$ (by the *Frontiers* condition). Hence, *TraverseBefore* is preserved.

PUSH3 We have $t' = t$ and show $t \in fs_2[s']$. We consider two cases: $n_s(k) \rightarrow taken$ is **true** or it is **false**. For the former case, $k \notin O_t$. The only new instantiation we check is $k \notin O_t$ does not violate *Nodes* condition while relating s' to t .

For the latter case, we have $k \in O_t$. *Nodes* and *Pend/Comp* conditions are still satisfied since k remains to be a pending vertex after changing s to s' .

PUSH4 We have $t' = t$ and show $t \in fs_2[s']$.

We consider two cases: $n_s(k) \rightarrow taken$ is **true** or it is **false**. For the former case, *Nodes* condition is still satisfied since k remains to be not a vertex.

For the latter case *Nodes* and *Pend/Comp* conditions are still satisfied since k remains to be a pending vertex. *TSTOrder* condition is still not violated since if $k' <_t k$, then k' is a completed vertex in s and s' . By the premise of the rule

(which can be shown to hold for every operation at control point A_4) $i_s(k') < i_s(k)$ and consequently $n_{s'}(k') \rightarrow ts < n_{s'}(k) \rightarrow ts$. Since every other push by the thread of k is completed, *TidOrder* still continues to hold for the same reasons. *TraverseAfter* condition is also preserved. Let k' be the push and p be the pop such that $n_s(k') = \text{youngest}_s(p)$, $n_s(k') \leq_s^{tr} n_s(k)$, $n_s(k') \rightarrow ts < n_s(k) \rightarrow ts$ and $k \in \text{ov}_t(p)$ or $k \in \text{be}_t(p)$. Assume $n_{s'}(k') \rightarrow ts \geq N_{s'}(k) \rightarrow ts$ after the action. Then, k' must be a pending push both in s and s' by the premise of the derivation rule and $k' \in \text{ov}_t(p)$ must be true by *Frontiers* condition and $t \in fs_2[s]$. Hence, the *TraverseAfter* condition is preserved.

RET-PUSH We consider two cases, $n_s(k) \rightarrow \text{taken}$ is **false** or **true**. For the former case, we obtain t' by applying RET-PUSH1 rule of *AbsS*. *Nodes* and *Pend/Comp* conditions are still satisfied since k becomes a completed vertex in t' . *Frontiers* condition still holds since although k become a maximally closed vertex in t' , we have $k \in \text{ov}_{t'}(p)$ for all pending nodes p (due to *Frontiers* condition, $t \in fs_2[s]$ and k was a pending operation in state t , $k \in \text{ov}_t(p)$).

For the latter case, we obtain t' by applying RET-PUSH2 rule of *AbsS*. *Nodes* condition is still satisfied since $k \notin O_{t'}$.

CALL-POP The same derivation rule of *TSS* is applied to t to obtain t' . *Frontiers* condition holds for $p = k$ relating s' to t' since $k' \in \text{ov}_{t'}(k)$ for every pending vertex k' and $k'' \in \text{be}_{t'}(p)$ for all completed vertex k'' . t' due to action $\text{inv}(\text{pop}, k)$ applied on t . *MaximalOV* condition holds for $p = k$ since pending vertices are maximal in t' and for any maximally closed vertex k' in t' , if k' is ordered before other vertex k'' , then k'' is a pending operation by definition of being maximally closed and $k'' \in \text{ov}_t(k)$ due to the changes by INV-POP action on t . *MinimalBE* condition holds while relating s' to t' for the pop $p = k$ because only maximally closed vertices are in $\text{be}(k)$ and if a push k' is ordered before a maximally closed push k'' in t , neither $k'' \in \text{be}_{t'}(k)$ (since k'' is not maximally closed) nor $k'' \in \text{ov}_{t'}$ (since k'' cannot be pending). *ReverseFrontiers* condition holds while relating

s' to t' for the pop $p = k$ because, if $k'' \in ov_t(k)$ for all immediate successors of k' in t , then k'' are pending vertices (due to *call-pop* action of *AbsS*), k' is a maximally closed vertex and $k' \in be_t(k)$ (due to *call-pop* action of *AbsS*).

POP1 We have $t' = t$ and $t \in fs_2[s']$.

POP2 We have $t' = t$ and show $t \in fs_2[s']$. *TraverseBefore* condition while relating s' to t still holds for $p = k$. Assume $youngest_{s'}(k) = y$ is a non-null node. Then, for all nodes m in s' such that $n_s(k) \leq_{s'}^{tr} m <_{s'}^{tr} n_{s'}(k)$ we have $m \rightarrow ts < y \rightarrow ts$ in s' because $n_s(k) \rightarrow ts > m \rightarrow ts$ (since $n_s(k)$ is added to the pool after m by the same thread) and $y \rightarrow ts \geq n_s(k) \rightarrow ts$ in s' (since either $youngest_{s'}(k) = n_s(k)$ or $youngest_{s'}(k) \rightarrow ts > n_s(k) \rightarrow ts$). *TraverseAfter* does not have any new instantiations since the guard mentions the nodes after $n_s(k)$ while relating s to t whereas it mentions nodes after or including $n_{s'}(k)$ which contains the all nodes in the former case.

POP3 We have $t' = t$ and $t \in fs_2[s']$.

POP4 We have $t' = t$ and $t \in fs_2[s']$.

POP5 We have $t' = t$ and show $t \in fs_2[s']$. *TraverseBefore* condition while relating s' to t still holds for $p = k$ since $youngest_s(k) \rightarrow ts < youngest_{s'}(k) \rightarrow ts$ and *TraverseBefore* holds while relating s to t .

TraverseAfter condition also continues to hold for $p = k$. There are two possible cases: $youngest_s(k) = \text{null}$ or not.

First, consider the former case. Since *TraverseBeforeNull* is satisfied while relating s to t , for every operation $k', k'' \in O_t$ such that $k'' <_s^{tr} k'$ and $n_s(k') = youngest_{s'}(k)$ we have $k'' \in ov_t(k)$. Consider all such k'' such that $n_s(k'') \rightarrow ts > n_s(k') \rightarrow ts$. If there exists such a k'' such that $k' \in pred_{<_t}(k'')$, then $k' \in ov_t(k) \cup be_t(k)$ since *ReverseFrontiers* condition holds relating s to t . Otherwise, either k' is maximal in t or all the vertices v ordered after k' in t we have $v >_s^{tr} k'$.

Then, either k' or one of these v vertices must be maximal in t and must be in $be_t(k) \cup ov_t(k)$ since *Frontiers* condition holds (one of them is maximal in t) while relating s to t .

Second, assume there exists push operations j, k' such that $n_s(j) = youngest_s(k) \neq \text{null}$ and $n_s(k') = n_s(k) = youngest_{s'}(k)$. Since *TraverseBefore* is satisfied while relating s to t , if there exists a push $k'' <_s^{tr} k'$ such that $n_s(k'')$ is not taken and $n_s(k'') \rightarrow ts \geq n_s(j) \rightarrow ts$, then $k'' \in ov_t(k)$. Then, for all $k'' <_s^{tr} k'$ such that $n_s(k'')$ is not taken and $n_s(k'') \rightarrow ts \geq n_s(k') \rightarrow ts$, then $k'' \in ov_t(k)$ since $n_s(k') \rightarrow ts \geq n_s(j) \rightarrow ts$. If there exists such a k'' such that $k' \in pred_{<_t}(k'')$, then $k' \in ov_t(k) \cup be_t(k)$ since *ReverseFrontiers* condition holds relating s to t . Otherwise, either k' is maximal in t or all the vertices v ordered after k' in t we have $v >_s^{tr} k'$. Then, either k' or one of these v vertices must be maximal in t and must be in $be_t(k) \cup ov_t(k)$ since *Frontiers* condition holds (one of them is maximal in t) while relating s to t .

POP6 We have $t' = t$ and show $t \in fs_2[s']$. *TraverseAfter* continues to hold while relating s' to t for $p = k$. Let $k', k'' \in O_t$ such that $youngest_s(k) = n_s(k')$, $n_s(k) = n_s(k'')$ and $k' \notin ov_t(k) \cup be_t(k)$. Note that $k' <_s^{tr} k''$. Then, $n_s(k'') \rightarrow ts < n_s(k') \rightarrow ts$ since $n_s(k'') \rightarrow ts < maxTS(k)$ and $maxTS(k) = n_s(k') \rightarrow TS$ ($n_s(k') \rightarrow ts$ cannot be MAXINT since k' would be pending and $k' \in ov_t(k)$ otherwise). Hence, there exists another push j such that $j >_s^{tr}$ and $j \in ov_t(k) \cup be_t(k)$.

POP7 We have $t' = t$ and $t \in fs_2[s']$.

POP8 We have $t' = t$ and $t \in fs_2[s']$.

COM-POP t' is obtained by applying COM-POP1 rule of *AbsS*. We first show that precondition of COM-POP1 rule of *AbsS* is satisfied by t . If $com(pop, d, k)$ removes a node n such that there exists a push k' such that $n_s(k') = n$ in s , then $k' \in O_t$

since it is non-null and not taken. Moreover, $k' \in ov_t(k) \cup be_t(k)$ since *TraverseAfter* is preserved while relating s to t and all the nodes that come after $n_s(k)$ in terms of traverse order in s have lower timestamp values than $n_s(k) \rightarrow ts$ and $n_s(k) \rightarrow ts \leq youngest_s(k) \rightarrow ts$.

Next, we show that $t' \in fs_2[s']$. We case split on the conditions of fs_2 considering new instantiations.

Nodes condition is still preserved after k removes the node pushed by operation k' in s since $k' \notin O_{t'}$ anymore by due to $com(pop, d, k)$ action.

Frontiers condition is still preserved if k removes the vertex k' and makes another k'' maximally closed in t . Since all the other nodes j ordered after k'' (except possibly k') in t are pending, $j \in ov_t(p)$ (due to *Frontiers* condition while relating s to t) for some pending pop $p \neq k$. Then, $k'' \in be_{t'}(p)$ by $com(pop, d, k)$ action.

For the *MinimalBE* condition, we do not have a new instance. If $k' \in be_{t'}(p)$ becomes true although $k' \notin be_t(p)$, we cannot have $k'' \in O_{t'}$ such that $k' \in pred_{<_{t'}}(k'')$ and $k'' \in be_{t'}(p)$ since $com(pop, d, k)$ does not add k'' to $ov(p)$ if its successor is not pending with respect to p .

ReverseFrontiers condition is still preserved. If k removes the vertex k' and there exists an immediate predecessor k'' of k' such that all of immediate successors of k'' are in $ov_{t'}(p)$, then $k'' \in ov_{t'}(p)$ due to the action $com(pop, d, k)$.

TraverseAfter condition is still preserved after k removes the node of push k' . Let $p \neq k$ be another pop operation such that $n_s(j) = youngest_s(p)$ for some push j and $n_s(k')$ be the only node such that $n_s(k') \rightarrow ts > youngest_s(p) \rightarrow ts$ and $n_s(k')$ comes after $n_s(p)$ in the traverse order of s and $k' \in ov_t(p) \cup be_t(p)$. Hence, there is no k'' such that $n_s(k'')$ comes after $n_s(p)$ in the traverse order and $j <_t k''$ except k' (i). In other direction, if for all $k'' \in O_t$ such that $n_s(k'')$ comes before $n_s(p)$ in the traverse order and $n_s(k'') \rightarrow ts > youngest_s(p) \rightarrow ts$, then $k'' \in ov_t(p)$ since *TraverseBefore* condition holds while relating s to t .

Then, for all $k'' \in O_t$ such that $n_s(k'')$ comes before $n_s(p)$ in the traverse order of s and $k'' >_t j$ implies $k'' \in ov_t(p)$ since $n_s(k'') \rightarrow ts > n_s(j) \rightarrow ts$ if $k'' >_t j$ (ii). Then, for all $k'' \in O_t$ such that if $k'' >_t j$, then $k'' \in ov_t(p)$ except k' due to (i) and (ii). If $j \not<_t k'$, then $j \in ov_t(p) \cup be_t(p)$ since *ReverseFrontiers* hold while relating s to t and $j \in ov_{t'} \cup be_{t'}$ after applying the action $com(pop, d, k)$. Otherwise, if $j <_t k'$, then $k \in be_{t'}$ after applying $com(pop, d, k)$.

FixReturn condition continues to hold. If $com(pop, d, k)$ removes the node pushed by k' in s , then $com(pop, d, k)$ removes the vertex k' (assuming data independence) and $youngest_s(k') \rightarrow data = \ell_t(k')_1$. Then, $youngest_{s'}(p) \rightarrow data = rv_{t'}(p)$ after applying commit actions at both sides.

POP9 We have $t' = t$ and $t \in fs_2[s']$.

RET-POP t' is obtained by applying RET-POP rule of *AbsS* and $t' \in fs_2[s']$.

□

Chapter 3

A CORRECTNESS PROOF OF CHASE-LEV WSQ BY ABSTRACTION AND REDUCTION

3.1 Introduction

Work stealing is a widely accepted and applied method for scheduling tasks used by many programming languages, run-time systems and frameworks that support distribution of computation into tasks in shared-memory parallel programs. Work stealing queue data structures constitute the core of this method. The queue keeps a pool of tasks to be executed and provides methods to threads for putting and taking tasks from the pool. The WSQ algorithm must provide certain guarantees such as the same task cannot be scheduled twice or given sufficient number of requests, all the tasks in the pool are scheduled. These guarantees are vital for the correct functioning of the system.

In this study, we verify the Chase-Lev WSQ algorithm [22], a widely-used non-blocking algorithm, by providing a linearizability proof for its sequentially consistent (SC) executions. Starting with fine-grained concurrent method bodies, we obtain atomic method abstractions. Those abstractions are tight enough to show that the WSQ algorithm satisfies the desired properties.

The proof is performed using the CIVL proof system [31] and it has four layers¹. At the bottom layer, method bodies consist of fine-grained atomic statements supported by most hardware and programming languages. In the following layers, atomic blocks inside the method bodies grow using abstraction, reduction and location annotations, until we obtain the desired abstract atomic bodies of the methods at the fourth layer

¹CIVL proof files can be obtained from: <http://msrc.ku.edu.tr/projects/chase-lev-wsq/>.

(depicted in the top rows of Figures 3.4 and 3.5).

This study has the following contributions and results:

- We present the first mechanized linearizability proof of the Chase-Lev work stealing queue algorithm for its SC executions using a proof system.
- Obtaining correct location and mover annotations for the fine-grained method bodies require reasoning about all possible interleavings of the program. We believe that the proof annotations at lower layers provide insight about the behavior of SC executions of this program.
- Our proof is based on two important techniques: Owicki-Gries [56] and Lipton’s reduction/abstraction [44] method. We show that the combined use of these two techniques is powerful, and each is best suited to carry out certain parts of the reasoning.

Section 3.2 gives an overview of the Chase-Lev work stealing queue algorithm. Some observations on the WSQ algorithm that will be useful for our proofs are made in Section 3.3. Initial abstractions/simplifications on the algorithm applied before the mechanized proof are explained in Section 3.4. In Section 3.5, we give a brief overview of the proof techniques we utilized. Details of the mechanized proof are presented in Section 3.6. We compare our work with related studies in Section 3.7 and finish with closing remarks and future work in Section ??.

3.2 The Chase-Lev Work Stealing Queue Algorithm

A CIVL implementation of the Chase-Lev WSQ algorithm is presented in Figure 3.1.

Shared variables H , T and $items$ represent the current head (top), tail (bottom) and the task pool, respectively. The data structure offers three methods that can be called by its clients. The `put` and `take` methods can be called exclusively by a special unique thread called worker thread (or `ptTid` for short). On the other hand, other threads (called stealer threads) may call the `steal` method concurrently.

<pre> H:int; T:int; items:[int]int; put(task:int) { var t:int; 1 t := T; 2 items[t] := task; 3 T := t+1; 4 return; } </pre>	<pre> take(): (task:int) { var h,t:int; var chk:bool; 1 t := T 1; 2 T := t; 3 h := H; 4 if(t<h) { 5 T := h; 6 task := EMPTY; 7 return; } 8 task := items[t]; 9 if(h<t) { 10 return; } 11 [if(h==H) { H := h+1; chk := true; } else { chk := false; }] 12 if(!chk) { 13 task := EMPTY; } 14 T := t+1; 15 return; } </pre>	<pre> steal(): (task:int) { var h,t:int; var chk:bool; 1 while(true) { 2 h := H; 3 t := T; 4 if(h>=t) { 5 task := EMPTY; 6 return; } 7 task := items[h]; 8 [if (h==H) { H := h+1; chk := true; } else { chk := false; }]] 9 if(chk) { 10 return; } 11 return; } </pre>
---	---	--

Figure 3.1: Chase-Lev Work Stealing Queue Algorithm

The **put** method adds one more element to the tail of the queue.

The **take** method tries to remove an element from the tail. First, **take** reserves the last element by decrementing T by one (Line 2). If it observes an empty queue, then it increments T back and returns an **EMPTY** task (if block at Line 4). If it observes more than one elements in the queue, it returns the element at index T (if block at Line 9). If there is a single element in the queue, the **ptTid** tries to take it by a **CAS** operation on H (Line 11). **CAS** operation takes a shared variable, an old value and a new value for this variable. If the current value of the shared variable is the old value, it assigns the new value to the shared variable and returns true, otherwise it does not modify the shared variable and returns false. **CAS** is executed atomically.

The **steal** method is executed by stealer threads. If it sees the queue empty, it returns **EMPTY** task (if block at Line 4). Otherwise, it iteratively tries to steal an element by incrementing H by one via a **CAS** statement. If **CAS** is successful, then **steal** returns successfully with the element at index H (if block at Line 9). If **CAS** is not successful, then the current element at index h is stolen or taken. Hence, **steal** tries to steal another element in a new iteration.

For this implementation, we have made some assumptions. Tasks are assumed to be of type *int*. The *items* has an infinite domain. It is never required to resize or reuse the old cells *items* due to an overflow. Hence, we omitted the parts of the implementation that resizes the array or implements the queue in a circular fashion.

Since SC allows numerous interleavings among worker and stealer threads, reasoning about the program correctness is difficult. We present some observations that will help us to infer program invariants and annotations in the next section.

3.3 Observations on the SC Executions of the Program

In this section, we present some observations on execution fragments of the WSQ algorithm with the finest-grained actions (the algorithm in Figure 1). They will be helpful for obtaining location annotations and enlarging atomic blocks in upper layers of the proof.

Our initial observations are simple.

Observation 1: H is non-decreasing throughout an execution.

Observation 2: Let us call a **steal** operation successful if it returns a value other than **EMPTY**. Then, last iterations of two successful **steals** cannot be concurrent.

Next, we want to understand the relation between global variables H and T . For a sequential execution, one expects that $H \leq T$ is an invariant throughout the execution. This is not true for a fine-grained SC execution. We observe that H could exceed T in some special cases. However, this violations occur temporarily if the **take** method executes some paths and they begin to hold again after **take** method finishes.

We examine execution fragments in a systematic way to obtain observations showing the relation between H and T .

Observation 3: If an execution fragment consists of only concurrent **steal** and **put** operations, we observe that $H \leq T$ is preserved throughout the execution fragment.

Next, we consider execution fragments that have the **take** method executed concurrently with **steal** operations. The **take** method could follow three different paths by either entering the if block at Line 4 (path 1), by entering the if block at Line 9 (path 2) or by not entering those if blocks and returning by Line 15 (path 3).

Observation 4: If an execution fragment consists of path 1 of the take method and concurrent steals, then $H \leq T$ holds before Line 2 and after Line 5 of the **take** method and $H = T + 1$ holds between Lines 2 and 5 of the **take** method. This path is followed if $H = T$ just before execution of Line 2 of the take method.

Observation 5: If an execution fragment consists of path 2 of take method concurrent with steals, then $H \leq T$ holds throughout this execution fragment. This path is followed if $H < T$ just before execution of Line 3 of take method.

Observation 6: If an execution fragment consists of path 3 of the take method concurrent with steals, then $H \leq T$ holds before Line 2 and after Line 14 of the **take** method and $H = T \vee H = T + 1$ holds between Lines 2 and 14 of **take** method. This path is followed if $H = T$ just before execution of Line 3 of take method.

<pre> steal(lnr tid:Tid): (task:int) { var h,t:int; var chk:bool; 1 while(*) { 2 h := H; 3 t := T; 4 assume h<t; 5 task := items[h]; 6 [assume h != H; chk := false;] } 7 h := H; 8 t := T; 9 if(h>=t) { 10 task := EMPTY; 11 return; } 12 task := items[H]; 13 [assume h==H; H == h;] 14 return; } </pre>	<pre> steal(lnr tid:Tid): (task:int) { var h,t:int; var chk:bool; 1 while(*) { 2 havoc h,t,chk,task; } 3 h := H; 4 t := T; 5 if(h>=t) { 6 task := EMPTY; 7 return; } 8 task := items[H]; 9 [assume h==H; H == h;] 10 return; } </pre>	<pre> steal(lnr tid:Tid): (task:int) { var h,t:int; var chk:bool; 1 h := H; 2 t := T; 3 if(h>=t) { 4 task := EMPTY; 5 return; } 6 task := items[H]; 7 [assume h==H; H == h;] 8 return; } </pre>
--	---	--

Figure 3.2: Initial abstractions on **steal** method

3.4 Path Splitting and Loop Peeling

In this section we present our initial abstractions on **steal** and **take** methods. These abstractions are not performed by CIVL. Rather, they are obtained by applying some proof rules that are not currently supported by CIVL. We explain the rules and their applications in this section. The methods we obtained after initial abstractions constitute the bottom layer of our mechanized proof.

Our first abstraction is performed on the **steal** method. If we consider the iterations of the loop at Line 1 of **steal** before the last iteration, they do not modify any global variable and the value assigned to the return variable is reset by the last iteration. Moreover, the values assigned to local variables by reading global variables are also reread by the last iteration before using them. Hence, those iterations do not affect the behaviors of other concurrent methods and future behavior of the current steal method. Our aim is to abstract the **steal** method so that we do not need to

deal with unsuccessful previous iterations of the **steal**.

On the left of Figure 3.2, we have the **steal** method obtained by peeling out the last iteration of the **while** loop. All the unsuccessful iterations are captured by the loop at Line 1. The $*$ denotes a non-deterministic value of **true** or **false**. The iterations are guaranteed to be unsuccessful by the **assume** statements at lines 4 and 6. The last successful iteration is modeled from Line 7 on. Statements inside brackets are assumed to be atomic.

Lines 2, 3, 5 and 6 could be abstracted by *havoc* $h, t, task, chk$ and line 4 can be abstracted by *skip*. With these abstractions, we obtain the method body in the middle column of Figure 3.2. Since h, t, chk and $task$ variables have non-deterministic values at the beginning of **steal**, the whole loop at Line 1 could be removed and we obtain the method at the right of Figure 3.2 as our basis of **steal** for the mechanized proof.

For the **take** method, we have different observations for the different paths **take** can go. We want to separate **take** in such a way that we can reason about each possible path separately. We use the following rule for this purpose:

Rule 1: Let procedure *foo* have the following body:

```
{ s0; s1; s2; }
```

where **s0** and **s2** are sequence of statements and **s1** is an atomic statement block. Then, replacing this body with the following one is a valid abstraction of *foo*:

```
{ if(*){ s0; [s1;assume p;] s2; }
  else { s0;[s1;assume !p;] s2;} }
```

where p is a boolean expression on local variables. To obtain the desired method body in Figure 3.3, we apply the following steps:

1. Apply Rule 1 to **take** in Figure 3.1 with taking $s0, s1, s2$ as lines 1,2; 3 and 4-15, respectively. We also pick p as $h > t$.
2. Collect statements inside the if block in the method **take1** and statements in the else part in the method **take23**.

<pre> take(lnr tid:Tid) :(task:int){ var h,t:int; var chk:bool; 1 if(*){ 2 task := take1(tid); 3 } 4 else{ 5 if(*){ 6 task := take2(tid); 7 } 8 else{ 9 task := take3(tid); 10 } 11 } 12 return; 13 } </pre>	<pre> take1(lnr tid:Tid) :(task:int){ var h,t:int; var chk:bool; 1 t := T 1; 2 T := t; 3 [h := H; <u>assume</u> t<h;] 4 if(t<h){ 5 T := h; 6 task := EMPTY; 7 return; 8 } 9 task := items[t]; 10 if(h<t){ 11 return; 12 } 13 [if(h==H){ 14 H := h+1; 15 chk := true; 16 }else{ 17 chk := false; 18 }] 19 if(!chk){ 20 task := EMPTY; 21 } 22 T := t+1; 23 return; 24 } </pre>
<pre> take2(lnr tid:Tid) :(task:int){ var h,t:int; var chk:bool; 1 t := T 1; 2 T := t; 3 [h := H; <u>assume</u> t>=h; <u>assume</u> t != h;] 4 if(t<h){ 5 T := h; 6 task := EMPTY; 7 return; 8 } 9 task := items[t]; 10 if(h<t){ 11 return; 12 } 13 [if(h==H){ 14 H := h+1; 15 chk := true; 16 }else{ 17 chk := false; 18 }] 19 if(!chk){ 20 task := EMPTY; 21 } 22 T := t+1; 23 return; 24 } </pre>	<pre> take3(lnr tid:Tid) :(task:int){ var h,t:int; var chk:bool; 1 t := T 1; 2 T := t; 3 [h := H; <u>assume</u> t>=h; <u>assume</u> t==h;] 4 if(t<h){ 5 T := h; 6 task := EMPTY; 7 return; 8 } 9 task := items[t]; 10 if(h<t){ 11 return; 12 } 13 [if(h==H){ 14 H := h+1; 15 chk := true; 16 }else{ 17 chk := false; 18 }] 19 if(!chk){ 20 task := EMPTY; 21 } 22 T := t+1; 23 return; 24 } </pre>

Figure 3.3: Initial abstractions on take method

3. Apply Rule 1 to `take23` with the same line choice as in step 1 but taking p as $h = t$.
4. Collect the statements inside the if block of `take23` in `take2` method and statements in the else part in `take3` method.
5. Inline call of `take23` inside the else block of `take` with its body.

Consequently, we decompose `take` method into three other methods each representing a different path `take` could follow.

3.5 Overview of Proof Methodology

In this section, we provide some important techniques we utilize in our proof. These techniques are supported by the proof system CIVL. We only give high-level definitions and explain how we utilized them. Formal definitions of the concepts described here can be found in [26] and [31].

The language CIVL is specially developed for verification purposes. It allows usual constructs existing in many imperative programming languages and some additional constructs for verification purposes. A CIVL program consists of method bodies and atomic actions. Method bodies contain usual imperative constructs like assignments, sequencing, conditional statements, loops, method calls, thread creation and atomic action calls. An atomic action consists of a location annotation (precondition of the action) followed by a two-state transition relation representing the statement.

The method bodies are partitioned into steps. CIVL allows programmers to decide on the granularity of a step in a method body. Hence, a step may contain multiple statements. Atomic actions are single step. We denote steps inside brackets in this chapter. A program executes by picking a thread non-deterministically and executing a non-deterministic number of next steps of this thread. An SC execution is obtained by interleaving steps from different threads. If the location annotation of an atomic action does not hold in a state of an execution just before executing this action, the

program fails. The program is safe if no execution fails the location annotation of an atomic action.

To check safety of a program, CIVL utilizes Owicki-Gries (OG) reasoning [56]. The OG checks two things: i) a location annotation holds after a thread takes a step (sequential correctness) and ii) the location annotation is preserved by concurrent threads (non-interference). Let P be a program with two concurrent methods each executing a single atomic statement. Assume $\langle P_1 \rangle S_1 \langle Q_1 \rangle$ be the first method and $\langle P_2 \rangle S_2 \langle Q_2 \rangle$ is the second method where P_i and Q_i are location annotations and S_i are statements. To prove correctness of location annotations, the following checks are performed: Q_i holds after executing S_i from any state P_i holds (sequential correctness), $P_i (Q_i)$ holds after executing S_j from any state $P_i \wedge P_j$ ($Q_i \wedge P_j$) holds for all $i \neq j$ (non-interference).

In addition to the location annotations, CIVL allows programmers to write method pre- and post-conditions. They are also checked via OG reasoning. Moreover, programmers can write conditions to be satisfied inside the atomic blocks as assert statements. Correctness of these statements are checked again by OG reasoning without the non-interference part.

CIVL enables programmers to grow atomic steps inside the procedure bodies using a technique called reduction. To achieve this, each atomic action is annotated with R, L, N or B tags standing for right-, left-, non- and both-movers. A sequence of steps that begins with a sub-sequence of right- or both-movers, followed by an optional non-mover, followed by a sub-sequence of left-movers or both-movers could form a single atomic step. CIVL also performs a check to validate that the atomic action conforms to its mover type. An action A is a right-mover if executing A first and then an action B from another thread in all executions can be simulated by first executing B and then A . A dual definition applies for left-movers. An action is a both-mover (non-mover) if it is both (neither) right-mover and (nor) left-mover.

A layer in CIVL is a program and CIVL performs refinement proofs in a sequence of layers. While moving from one layer to the next layer, CIVL allows programmers

to abstract atomic actions so that they have more behavior in the next layer. Hence, they have more relaxed location annotations and they can make actions from other threads mover and grow the steps of method bodies in the next layer. For instance, *havoc* is a keyword in CIVL, used for assigning non-deterministic values to variables. *havoc* x abstracts $x := t$ since the former statement allows variable x to have a range of values including the latter value t .

Another option that a programmer can benefit from between layers is the method abstraction. It allows programmers to replace a method body with a single atomic action. This method enables programmers to increase granularity of the program by replacing fine-grained method bodies with a single coarser action block. CIVL also performs a check between layers to validate method abstraction. An atomic action A abstracts a method body B if and only if A abstracts a single step of B in all possible execution paths of B and all the other steps of B in this execution path refines *skip*. Moreover, the return variable must not be modified after the step that refines A . From now on, we call the step that refines A as the action block.

CIVL validates that the program at the bottom layer refines the program at the top-layer if the OG and mover checks pass for all layers and action abstraction and method abstraction checks pass between all layers.

CIVL allows the programmers to use primitive types like boolean or integer and allow users to define their own types. Moreover, it supports linear variables. The difference of a linear variable from a regular variable is that the value inside a linear variable cannot be duplicated. Since thread identifiers are unique, we use linear thread identifiers. We modify methods such that each thread gets a linear variable *tid* as input. For the methods **take** and **put**, we know that *tid* must have the value **ptTid** and for **steal** method, *tid* must be different than **ptTid**. This additional information makes CIVL know that **put** and **take** methods cannot be concurrent whereas **steal** can be concurrent with other methods and itself.

We intensively utilize the techniques above in our proof. In all layers, we provide location annotations that show the relation between the global variables H and T .

Those annotations play a crucial role during the method abstractions between layers and mover checks inside the layers. We start with a relatively complicated relation between those global variables in fine-grained method bodies of lower layers. As the methods get coarser at later layers, we establish $H \leq T$ as a global invariant.

3.6 Mechanized Proof Steps

Our proof in CIVL contains four layers. At the end of Layer 0, we abstract some of the atomic actions so that steps inside the method bodies of **put** and sub-methods of **take** could grow bigger via reduction. Consequently, method bodies of **put** and sub-methods of **take** contain single steps at the end of Layer 1. Then, we can turn them into atomic actions in Layer 2 by applying method abstraction. Since all of the sub-methods of **take** turn into atomic actions at Layer 2, we can use method abstraction at the end of Layer 2 to obtain the desired atomic action of **take** at Layer 3. Moreover, $H \leq T$ becomes a global location invariant at Layer 2 and it enables us to use method abstraction at the end of Layer 2 on the **steal** method to obtain desired atomic action of **steal** at Layer 3.

A schematic of the proof is given in Figures 3.4 and 3.5. Before diving into the details of the proof, we provide a brief explanation about the programs in these figures. Numbers inside the method bodies correspond to steps or control points of those methods. Atomic actions are written inside brackets. We may omit the brackets if the atomic action consists of a single statement. Location annotations of atomic actions are given between $\langle$ and $\rangle$ symbols. If a location annotation has not changed since the previous layer, we denote this as $\langle \dots \rangle$ or if it is tightened by adding a new constraint ϕ , we denote this as $\langle \dots \wedge \phi \rangle$.

Mover annotations of the atomic actions are also present in brackets before atomic actions. R, L, B and N denote right-, left-, both- and none-mover, respectively. We may omit the mover tag if the atomic action is labeled as non-mover. Note that we may have labeled an action non-mover although it is a mover, if it is not necessary for our proof.

Layer	steal(linear tid:Tid):(task:int)	put(linear tid:tid, task:int)
3	<pre> steal(linear tid:Tid):(task:int) [goto 1A, 1B; 1A: assume H<T; task := items[H]; H := H+1; return; 1B: assume H<=T; task := EMPTY; return;] </pre>	<pre> put(linear tid:tid, task:int) {ptInv()} [items[T] := task; T := T+1;] </pre>
2	<pre> pre: (stTid(tid) ∧ ticsCond() ∧ !tics) post: (stTid(tid) ∧ ticsCond() ∧ !tics) { 1 <... ∧ !tics> h := H; 2 <... ∧ !tics> [if (h<T) assume h==H ==> H<T; else assume h>=t;] 3 if (h >= t) { task := EMPTY; } 4 <... ∧ !tics> return; 5 <... ∧ !tics> [assert !tics && h==H ==> H<T; if (h==H) task := items[h]; else havoc(task);] 6 <... ∧ !tics ∧ (H=h → task = items[h])> [assume H == h; H := h+1;] 7 <... ∧ !tics> return; } </pre>	<pre> {ptInv()} [items[T] := task; T := T+1;] </pre>
1	<pre> pre: (stTid(tid) ∧ ticsCond()) post: (stTid(tid) ∧ ticsCond()) { 1 <...> [N] h := H; 2 <...> [N] [if (h<T) assume h==H ==> H<T; else assume h>=t;] 3 if (h >= t) { [B] task := EMPTY; } 4 <...> return; 5 <...> [N] [assert !tics && h==H ==> H<T; if (h==H) task := items[h]; else havoc task;] 6 <...> [N] [assume H == h; H := h+1;] 7 <...> return; } </pre>	<pre> pre: {ptInv()} post: {ptInv()} { {ptInv()} 1 [R] t := T; 2 [R] [assert t==T && !tics; items[t] := task;] 3 [N] T := t+1; 4 {ptInv()} return; } </pre>
0	<pre> pre: (stTid(tid) ∧ ticsCond()) post: (stTid(tid) ∧ ticsCond()) { 1 <ticsCond()> h := H; 2 <H>=h ∧ ticsCond()> t := T; 3 if (h >= t) { <ticsCond()> task := EMPTY; } 4 return; 5 <H>=h ∧ ticsCond() ∧ ticsCond2(h)> task := items[h]; 6 <H>=h ∧ ticsCond() ∧ ticsCond2(h)> [assume H == h; H := h+1;] 7 <ticsCond()> return } ticsCond(): (tics ⇒ H ≤ T+1) ∧ (!tics ⇒ H ≤ T) ticsCond2(h:int): (tics ∧ h==H ⇒ H ≤ T) ∧ (!tics ∧ h==H ⇒ H < T) stTid(tid:Tid): tid ≠ NULL ∧ tid ≠ ptTid </pre>	<pre> pre: {ptInv()} post: {ptInv()} { 1 {ptInv()} t := T; 2 {ptInv()} items[t] := task; 3 {ptInv()} T := t+1; 4 {ptInv()} return; } ptInv(): tid = ptTid ∧ !tics ∧ H ≤ T </pre>

Figure 3.4: Proof layers for the mechanized proof of steal and put methods

Layer	take(linear tid:Tid):(task:int)		
3	<pre> <ptInv()> [goto 1A, 1B, 1C; 1A: assume H==T; task := EMPTY; return; 1B: assume H==T 1; task := items[T 1]; H := H+1; return; 1C: assume H<T 1; T := T 1; task := items[T]; return;]</pre>		
2	<pre> take1(lnr tid:Tid): (task:int) <ptInv()> [assume H==T; task := EMPTY;]</pre>	<pre> take2(lnr tid:Tid): (task:int) <ptInv()> [assume H < T 1; T := T 1; task := items[T];]</pre>	<pre> take3(lnr tid:Tid): (task:int) <ptInv()> [goto 1A,1B; 1A: assume H==T; task := EMPTY; return; 1B: assume H==T 1; task := items[T 1]; H := H+1; return;]</pre>
1	<pre> pre: <ptInv()> post: <ptInv()> { <ptInv()> 1 [R] t := T 1; 2 [R] [T := t; tics := true;] 3 [R] assume h <= H && t < h; 4 if (t < h) { 5 [L] [T := h; tics := false;] 6 [B] task := EMPTY; 7 <ptInv()> return; }... }</pre>	<pre> pre: <ptInv()> post: <ptInv()> { <ptInv()> 1 [R] t := T 1; 2 [R] [T := t; tics := true;] 3 [N] [h := H; assume h < t; tics := false;] 4 if (t < h) 8 [B] task := items[t]; 9 if (h < t) { 10 <ptInv()> return; }... }</pre>	<pre> pre: ~<ptInv()> post: <ptInv()> { <ptInv()> 1 [R] t := T 1; 2 [R] [T := t; tics := true;] 3 [R] assume h==t && h<=H; 4 if (t < h) 8 [B] task := items[t]; 9 if (h < t) 11 [N] [if (h==H) H := h+1; chk := true; else chk := false;] 12 if (!chk) { 13 [B] task := EMPTY; } 14 [L] [T := t+1; tics := false;] 15 <ptInv()> return; }</pre>
0	<pre> pre: <ptInv()> post: <ptInv()> { 1 <ptInv()> t := T 1; 2 <ptInv() & t=T-1> [T := t; tics := true;] 3 <t=T & H≤T+1 & tics> [h := H; assume t < h;] 4 if (t < h) { 5 <t=T & h≤H & H=T+1 & tics> [T := h; tics := false;] 6 <H≤T & !tics> task := EMPTY; 7 <H≤T & !tics> return; }... }</pre>	<pre> pre: <ptInv()> post: <ptInv()> { 1 <ptInv()> t := T 1; 2 <ptInv() & t=T-1> [T := t; tics := true;] 3 <t=T & H≤T+1 & tics> [h := H; assume h < t; tics := false;] 4 if (t < h) 8 <t=T & H≤T & h < t & !tics> task := items[t]; 9 if (h < t) { 10 <H≤T & !tics> return; }... }</pre>	<pre> pre: ~<ptInv()> post: <ptInv()> { 1 <ptInv()> t := T 1; 2 <ptInv() & t=T-1> [T := t; tics := true;] 3 <t=T & H≤T+1 & tics> [h := H; assume h==t;] 4 if (t < h) 8 <h≤H & H≤T+1 & h=t & tics> task := items[t]; 9 if (h < t) 11 <h≤H & H≤T+1 & h=t & tics> [if (h==H) H := h+1; chk := true; else chk := false;] 12 if (!chk) { 13 <t=T & H=T+1 & tics> task := EMPTY; } 14 <t=T & H=T+1 & tics> [T := t+1; tics := false;] 15 <ptInv()> return; }</pre>

Figure 3.5: Proof layers for the mechanized proof of take methods

Atomic actions or statements may contain constructs like **assume**, **assert** and **havoc** which are special for verification. Semantics of **havoc** is explained via an example in Section 3. The statements **assume e** or **assert e** cause an execution to block (all threads' next statement to execute becomes not enabled) or fail, respectively, if the boolean expression **e** evaluates to **false** in the state just before executing this statement. Otherwise, they are equivalent to **skip**.

For the **take1**, **take2** and **take3** methods some paths are unreachable due to **assume** statements at Line 3. We omit the program text for the if blocks leading to those paths in Figure 3 by representing them with three dots as in Line 4 of **take2** at Layer 0.

The programs contain a new boolean ghost variable named **tics**. A ghost variable is similar to a regular variable with only difference that it does not modify the program state i.e., its value is never assigned to a real program variable. Its sole purpose is to guide CIVL during mover and OG checks.

The name **tics** is short for "take in critical section". We know from Observations 3-6 in Section 3.3 that $H \leq T$ can be temporarily violated inside the **take** method. We say that **take** is in critical section if execution of **take** is in the area that $H \leq T$ invariant can be violated. The **tics** is used to write location annotations considering the current instruction of **ptTid**.

At the bottom layer (Layer 0), we start with the method bodies that we obtained at the end of Section 3.4. We decorate the method bodies with location annotations to establish relation between H and T global variables. While going from Layer 0 to Layer 1, we abstract some of the actions of Layer 0 and we use reduction at Layer 1 to make bodies of the **put**, **take1**, **take2** and **take3** methods single step. Between Layer 1 and Layer 2, we use method abstraction on **put**, **take1**, **take2** and **take3** methods and abstract them to single step atomic actions. In Layer 2, $H \leq T$ begins to hold as a global location annotation since **take1**, **take2** and **take3** methods become coarse enough. Finally, we apply method abstraction on **take** and **steal** methods between Layer 2 and Layer 3 to obtain desired atomic actions for these methods. In Layer 3,

all the methods of the WSQ algorithm are in the form of atomic actions.

Layer 0 We begin with stating location annotations. Based on the observations in Section 3.3 and the meaning of `tics` explained before, we provide annotations relating `tics` and global variables H and T . We annotate the locations such that when `tics` is *true*, $H \leq T + 1$ holds and when `tics` is *false*, $H \leq T$ holds. This condition is expressed in the sub-formulae called `ticsCond` provided after `steal` method. The condition when the `tics` is *true*, is adjusted in precision. It is tight enough to continue proof in later layers and relaxed enough to be satisfied after non-interference checks. For instance, replacing it with $H \equiv T + 1$ would be too tight and make it unsatisfiable.

$H \leq T$ is not a global invariant at Layer 0, but a relaxed version of it that we call `ticsCond` is a global invariant. All the location annotations in the method bodies and method pre- post-conditions imply `ticsCond`.

However, location annotations explained so far does not pass the OG check directly. We need to make location annotations stronger.

First, if $H = T \wedge !tics$ or $H = T + 1 \wedge tics$ holds at some state during execution of a `ptTid` method, a successful `CAS` operation of a `steal` could interfere and violate location annotation by incrementing H . We observe that this corner case is not possible in real executions. $H < T$ must hold if $!tics$ or $H < T + 1$ must hold if `tics` just before the execution of `CAS` action of `steal`. For this reason, we introduce `ticsCond2` function that reflects our condition and add it to the annotations of Lines 6 and 7 of `steal`.

Adding `ticsCond2` to location annotations of Lines 6 and 7 is still insufficient because two concurrent steals may violate the `ticsCond2`. If $h = H - 1 \wedge H = T - 1 \wedge !tics$ holds just before a stealer thread t_1 executes Line 6 and another stealer thread t_2 interferes at this point and performs a successful `CAS` and increments H , we come up with a state satisfying $h = H \wedge H = T \wedge !tics$ for t_2 which violates the `ticsCond2`. But we know that two successful `steals` cannot be concurrent by Observation 2. This observation helps us to infer that $H \geq h$ holds during Lines 6 and 7 of the `steal` method (by Observation 1) which prevents the previous erroneous

execution sample. By adding $H \geq h$ on Lines 6 and 7 of **steal**, we make sure that OG checks for location annotations of **steal** pass.

Second, methods of the WSQ modify global variables H and T by assigning them local values of h and t . To show that these assignments do not violate the conditions relating H and T , we need to relate local value t to value of T in **ptTid** methods and h to H in **steal**, **take1**, **take2** and **take3** methods. Since T is only modified by **ptTid**, adding $t = T$ to location annotations of **ptTid** methods is correct. By Observation 1, H is always non-decreasing. Hence $H \geq h$ holds for all methods. Adding these two conditions to the location annotations of certain lines is sufficient to show that modifications on global variables do not violate the required conditions.

We omit the mover tags for this layer, since no reduction is performed at Layer 0.

Layer 0 $\rightarrow$ Layer 1 Our aim for Layer 1 is to grow steps of **take1**, **take2**, **take3** and **put** method bodies using reduction. For this reason, we abstract some of the atomic actions between Layer 0 and Layer 1. Lines 2 and 6 of **steal** method and Line 3 of **take1** and **take3** methods are abstracted for this purpose. Rationale behind these abstractions are explained in Layer 1 while explaining how the actions satisfy their mover annotations.

Layer 1 We assign mover tags to atomic actions of **put**, **take1**, **take2** and **take3** methods so that we can grow the steps of the action blocks of these methods as large as the ones we need.

First, let us explain how we grow the step of **put** method. Line 1 becomes right-mover without any abstraction since it reads the global variable T which is not modified by other threads. However, Line 2 of **put** method is not a right-mover without abstraction since it may be modifying an index of *items* that is read by Line 6 of **steal**. If h value of **steal** at Line 6 is equal to t value of **put** at Line 3, they may be accessing the same index. Since action of **put** method modifies this index, mover check fails. But, we observe that the actual value of *items*[h] is not needed if $h \neq H$ holds just before execution of Line 7 of **steal** method. Moreover, we need to read

the actual value of $items[h]$ if $H = h$ at Line 7 in order to know that **steal** returns the correct element. Hence, we abstracted Line 6 of **steal** such that it assigns $items[h]$ to $task$ if $h = H$ at Line 6 and assigns a non-deterministic value to $task$ otherwise.

Next, we enlarge the step of **take1** method. Line 1 is a right-mover since it is same as Line 1 of **put** method and Line 5 is a both-mover since it is a local assignment. However, lines 2 and 6 are not movers in Layer 0. They do not commute with Line 2 of **steal** method since Line 2 of **steal** reads the global variable T which is modified by Line 2 (or Line 6) of **take1** method. To overcome this problem, let us explore what we expect from Line 2 of **steal** method in our proof. If **steal** observes $h < t$ after execution of Line 2 and the value of H has not changed yet, then **ticsCond2** must hold if it continues execution through Line 6. We may satisfy this condition by assuming only $H < T$ holds after Line 2 if $h = H \wedge h < T$ holds just before Line 2. **steal** may continue through the if block at Line 3 if it observes $h < t$ before Line 2, since our top-level implementation of **steal** allows it to return **EMPTY** even if $H < T$. But, if **steal** observes $h \geq T$ before Line 2, it must enter the if block at Line 3 since the WSQ is empty. Obtaining only this information after Line 2 would be sufficient for abstracting **steal** on later layers.

Line 3 of **take1** is not a right-mover at Layer 0 since it does not commute with Line 7 of **steal** method. Instead of reading actual value of H at Line 3 of **take1**, we abstract to read a non-deterministic value less than or equal to H . This abstract read can commute right of Line 7 of **steal** since after moving right of Line 7 of **steal** local value h of **take1** can have more distinct values and it is tight enough to infer that $h \geq t$. Consequently, we obtain a step for **take1** that spans lines through 1 to 6.

Lines 1 and 2 of **take2** are right-movers due to reasons explained above. Line 8 becomes a right-mover since **steal** does not modify the $items$ array. Consequently, lines from 1 to 8 of **take2** form a step in Layer 1.

Mover annotations for **take3** method also hold and lines from 1 to 14 become a step. Reason for their correctness can be explained with the same arguments above.

Note that location annotations of Layer 1 are same as Layer 0. Abstractions do not violate the conditions established at Layer 0.

Layer 1 $\rightarrow$ Layer 2 We apply method abstraction on `put`, `take1`, `take2` and `take3` methods between Layer 1 and Layer 2 since action blocks of those methods grow large enough at the end of Layer 1. We obtain the desired atomic action for `put` method.

Layer 2 In this layer, we tighten the location annotations in `steal` method such that they become the old condition and `!tics`. The OG checks pass for these new tighter annotations because no step of the program leaves the `tics` *true* after its execution at Layer 2.

In addition, we add a condition to location annotation of Line 7 of `steal` stating that if the value of H had not changed since `steal` read it, the return variable `task` contains `items[H]`. OG check for this condition passes since Line 6 of `steal` is tight enough to assign correct value to `task` if $H = h$. This extra condition on the location invariant is crucial when we apply method abstraction on `steal`.

Layer 2 $\rightarrow$ Layer 3 Between Layers 2 and 3, we apply method abstraction on `steal` and `take` methods. These methods become atomic actions.

The reason we can not apply method abstraction on `steal` so far is that location annotations of the action blocks of `steal` were not tight enough to obtain the desired atomic action for `steal`. It was possible to perform a successful CAS when $H = T$.

Applying method abstraction on `take` also becomes possible after Layer 2 since we obtained the atomic actions for its sub-methods at Layer 2.

Layer 3 All of the `take`, `put` and `steal` methods are atomic actions.

With these single atomic action bodies of the methods, it is easier to reason about the WSQ algorithm. For instance, one can show that a task pushed into deque cannot be taken or stolen more than once since `take` and `steal` methods' top-level actions atomically increment H or decrement T after taking the first or the last item from

the queue.

3.7 Related Work

Due to its key importance in parallel systems, there are various WSQ algorithms. A notable one is presented in Cilk multi-threaded language [28]. This algorithm is blocking and method bodies are protected by a global lock. Reasoning about correctness of Cilk WSQ algorithm is simpler but it is not efficient due to its blocking nature.

Another WSQ algorithm introduced by Arora et al [11] is non-blocking, but it requires fixed size queues. This algorithm has been verified in [38] using a model checking approach. Model checking approach validates that the algorithm satisfies desired properties but it does not provide any insight about the behavior of the algorithm. Hence, it is difficult to reason about some side-properties and possible optimizations of the algorithm using this approach. The Chase-Lev WSQ [22] we have studied is an improvement over [11] such that size of the queue can grow without memory leaks.

Since work stealing queues are used in low-level task schedulers, the semantics of low level architectures may provide weaker guarantees. It is known that executions of Chase-Lev WSQ under TSO semantics show more behaviour than SC executions [46]. If memory fences are inserted after Line 3 of `put` and Line 2 of `take`, non-SC behaviors are prevented [46]. In [42], a pen and pencil proof has been presented that the Chase-Lev WSQ algorithm with previously mentioned memory fences satisfy some desired specifications. A modified version of the Chase-Lev WSQ is presented in [50] such that it is correct under TSO memory-model if we know the size of store buffers.

In [60], they show that Arora WSQ algorithm is linearizable under TSO semantics i.e., it behaves as if all the method bodies execute atomically. This proof is also based on manual annotations of the program.

In [48], they propose idempotent work stealing queue, a weaker WSQ in which popping the same element multiple times is allowed. They introduce three new idem-

potent WSQ algorithms. It is simpler to show their correctness. Our approach can be easily applied for verifying idempotent WSQ algorithms. We leave this task as a future work.

There are other WSQ algorithms like [24, 35]. We have not encountered any study that proves their correctness in a formal mechanical way.



Chapter 4

REASONING ABOUT TSO PROGRAMS USING REDUCTION AND ABSTRACTION

4.1 *Introduction*

A classical memory model for shared-memory concurrency is Sequential Consistency [41] (SC), where the actions of different threads are interleaved while the program order between actions of each thread is preserved. For performance reasons, modern multi-processors implement weaker memory models, e.g., Total Store Ordering (TSO) [58] in x86 machines, which relax the program order. For instance, the main feature of TSO is the write-to-read relaxation, which allows reads to overtake writes. This relaxation reflects the fact that writes are buffered before being flushed non-deterministically to the main memory.

Nevertheless, most programmers usually assume that memory accesses happen instantaneously and atomically like in the SC memory model. This assumption is safe for data-race free programs [6]. However, many programs employing lock-free synchronization are not data-race free, e.g., programs implementing synchronization operations and libraries implementing concurrent objects. In most cases, these programs are designed to be robust against relaxations, i.e., they admit the same behaviors as if they were run under SC. Memory fences must be included appropriately in programs in order to prevent non-SC behaviors. Getting such programs right is a notoriously difficult and error-prone task. Robustness can also be used as a proof method, that allows to reuse the existing SC verification technology. Invariants of a robust program running under SC are also valid for the TSO executions. Therefore, the problem of checking robustness of a program against relaxations of a memory

model is important.

In this chapter, we address the problem of checking robustness in the case of TSO. We present a methodology for proving robustness which uses the concepts of left/right mover in Lipton’s reduction theory [45]. Intuitively, a program statement is a left (resp., right) mover if it commutes to the left (resp., right) with respect to the statements in the other threads. These concepts have been used by Lipton [45] to define a program rewriting technique which enlarges the atomic blocks in a given program while preserving the same set of behaviors. In essence, robustness can also be seen as an atomicity problem: every write statement corresponds to two events, inserting the write into the buffer and flushing the write from the buffer to the main memory, which must be proved to happen atomically, one after the other. However, differently from Lipton’s reduction theory, the events that must be proved atomic do not correspond syntactically to different statements in the program. This leads to different uses of these concepts which cannot be seen as a direct instantiation of this theory.

In case programs are not robust, or they cannot be proven robust using our method, we define a program abstraction technique that roughly, makes reads non-deterministic (this follows the idea of combining reduction and abstraction introduced in [26]). The non-determinism added by this abstraction can lead to programs which can be proven robust using our method. Then, any invariant (safety property) of the abstraction, which is valid under the SC semantics, is also valid for the TSO semantics of the original program. As shown in our experiments, this abstraction leads in some cases to programs which reach exactly the same set of configurations as the original program (but these configurations can be reached in different orders), which implies no loss of precision.

We tested the applicability of the proposed reduction and abstraction based techniques on an exhaustive benchmark suite containing 34 challenging programs (from [3] and [13]). These techniques were precise enough for proving robustness of 32 of these programs. One program (presented in Figure 3.1) is not robust, and required ab-

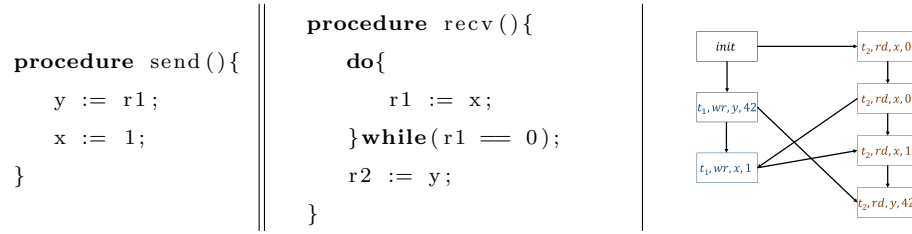


Figure 4.1: An example message passing program and a sample trace. Edges of the trace shows the happens before order of shared accesses and they are simplified by applying transitive reduction.

straction in order to derive a robust over-approximation. There is only one program which cannot be proved robust using our techniques (although it is robust). We believe however that an extension of our abstraction mechanism to atomic read-write instructions will be able to deal with this case. We leave this question for future work.

4.2 Overview

The TSO memory model allows strictly more behaviors than the classic SC memory model: writes are first stored in a thread-local buffer and non-deterministically flushed into the shared memory at a later time (also, the write buffers are accessed first when reading a shared variable). However, in practice, many programs are *robust*, i.e., they have exactly the same behaviors under TSO and SC. Robustness implies for instance, that any invariant proved under the SC semantics is also an invariant under the TSO semantics. We describe in the following a sound methodology for checking that a program is *robust*, which avoids modeling and verifying TSO behaviors. Moreover, for non-robust programs, we show an abstraction mechanism that allows to obtain robust programs over-approximating the behaviors of the original program.

As a first example, consider the simple “message passing” program in Figure 4.1. The **send** method sets the value of the “communication” variable y to some predefined value from register $r1$. Then, it raises a flag by setting the variable x to 1. Another thread executes the method **recv** which waits until the flag is set and then, it reads y (and stores the value to register $r2$). This program is robust, TSO doesn’t enable new

behaviors although the writes may be delayed. For instance, consider the following TSO execution (we assume that $r1 = 42$):

$$\begin{array}{lll}
 (t_1, isu) & (t_1, isu)(t_1, com, y, 42) & (t_1, com, x, 1) \\
 (t_2, rd, x, 0) & (t_2, rd, x, 0) & (t_2, rd, x, 1)(t_2, rd, y, 42)
 \end{array}$$

The actions of each thread (t_1 or t_2) are aligned horizontally, they are either *issue* actions (*isu*) for writes being inserted into the local buffer (e.g., the first (t_1, isu) represents the write of y being inserted to the buffer), *commit* actions (*com*) for writes being flushed to the main memory (e.g., $(t_1, com, y, 42)$ represents the write $y := 42$ being flushed and executed on the shared memory), and *read* actions for reading values of shared variables. Every assignment generates two actions, an issue and a commit. The issue action is “local”, it doesn’t enable or disable actions of other threads.

The above execution can be “mimicked” by an SC execution. If we had not performed the *isu* actions of t_1 that early but delayed them until just before their corresponding *com* actions, we would obtain a valid SC execution of the same program with no need to use store buffers:

$$\begin{array}{lll}
 & (t_1, wr, y, 42) & (t_1, wr, x, 1) \\
 (t_2, rd, x, 0) & (t_2, rd, x, 0) & (t_2, rd, x, 1)(t_2, rd, y, 42)
 \end{array}$$

Above, consecutive *isu* and *com* actions are combined into a single *write* action (*wr*). This intuition corresponds to an equivalence relation between TSO executions and SC executions: if both executions contain the same actions on the shared variables (performing the same accesses on the same variables with the same values) and the order of actions on the same variable are the same for both executions, we say that these executions have the same *trace* [59], or that they are *trace-equivalent*. For instance, both the SC and TSO executions given above have the same trace given in Figure 4.1. The notion of trace is used to formalize robustness for programs running under TSO [13]: a program is called *robust* when every TSO execution has the same trace as an SC execution.

Our method for showing robustness is based on proving that every TSO execution

<pre> procedure foo(){ x := 1; r1 := z; fence r2 := y; } </pre>		<pre> procedure bar(){ y := 1; fence r3 := x; } </pre>
---	--	--

Figure 4.2: An example store buffering program.

can be permuted to a trace-equivalent SC execution (where issue actions are immediately followed by the corresponding commit actions). We say that an action α moves right until another action β in an execution if we can swap α with every later action until β while preserving the feasibility of the execution (e.g., not invalidating reads and keeping the actions enabled). We observe that if α moves right until β then the execution obtained by moving α just before β has the same trace with the initial execution. We also have the dual notion of moves-left with a similar property. As a corollary, if every issue action moves right until the corresponding commit action or every commit action moves left until the corresponding issue action, we can find an equivalent SC execution. For our execution above, the issue actions of the first thread move right until their corresponding *com* actions. Note that there is a commit action which doesn't move left: moving $(t_1, \text{com}, x, 1)$ to the left of $(t_2, \text{rd}, x, 0)$ is not possible since it would disable this read.

In general, issue actions and other thread local actions (e.g. statements using local registers only) move right of other threads' actions. Moreover, issue actions (t, isu) move right of commit actions of the same thread that correspond to writes issued before (t, isu) . For the message passing program, the issue actions move right until their corresponding commits in all TSO executions since commits cannot be delayed beyond actions of the same thread (for instance reads). Hence, we can safely deduce that the message passing program is robust. However, this reasoning may fail when an assignment is followed by a read of a shared variable in the same thread.

Consider the “store-buffering” like program in Figure 4.2. This program is also robust. However, the issue action generated by $x := 1$ might not always move

right until the corresponding commit. Consider the following execution (we assume that initially, $z = 5$):

$$\begin{array}{ccccccc}
 (t_1, isu) & & (t_1, rd, z, 5) & & & & (t_1, com, x, 1) \dots \\
 & (t_2, isu) & & (t_2, com, y, 1) & (t_2, \tau) & (t_2, rd, x, 0) & \dots
 \end{array}$$

Here, we assumed that t_1 executes **foo** and t_2 executes **bar**. The **fence** instruction generates an action τ . The first issue action of t_1 cannot be moved to the right until the corresponding commit action since this would violate the program order. Moreover, the corresponding commit action does not move left due to the read action of t_2 on x (which would become infeasible).

The key point here is that a later read action by the same thread, $(t_1, rd, z, 5)$, doesn't allow to move the issue action to the right (until the commit). However, this read action moves to the right of other threads actions. So, we can construct an equivalent SC execution by first moving the read action right after the commit $(t_1, com, x, 1)$ and then move the issue action right until the commit action.

In general, we say that an issue (t, isu) of a thread t moves right until the corresponding commit if each read action of t after (t, isu) can move right until the next action of t that follows both the read and the commit. Actually, this property is not required for all such reads. The read actions that follow a fence cannot happen between the issue and the corresponding commit actions. For instance, the last read action of **foo** cannot happen between the first issue of **foo** and its corresponding commit action. Such reads that follow a fence are not required to move right. In addition, we can omit the right-moves check for read actions that read from the thread local buffer (see Section 4.3 for more details).

In brief, our method for checking robustness does the following for every write instruction (assignment to a shared variable): either the commit action of this write moves left or the actions of later read instructions that come before a fence move right in all executions. This semantic condition can be checked using the concept of movers [44] as follows: every write instruction is either a left-mover or all the read instructions that come before a fence and can be executed later than the write (in an

SC execution) are right-movers. Note that this requires no modeling and verification of TSO executions.

For non-robust programs that might reach different configurations under TSO than under SC, we define an abstraction mechanism that replaces read instructions with “non-deterministic” reads that can read more values than the original instructions. The abstracted program has more behaviors than the original one (under both SC and TSO), but it may turn to be robust. When it is robust, we get that any property of its SC semantics holds also for the TSO semantics of the original program.

Consider the work stealing queue implementation in Figure 3.1. This program is not robust. If there is a single element in the queue and the **take** method takes it by delaying its writes after some concurrent **steals**, one of the concurrent **steals** might also remove this last element. Popping the same element twice is not possible under SC, but it is possible under TSO semantics. However, we can still prove some properties of this program under TSO. Our robustness check fails on this program because the writes of the worker thread (executing the **put** and **take** methods) are not left movers and the read from the variable H in the **take** method is not a right mover. This read is not a right mover w.r.t. successful CAS actions of the **steal** procedure that increment H .

We apply an abstraction on the instruction of the **take** method that reads from H (Line 3) such that instead of reading the exact value of H , it can read any value less than or equal to the value of H . We write this instruction as **havoc**($\mathbf{h}, \mathbf{h} \leq H$) (it assigns to $\mathbf{h}$ a nondeterministic value satisfying the constraint $\mathbf{h} \leq H$). Note that this abstraction is sound in the sense that it reaches more states under SC/TSO than the original program.

The resulting program is robust. The statement **havoc**($\mathbf{h}, \mathbf{h} \leq H$) is a right mover w.r.t. successful CAS actions of the stealer threads. Hence, for all the write instructions, the reachable read instructions become right movers and our check succeeds. The abstract program satisfies the specification of an idempotent work stealing queue (elements can be dequeued multiple times) which implies that the original program

satisfies this specification as well.

4.3 TSO Robustness

We present the syntax and the semantics of a simple programming language used to state our results. We define both the TSO and the SC semantics, an abstraction of executions called *trace* [59] that intuitively, captures the happens-before relation between actions in an execution, and the notion of robustness.

Syntax. We consider a simple programming language which is defined in Figure 4.3. Each program $\mathcal{P}$ has a finite number of shared variables $\vec{x}$ and a finite number of threads $(\vec{t})$. Also, each thread t_i has a finite set of local registers $(\vec{r}_i)$ and a start label l_i^0 . Bodies of the threads are defined as finite sequences of labelled instructions. Each instruction is followed by a **goto** statement which defines the evolution of the program counter. Note that multiple instructions can be assigned to the same label which allows us to write non-deterministic programs and multiple **goto** statements can direct the control to the same label which allows us to mimic imperative constructs like loops and conditionals. An assignment to a shared variable $\langle var \rangle := \langle expr \rangle$ is called a *write instruction*. Also, an instruction of the form $\langle reg \rangle := \langle var \rangle$ is called a *read instruction*.

Instructions can read from or write to shared variables or registers. Each instruction accesses at most one shared variable. We assume that the program $\mathcal{P}$ comes with a domain $\mathcal{D}$ of values that are stored in variables and registers, and a set of functions $\mathcal{F}$ used to calculate arithmetic and boolean expressions. We also assume that there exists a special element $0 \in \mathcal{D}$ that represents the initial default value of registers and shared variables.

The **fence** statement empties the buffer of the executing thread. The **cas** (compare-and-swap) instruction checks whether the value of its input variable is equal to its second argument. If so, it writes sets third argument as the value of the variable and returns *true*. Otherwise, it returns *false*. In either case, **cas** empties the buffer immediately after it executes. The **assume** statement allows us to check conditions.

$$\begin{aligned}
\langle prog \rangle &::= \text{program } \langle pid \rangle \text{ vars } \langle var \rangle^* \langle thread \rangle^* \\
\langle thread \rangle &::= \text{thread } \langle tid \rangle \text{ regs } \langle reg \rangle^* \text{ init } \langle label \rangle \text{ begin } \langle linst \rangle^* \text{ end} \\
\langle linst \rangle &::= \langle label \rangle : \langle inst \rangle ; \text{ goto } \langle label \rangle ; \\
\langle inst \rangle &::= \langle var \rangle := \langle expr \rangle \\
&| \quad \langle reg \rangle := \langle expr \rangle \\
&| \quad \langle reg \rangle := \langle var \rangle \\
&| \quad \text{fence} \\
&| \quad \langle reg \rangle := \text{cas}(\langle var \rangle, \langle expr \rangle, \langle expr \rangle) \\
&| \quad \text{skip} \\
&| \quad \text{assume } \langle bexpr \rangle
\end{aligned}$$

Figure 4.3: Syntax of the programs. The star (*) indicates zero or more occurrences of the preceding element. $\langle pid \rangle$, $\langle tid \rangle$, $\langle var \rangle$, $\langle reg \rangle$ and $\langle label \rangle$ are elements of their given domains representing the program identifiers, thread identifiers, shared variables, registers and instruction labels, respectively. $\langle expr \rangle$ is an arithmetic expression over $\langle reg \rangle^*$. Similarly, $\langle bexpr \rangle$ is a boolean expression over $\langle reg \rangle^*$.

If the boolean expression it contains holds at that state, it behaves like a **skip**. Otherwise, the execution blocks. Formal description of the instructions are given in Figure 4.4.

TSO Semantics. Under the TSO memory model, each thread maintains a local queue to buffer write instructions. A state s of the program is a triple of the form (pc, mem, buf) . Let $\mathcal{L}$ be the set of available labels in the program $\mathcal{P}$. Then, $pc : \vec{t} \rightarrow \mathcal{L}$ shows the next instruction to be executed for each thread, $mem : \bigcup_{t_i \in \vec{t}} \vec{r}_i \cup \vec{x} \rightarrow \mathcal{D}$ represents the current values in shared variables and registers and $buf : \vec{t} \rightarrow (\vec{x} \times \mathcal{D})^*$ represents the contents of the buffers.

There is a special initial state $s_0 = (pc_0, mem_0, buf_0)$. At the beginning, each thread t_i points to its initial label l_i^0 i.e., $pc_0(t_i) = l_i^0$. We assume that there is a special default value $0 \in \mathcal{D}$. All the shared variables and registers are initiated as 0 i.e., $mem_0(x) = 0$ for all $x \in \bigcup_{t_i \in \vec{t}} \vec{r}_i \cup \vec{x}$. Lastly, all the buffers are initially empty

$$\begin{array}{c}
\frac{x := ae(\vec{r}_t) \in ins(pc(t)) \quad v = eval(ae(\vec{r}_t)) \quad x \in \vec{x}}{(pc, mem, buf) \xrightarrow{(t, isu)}_{TSO} (pc', mem, buf[t \rightarrow buf(t) \circ \langle(x, v)\rangle])} \\
\\
\frac{buf(t) = \langle(x, v)\rangle \circ buf' \quad x \in \vec{x}}{(pc, mem, buf) \xrightarrow{(t, com, x, v)}_{TSO} (pc, mem, buf[t \rightarrow buf'])} \\
\\
\frac{r := ae(\vec{r}_t) \in ins(pc(t)) \quad v = eval(ae(\vec{r}_t)) \quad r \in \vec{r}_t}{(pc, mem, buf) \xrightarrow{(t, \tau)}_{TSO} (pc', mem[r \rightarrow v], buf)} \\
\\
\frac{r := x \in ins(pc(t)) \quad x \in \vec{x} \quad v = mem(x) \quad x \notin varsOfBuf(buf(t)) \quad r \in \vec{r}_t}{(pc, mem, buf) \xrightarrow{(t, rd, x, v)}_{TSO} (pc', mem[r \rightarrow v], buf)} \\
\\
\frac{r := x \in ins(pc(t)) \quad x \in \vec{x} \quad buf = \alpha \circ \langle(x, v)\rangle \circ \beta \quad x \notin varsOfBuf(\beta) \quad r \in \vec{r}_t}{(pc, mem, buf) \xrightarrow{(t, rd, x, v)}_{TSO} (pc', mem[r \rightarrow v], buf)} \\
\\
\frac{fence \in ins(pc(t)) \quad buf(t) = \epsilon}{(pc, mem, buf) \xrightarrow{(t, \tau)}_{TSO} (pc', mem, buf)} \\
\\
\frac{r := cas(x, ae_1(\vec{r}_t), ae_2(\vec{r}_t)) \in ins(pc(t)) \quad mem(x) = eval(ae_1(\vec{r}_t)) \quad buf(t) = \epsilon \quad v = eval(ae_2(\vec{r}_t))}{(pc, mem, buf) \xrightarrow{(t, isu)(t, com, x, v)}_{TSO} (pc', mem[r \rightarrow 1][x \rightarrow v], buf)} \\
\\
\frac{r := cas(x, ae_1(\vec{r}_t), ae_2(\vec{r}_t)) \in ins(pc(t)) \quad mem(x) \neq eval(ae_1(\vec{r}_t)) \quad buf(t) = \epsilon \quad v = mem(x)}{(pc, mem, buf) \xrightarrow{(t, rd, x, v)}_{TSO} (pc', mem[r \rightarrow 0], buf)} \\
\\
\frac{assume \ be(\vec{r}_t) \in ins(pc(t)) \quad eval(be(\vec{r}_t)) = \top}{(pc, mem, buf) \xrightarrow{(t, \tau)}_{TSO} (pc', mem, buf)}
\end{array}$$

Figure 4.4: The TSO Transition Relation. The function *ins* takes a label *l* and returns the set of instructions labeled by *l*. We always assume that $x \in \vec{x}$, $r \in \vec{r}_t$ and $pc' = pc[t \rightarrow l']$ where $pc(t) : inst \ goto \ l'$; is a labeled instruction of *t* and *inst* is the instruction described at the beginning of the rule. The evaluation function *eval* calculates the value of an arithmetic or boolean expression based on *mem* (*ae* stands for arithmetic expression). Sequence concatenation is denoted by $\circ$. The function *varsOfBuf* takes a sequence of pairs and returns the set consisting of the first fields of these pairs.

i.e., $buf_0(t_i) = \epsilon$ for all $t_i \in \vec{t}$.

The transition relation $\rightarrow_{TSO}$ between program states is defined in Figure 4.4. Transitions are labelled by actions. Each action is an element from $\vec{t} \times (\{\tau, isu\} \cup (\{com, rd\} \times \vec{x} \times \mathcal{D}))$. Actions keep the information about the thread performing the transition and the actual parameters of the reads and the writes to shared variables. We are only interested in accesses to shared variables, therefore, other transitions are labelled with τ as thread local actions.

A TSO execution of a program $\mathcal{P}$ is a sequence of actions $\pi = \pi_1, \pi_2, \dots, \pi_n$ such

that there exists a sequence of states $\sigma = \sigma_0, \sigma_1, \dots, \sigma_n$, $\sigma_0 = s_0$ is the initial state of $\mathcal{P}$ and $\sigma_{i-1} \xrightarrow{\pi_i} \sigma_i$ is a valid transition for any $i \in \{1, \dots, n\}$. We assume that buffers are empty at the end of the execution.

SC Semantics. Under SC, a program state is a pair of the form (pc, mem) where pc and mem are defined as above. Shared variables are read directly from the memory mem and every write updates directly the memory mem . To make the relationship between SC and TSO executions more obvious, every write instruction generates *isu* and *com* actions which follow one another in the execution (each *isu* is immediately followed by the corresponding *com*). Since there are no write buffers, **fence** instructions have no effect under SC.

Traces and TSO Robustness. Consider a (TSO or SC) execution π of $\mathcal{P}$. The trace of π is a graph, denoted by $Tr(\pi)$: Nodes of $Tr(\pi)$ are actions of π except the τ actions. In addition, *isu* and *com* actions are unified in a single node. The *isu* action that puts an element into the buffer and the corresponding *com* action that drains that element from the buffer correspond to the same node in the trace. There exists a special node *Init* that represents write actions that write the default value 0 to all the variables and registers before the program initiates.

Edges of $Tr(\pi)$ represent the happens before order (*hb*) between these actions. The *hb* is union of four relations. The program order *po* keeps the order of actions performed by the same thread excluding the *com* actions. *Init* happens before the first action of all the threads with a *po* edge. The store order *so* keeps the order of *com* actions on the same variable that write different values¹. The read-from relation, denoted by *rf*, relates a *com* action to a *rd* action that reads its value. Lastly, the from-reads relation *fr* relates a *rd* action to a *com* action that overwrites the value read by *rd*; it is defined as the composition of *rf* and *so*.

¹Our definition of store order deviates slightly from the standard definition which relates any two writes writing on the same variable, independently of values. The notion of TSO trace robustness induced by this change is slightly weaker than the original definition, but still implies preservation of any safety property from the SC semantics to the TSO semantics. The results concerning TSO robustness used in this chapter (Lemma 6) are also not affected by this change. See Appendices A and B for details.

We say that the program $\mathcal{P}$ is TSO robust if for any TSO execution π of $\mathcal{P}$, there exists an SC execution π' such that $Tr(\pi) = Tr(\pi')$. It has been proven that robustness implies that the program reaches the same valuations of the shared memory under both TSO and SC [13].

4.4 A Reduction Theory for Checking Robustness

We present a methodology for checking robustness which builds on concepts introduced in Lipton's reduction theory [44]. This theory allows to rewrite a given concurrent program (running under SC) into an equivalent one that has larger atomic blocks. Proving robustness is similar in spirit in the sense that one has to prove that issue and commit actions can happen together atomically. However, differently from the original theory, these actions do not correspond to different statements in the program (they are generated by the same write instruction). Nevertheless, we show that the concepts of left/right movers can be also used to prove robustness.

Movers. Let $\pi = \pi_1, \dots, \pi_n$ be an SC execution. We say that the action π_i *moves right* (resp., *left*) in π if the sequence $\pi_1, \dots, \pi_{i-1}, \pi_{i+1}, \pi_i, \pi_{i+2}, \dots, \pi_n$ (resp., $\pi_1, \dots, \pi_{i-2}, \pi_i, \pi_{i-1}, \pi_{i+1}, \dots, \pi_n$) is also a valid execution of $\mathcal{P}$, the thread of π_i is different than the thread of π_{i+1} (resp., π_{i-1}), and both executions reach to the same end state σ_n . Since every issue action is followed immediately by the corresponding commit action, an issue action moves right, resp., left, when the commit action also moves right, resp., left, and vice-versa.

Let instOf_π be a function, depending on an execution π , which given an action $\pi_i \in \pi$, gives the labelled instruction that generated π_i . Then, a labelled instruction ℓ is a *right* (resp., *left*) *mover* if for all SC executions π of $\mathcal{P}$ and for all actions π_i of π such that $\text{instOf}(\pi_i) = \ell$, π_i moves right (resp., left) in π .

A labelled instruction is a *non-mover* if it is neither left nor right mover, and it is a *both mover* if it is both left and right mover.

Reachability Between Instructions. An instruction ℓ' is *reachable from* the instruction ℓ if ℓ and ℓ' both belong to the same thread and there exists an SC execution

π and indices $1 \leq i < j \leq |\pi|$ such that $\text{instOf}_\pi(\pi_i) = \ell$ and $\text{instOf}_\pi(\pi_j) = \ell'$. We say that ℓ' is reachable from ℓ *before a fence* if π_k is not an action generated by a **fence** instruction in the same thread as ℓ , for all $i < k < j$. When ℓ is a write instruction and ℓ' a read instruction, we say that ℓ' is *buffer-free* reachable from ℓ if π_k is not an action generated by a **fence** instruction in the same thread as ℓ or a write action on the same variable that ℓ' reads-from, for all $i < k < j$.

Definition 7. We say that a write instruction ℓ_w is atomic if it is a left mover or every read instruction ℓ_r buffer-free reachable from ℓ_w is a right mover. We say that $\mathcal{P}$ is write atomic if every write instruction ℓ_w in $\mathcal{P}$ is atomic.

Note that all of the notions used to define write atomicity (movers and instruction reachability) are based on SC executions of the programs. The following result shows that write atomicity implies robustness.

Theorem 5 (Soundness). *If $\mathcal{P}$ is write atomic, then it is robust.*

We will prove the contrapositive of the statement. For the proof, we need the notion of a minimal violation and witness defined in [13]. A minimal violation is a TSO execution in which the sum of the number of same thread actions between *isu* and corresponding *com* actions for all writes is minimal. A witness minimal violation is of the form $\pi = \pi_1, (t, \text{isu}), \pi_2, (t, \text{rd}, y, *), \pi_3, (t, \text{com}, x, *), \pi_4$ such that π_1 is an SC execution, only t can delay *com* actions, the first delayed action is the $(t, \text{com}, x, *)$ action after π_3 and it corresponds to (t, isu) after π_1 , π_2 does not contain any *com* or *fence* actions by t (writes of t are delayed until after $(t, \text{rd}, y, *)$), $(t, \text{rd}, y, *) \rightarrow_{hb^+} \text{act}$ for all $\text{act} \in \pi_3 \circ \{(t, \text{com}, x, *)\}$ (*isu* and *com* actions of other threads are counted as one action for this case), π_3 doesn't contain any action of t , π_4 contains only and all of the *com* actions of t that are delayed in $(t, \text{isu}) \circ \pi_2$ and no *com* action in $(t, \text{com}, x, *) \circ \pi_4$ touches y .

Witness minimal violations are important for us because of the following property:

Lemma 6 (Soundness and Completeness of Minimal Violations [13]). *The program $\mathcal{P}$ is robust iff it does not have a witness minimal violation.*

Before going into the proof of Theorem 5, we define some notations. Let π be a sequence representing an execution or a fragment of it. Let Q be a set of thread identifiers. Then, $\pi|_Q$ is the projection of π on actions from the threads in Q . Similarly, $\pi|_n$ is the projection of π on first n elements for some number n . $sz(\pi)$ gives the length of the sequence π . We also define a product operator $\otimes$. Let π and ρ be some execution fragments. Then, $\pi \otimes \rho$ is same as π except that if the i^{th} *isu* action of π is not immediately followed by a *com* action by the same thread, then i^{th} *com* action of ρ is inserted after this *isu*. The product operator helps us to fill unfinished writes in one execution fragment by inserting commit actions from another fragment immediately after the issue actions.

Theorem 5. Assume $\mathcal{P}$ is not robust. Then, there exists a witness minimal violation $\pi = \pi_1, \alpha, \pi_2, \theta, \pi_3, \beta, \pi_4$ satisfying the conditions described before, where $\alpha = (t, isu)$, $\theta = (t, rd, y, *)$ and $\beta = (t, com, x, *)$. Below, we show that the write instruction $w = \text{instOf}(\alpha)$ is not atomic.

1. w is not a left mover.

1.1. $\rho = \pi_1, \pi_2|_{\vec{t} \setminus \{t\}}, \pi_3|_{\vec{t} \setminus \{t\}}|_{sz(\pi_3|_{\vec{t} \setminus \{t\}})-1}, \gamma, (\alpha, \beta)$ is an SC execution of $\mathcal{P}$ where γ is the last action of π_3 . γ is a read or write action on x performed by a thread t' other than t and value of γ is different from what is written by β .

1.1.1. ρ is an SC execution because t never changes value of a shared variable in π_2 and π_3 . So, even we remove actions of t in those parts, actions of other threads are still enabled. Since other threads perform only SC operations in π , $\pi_1, \pi_2|_{\vec{t} \setminus \{t\}}, \pi_3|_{\vec{t} \setminus \{t\}}$ is an SC execution. From π , we also know that the first enabled action of t is α if we delay the actions of t in π_2 and π_3 .

1.1.2. The last action of π_3 is γ . By definition of a witness minimal violation, we know that $\theta \rightarrow_{hb+} \alpha$ and π_3 does not contain any action of t . So, there must exist an action $\gamma \in \pi_3$ such that either γ reads from x and

$\gamma \rightarrow_{fr} \beta$ in π or γ writes to x and $\gamma \rightarrow_{st} \beta$ in π . Moreover, γ is the last action of π_3 because if there are other actions after γ , we can delete them and can obtain another minimal violation which is shorter than π and hence contradict the minimality of π .

1.2. $\rho' = \pi_1, \pi_2|_{\vec{t} \setminus \{t\}}, \pi_3|_{\vec{t} \setminus \{t\}}|_{sz(\pi_3|_{\vec{t} \setminus \{t\}})-1}, (\alpha, \beta), \gamma$ is an SC execution with a different end state than ρ defined in 1.1 has or it is not an SC execution, where $\text{instOf}(\gamma') = \text{instOf}(\gamma)$.

1.2.1. In the last state of ρ , x has the value written by β . If γ is a write action on x , then x has a different value at the end of ρ' due to the definition of a witness minimal violation. If γ is a read action on x , then it does not read the value written by β in ρ . However, γ reads this value in ρ' . Hence, ρ' is not a valid SC execution.

2. There exists a read instruction r buffer-free reachable from w such that r is not a right mover. We will consider two cases: Either there exists a rd action of t on variable z in π_2 such that there is a later write action by another thread t' on z in π_2 that writes a different value or not. Moreover, z is not a variable that is touched by the delayed commits in π_4 i.e., it does not read its value from the buffer.

2.1. We first evaluate the negation of above condition. Assume that for all actions γ and γ' such that γ occurs before γ' in π_2 , either $\gamma \neq (t, rd, z, v_z)$ or $\gamma' \neq (t', isu)(t', com, z, v'_z)$. Then, $r = \text{instOf}(\theta)$ is not a right mover and it is buffer-free reachable from w .

2.1.1. $\rho = \pi_1, \pi_2|_{\vec{t} \setminus \{t\}}, \pi_2|_{\{t\}} \otimes \pi_4, \theta, \theta'$ is a valid SC execution of $\mathcal{P}$ where $\theta' = (t', isu)(t', com, y, *)$ for some $t \neq t'$.

2.1.1.1. ρ is an SC execution. $\pi_1, \pi_2|_{\vec{t} \setminus \{t\}}$ is a valid SC execution since t does not update value of a shared variable in π_2 . Moreover, all of the actions of t become enabled after this sequence since t never

reads value of a variable updated by another thread in π_2 . Lastly, the first action of π_3 is enabled after this sequence.

2.1.1.2. The first action of π_3 is $\theta' = (t', isu)(t', com, y, *)$. Let θ' be the first action of π_3 . Since $\theta \rightarrow_{hb} \theta'$ in π and θ' is not an action of t by definition of a witness minimal violation, the only case we have is $\theta \rightarrow_{fr} \theta'$. Hence, θ' is a write action on y that writes a different value than θ reads.

2.1.1.3. r is buffer-free reachable from w . ρ is a SC execution, first action of ρ after $\pi_1, \pi_2|_{\vec{t} \setminus \{t\}}$ is α, β ; $w = \text{instOf}((\alpha, \beta))$, $r = \text{instOf}(\theta)$ and actions of t in ρ between α, β and θ are not instances of a fence instruction or write to y .

2.1.2. $\rho' = \pi_1, \pi_2|_{\vec{t} \setminus \{t\}}, \pi_2|_{\{t\}} \otimes \pi_4, \theta', \theta$ is not a valid SC execution.

2.1.2.1. In the last state of ρ , the value of y seen by t is the value read in θ . It is different than the value written by θ' . However, at the last state of ρ' , the value of y t sees must be the value θ' writes. Hence, ρ' is not a valid SC execution.

2.2. Assume that there exists $\gamma = (t, rd, z, v_z)$ and $\gamma' = (t', isu)(t', com, z, v'_z)$ in π_2 . Then, $r = \text{instOf}(\gamma)$ is not a right mover and r is buffer-free reachable from w .

2.2.1. Let i be the index of γ and j be the index of γ' in π_2 . Then, define

$$\rho = \pi_1, \pi_2|_{j-1}|_{\vec{t} \setminus \{t\}}, \pi_2|_i|_{\{t\}} \otimes \pi_4, \gamma'. \rho \text{ is an SC execution of } \mathcal{P}.$$

2.2.1.1. ρ is an SC execution. $\pi_1, \pi_2|_{j-1}|_{\vec{t} \setminus \{t\}}$ prefix is a valid SC execution because t does not update any shared variable in π_2 . Moreover, all of the actions of t in $\pi_2|_i|_{\{t\}} \otimes \pi_4$ become enabled after this sequence since t never reads a value of a variable updated by another thread in π_2 and γ' is the next enabled in π_2 after this sequence since it is a write action.

2.2.2. Let i and j be indices of γ and γ' in π_2 respectively. Define $\rho' = \pi_1, \pi_2|_{j-1}|_{\vec{t} \setminus \{t\}}, \pi_2|_{i-1}|_{\{t\}} \otimes \pi_4, \gamma', \gamma$. Then, ρ' is not a valid SC execu-

tion.

2.2.2.1. In the last state of ρ , value of z seen by t is v_z . It is different than the v'_z , value written by γ' . However, in the last state of ρ' , the value of z t sees must be v'_z . Hence, ρ' is not a valid SC execution.

2.2.3. r is buffer-free reachable from w because ρ defined in 2.2.1 is an SC execution, first action after $\pi_1, \pi_2|_{j-1}|\vec{t}\setminus\{t\}$ is α, β , $w = \text{instOf}((\alpha, \beta))$, $r = \text{instOf}(\gamma)$ and actions of t in ρ between α, β and θ are not instances of a fence instruction or a write to z by t .

□

4.5 Abstractions and Verifying non-Robust Programs

In this section, we introduce program abstractions which are useful for verifying non-robust TSO programs (or even robust programs – see an example at the end of this section). In general, a program $\mathcal{P}'$ abstracts another program $\mathcal{P}$ for some semantic model $\mathbb{M} \in \{\text{SC}, \text{TSO}\}$ if every shared variable valuation σ reachable from the initial state in an $\mathbb{M}$ execution of $\mathcal{P}$ is also reachable in an $\mathbb{M}$ execution of $\mathcal{P}'$. We denote this abstraction relation as $\mathcal{P} \preceq_{\mathbb{M}} \mathcal{P}'$.

In particular, we are interested in *read instruction abstractions*, which replace instructions that read from a shared variable with more “liberal” read instructions that can read more values (this way, the program may reach more shared variable valuations). We extend the program syntax in Section 4.3 with havoc instructions of the form $\text{havoc}(\langle \text{reg} \rangle, \langle \text{varbexpr} \rangle)$, where $\langle \text{varbexpr} \rangle$ is a boolean expression over a set of registers and a single shared variable $\langle \text{var} \rangle$. The meaning of this instruction is that the register reg is assigned with any value that satisfies varbexpr (where the other registers and the variable var are interpreted with their current values). The program abstraction we consider will replace read instructions of the form $\langle \text{reg} \rangle := \langle \text{var} \rangle$ with havoc instructions $\text{havoc}(\langle \text{reg} \rangle, \langle \text{varbexpr} \rangle)$.

While replacing read instructions with havoc instructions, we must guarantee that

the new program reaches at least the same set of shared variable valuations after executing the havoc as the original program after the read. Hence, we allow such a rewriting only when the boolean expression $varbexpr$ is weaker (in a logical sense) than the equality $reg = var$ (hence, there exists an execution of the havoc instruction where $reg = var$).

Lemma 7. *Let $\mathcal{P}$ be a program and $\mathcal{P}'$ be obtained from $\mathcal{P}$ by replacing an instruction $l_1 : x := r; \text{goto } l_2$ of a thread t with $l_1 : \text{havoc}(r, \phi(x, \vec{r})); \text{goto } l_2$ such that $\forall x, r. x = r \implies \phi(x, \vec{r})$ is valid. Then, $\mathcal{P} \preceq_{SC} \mathcal{P}'$ and $\mathcal{P} \preceq_{TSO} \mathcal{P}'$.*

The notion of trace extends to programs that contain havoc instructions as follows. Assume that $(t, hvc, x, \phi(x))$ is the action generated by an instruction $\text{havoc}(r, \phi(x, \vec{r}))$, where x is a shared variable and $\vec{r}$ a set of registers (the action stores the constraint ϕ where the values of the registers are instantiated with their current values – the shared variable x is the only free variable in $\phi(x)$). Roughly, the hvc actions are special cases of rd actions. Consider an execution π where an action $\alpha = (t, hvc, x, \phi(x))$ is generated by reading the value of a write action $\beta = (com, x, v)$ (i.e., the value v was the current value of x when the havoc instruction was executed). Then, the trace of π contains a read-from edge $\beta \rightarrow_{rf} \alpha$ as for regular read actions. However, fr edges are created differently. If α was a rd action we would say that we have $\alpha \rightarrow_{fr} \gamma$ if $\beta \rightarrow_{rf} \alpha$ and $\beta \rightarrow_{st} \gamma$. For the havoc case, the situation is a little bit different. Let $\gamma = (com, x, v')$ be an action. We have $\alpha \rightarrow_{fr} \gamma$ if and only if either $\beta \rightarrow_{rf} \alpha$, $\beta \rightarrow_{st} \gamma$ and $\phi(v')$ is false or $\alpha \rightarrow_{fr} \gamma'$ and $\gamma' \rightarrow_{st} \gamma$ where γ' is an action. Intuitively, there is a from-read dependency from an havoc action to a commit action, only when the commit action invalidates the constraint $\phi(x)$ of the havoc (or if it follows such a commit in store order).

The notion of write-atomicity (Definition 7) extends to programs with havoc instructions by interpreting havoc instructions $\text{havoc}(r, \phi(x, \vec{r}))$ as regular read instructions $r := x$. Theorem 5 which states that write-atomicity implies robustness can also be easily extended to this case.

Read abstractions are useful in two ways. First, they allow us to prove properties

```

procedure foo(){
  x := 1;
  r2 := y;
}

procedure bar(){
  do{
    r1 = x;
    //havoc(r1, (x ≠ 0)?r1 = x ∨ r1 = 0 : r1 = 0)
  } while(r1 == 0);
  y := 1;
}

```

Figure 4.5: An example program that needs a read abstraction to pass our robustness checks. The **havoc** statement in comments reads as follows: if value of x is not 0 then $r1$ gets either the value of x or 0. Otherwise, it is 0.

of non-robust program as the work stealing queue example in Figure 3.1. We can apply appropriate read abstractions to relax the original program so that it becomes robust in the end. Then, we can use SC reasoning tools on the robust program to prove invariants of the program.

Second, read abstractions could be helpful for proving robustness directly. The method based on write-atomicity we propose for verifying robustness is sound but not complete. Some incompleteness scenarios can be avoided using read abstractions. If we can abstract read instructions such that the new program reaches exactly the same states (in terms of shared variables) as the original one, it may help to avoid executions that violate mover checks.

Consider the program in Figure 4.5. The write statement $x := 1$ in procedure **foo** is not atomic. It is not a left mover due to the read of x in the do-while loop of **bar**. Moreover, the later read from y is buffer-free reachable from this write and it is not a right mover because of the write to y in **bar**. To make it atomic, we apply read abstraction to the read instruction of **bar** that reads from x . In the new relaxed read, $r1$ can read 0 along with the value of x when x is not zero as shown in the comments below the instruction. With this abstraction, the write to x becomes a left mover because reads from x after the write can now read the old value which was 0. Thus, the program becomes write-atomic. If we think of TSO traces of the abstract program and replace *hvc* nodes with *rd* nodes, we get exactly the TSO traces of the original program. However, the abstraction adds more SC traces to the program and the program becomes robust.

4.6 Experimental Evaluation

To test the practical value of our method, we have considered the benchmark for checking TSO robustness described in [3], which consists of 34 programs. This benchmark is quite exhaustive, it includes examples introduced in previous works on this subject. Many of the programs in this benchmark are easy to prove being write-atomic. Every write is followed by no buffer-free read instruction which makes them trivially atomic (like the message passing program in Figure 4.1). This holds for 20 out of the 34 programs. Out of the remaining programs, 13 required mover checks and 4 required read abstractions to show robustness (our method didn't succeed on one of the programs in the benchmark, explained at the end of this section). Except Chase-Lev, the initial versions of all the 12 examples are trace robust². Besides Chase-Lev, read-abstractions are equivalent to the original programs in terms of reachable shared variable configurations. Detailed information for these examples can be found in Table 4.1.

To check whether writes/reads are left/right movers and the soundness of abstractions, we have used the tool CIVL [31]. This tool allows to prove assertions about concurrent programs (Owicki-Gries annotations) and also to check whether an instruction is a left/right mover. The buffer-free read instructions reachable from a write before a fence were obtained using a trivial analysis of the control-flow graph (CFG) of the program. This method is a sound approximation of the definition in Section 4 but it was sufficient for all the examples.

Our method was not precise enough to prove robustness for only one example, named as `nbw-w-lr-rl` in [13]. This program contains a method with explicit calls to the `lock` and `unlock` methods of a spinlock. The instruction that writes to the lock variable inside the `unlock` method is not atomic, because of the reads from the lock variable and the calls to the `getAndSet` primitive inside the `lock` method. Abstracting

²If we consider the standard notion of *so* (that relates any two writes on the same variable independent of their values), all examples except `MCSLock` and `dc-locking` become non trace robust.

Table 4.1: Benchmark results. The second column (RB) stands for the robustness status of the original program according to our extended *hb* definition. RA column shows the number of read abstractions performed. RM column represents the number of read instructions that are checked to be right movers and the LM column represents the write instructions that are shown to be left movers. PO shows the total number of proof obligations generated and VT stands for the total verification time in seconds.

Name	RB	RA	RM	LM	PO	VT
Chase-Lev:	-	1	2	-	149	0.332
FIFO-iWSQ:	+	-	2	-	124	0.323
LIFO-iWSQ:	+	-	1	-	109	0.305
Anchor- iWSQ:	+	-	1	-	109	0.309
MCSLock:	+	2	2	-	233	0.499
r+detour:	+	-	1	-	53	0.266
r+detours:	+	-	1	-	64	0.273
sb+detours+coh:	+	-	2	-	108	0.322
sb+detours:	+	-	1	1	125	0.316
write+r+coh:	+	-	1	-	78	0.289
write+r:	+	-	1	-	48	0.261
dc-locking:	+	1	4	1	52	0.284
inline_pgsql:	+	2	2	-	90	0.286

the reads from the lock variable is not sufficient in this case due to the conflicts with `getAndSet` actions. However, we believe that read abstractions could be extended to `getAndSet` instructions (which both read and write to a shared variable atomically) in order to deal with this example.

4.7 Related Work

The weakest correctness criterion that enables SC reasoning for proving invariants of programs running under TSO is *state-robustness* i.e., the reachable set of states is the same under both SC and TSO. However, this problem has high complexity

(non-primitive recursive for programs with a finite number of threads and a finite data domain [12]). Therefore, it is difficult to come up with an efficient and precise solution. A symbolic decision procedure is presented in [2] and over-approximate analyses are proposed in [39, 40].

Due to the high complexity of state-robustness, stronger correctness criteria with lower complexity have been proposed. Trace-robustness (that we call simply robustness in this chapter) is one of the most studied criteria in the literature. Bouajjani et al. [19] have proved that deciding trace-robustness is PSPACE-complete, resp., EXPSPACE-complete, for a finite, resp., unbounded, number of threads and a finite data domain.

There are various tools for checking trace-robustness. TRENCHER [13] applies to bounded-thread programs with finite data. In theory, the approach in Trencher can be applied to infinite-state programs, but implementing it is not obvious because it requires solving non-trivial reachability queries in such programs. In comparison, our approach (and our implementation based on CIVL) applies to infinite state programs. All our examples consider infinite data domains, while Chase-Lev, FIFO-iWSQ, LIFO-iWSQ, Anchor-iWSQ, MCSLock, dc-locking and inline_pgsq have an unbounded number of threads. MUSKETEEER [7] provides an approximate solution by checking existence of critical cycles on the control-flow graph. While Musketeer can deal with infinite data (since data is abstracted away), it is restricted to bounded-thread programs. Thus, it cannot deal with the unbounded thread examples mentioned above. Furthermore, Musketeer cannot prove robust even some examples with finitely many threads, e.g., nbw_w_wr, write+r, r+detours, sb+detours+coh. Other tools for approximate robustness checking, to which we compare in similar ways, have been proposed in [8, 20, 21].

Besides trace-robustness, there are other correctness criteria like triangular race freedom (TRF) and persistence that are stronger than state-robustness. Persistence ([3]) is incomparable to trace-robustness, and TRF [55] is stronger than both trace-robustness and persistence. Our method can verify examples that are state-robust

but neither persistent nor TRF.

Reduction and abstraction techniques were used for reasoning on SC programs. QED ([26]) is a tool that supports statement transformations as a way of abstracting programs combined with a mover analysis. Also, CIVL ([31]) allows proving location assertions in the context of the Owicki-Gries logic which is enhanced with Lipton's reduction theory [45]. Our work enables the use of such tools for reasoning about the TSO semantics of a program.

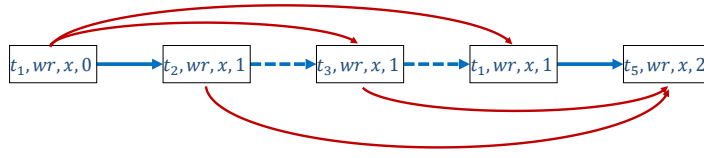


Figure 4.6: Old and new *so* relations of an execution π that writes 0, 1, 1, 1 and 2 in that order on the variable x .

A Properties of TSO Programs

In this section, we will show properties of TSO programs that are necessary for the proof of Theorem 5. In the scope of this section, we still omit the programs with **havoc** statements.

Our definition of *hb* (happens-before) order is slightly different than the one in traditional trace definition. Two actions that write the same value on the same variable are not related by *so* (store-order) according to our definition whereas they are related according to the traditional one. Differences between the standard and our definitions are depicted in Figure 4.6. Let π be a TSO execution of which commit actions on the variable x are ordered as shown with blue arrows in the figure. The standard *so* order of these actions are also represented by blue arrows. If two consequent commit actions on the same variable writes the same value, we do not order them by *so* according to our definition. Let us call them *fungible* commit actions. So, the dashed blue arrows that order the fungible commits do not exist in the new *so* definition. Since removing these edges break the ordering of prior and later commits, the new *so* includes new edges shown by red arrows in the figure. Basically, we put *so* edges from the last commit action on the same variable before the fungible commits to the fungible commits and from fungible commits to the first commit action on the same variable after the fungible commits. If the last commit before the fungible commits or the first action after the fungible commits is also a part of another fungible commit actions, we put an *so* edge from every member of the previous fungible commit group to every member of the later fungible commit group.

This modification in *so* implicitly affects *fr* (from-reads) relation part of *hb* since it is composition of *rf* and *so* relations. If two commit actions α and α' are fungible in an execution, for all read actions γ that reads from α' (i.e., there is an *rf* edge from α' to γ), we do not have an *fr* edge from γ to α . Note that, according to the standard trace definition, this *fr* edge must exist.

These differences in the definition of a trace propagates to the robustness definition. Consequently, correctness of the minimality lemma (Lemma 6) cannot be taken for granted since it depends on the traditional robustness definition. Our main aim in this section is to reprove Lemma 6 which is the only assumption in the proof of Theorem 5. However, we will need some side results until we can prove Lemma 6.

In order not to confuse the standard and new definitions, we use *std* subscript or “standard” prefix when we refer to the standard ones. For example, $Tr_{std}(\pi)$ means the standard trace of the execution π whereas $Tr(\pi)$ means the extended trace. This is compatible with the previous chapters since we have referred to the extended definitions when we say traces, robustness, happens-before relation etc.

First, we observe that the new trace definition is weaker than the standard trace definition in the following sense:

Lemma 8. *Let π be a TSO execution of program $\mathcal{P}$, α and β be two actions in π . Then, if there is a path from α to β in the new trace, then there is a path from α to β in the standard trace.*

Proof. Let $\sigma = \sigma_1, \sigma_2, \dots, \sigma_k$ be a path in the new trace such that $\sigma_1 = \alpha$ and $\sigma_k = \beta$. If this path contains no edges like the red ones in Figure 4.6, then the same path exists in the standard trace.

Let (σ_i, σ_{i+1}) be a red edge. Then, there is a set of fungible commit actions in π and either σ_i is the last commit action on the same variable before these fungible commits and σ_{i+1} is a fungible commit or σ_i is a fungible commit and σ_{i+1} is the first commit action on the same variable after the fungible commits. Since fungible commits are totally ordered by *so* according to the standard definition, there exists a path from σ_i to σ_{i+1} in the standard trace. Hence a path from α to β can be

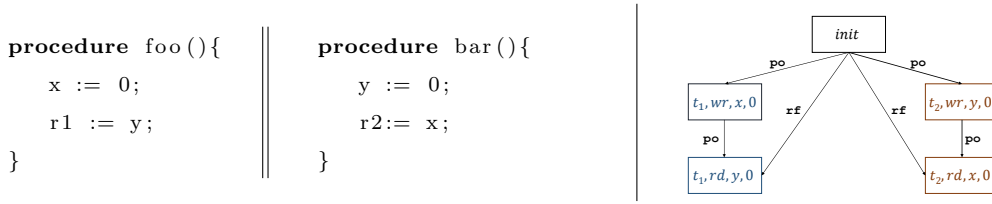


Figure 4.7: An example store buffering program and an acyclic TSO trace. For this program, x, y are shared variables and $r1, r2$ are local registers.

constructed in the standard trace. □

Next, we reprove a result by Shasha and Snir that characterizes the SC traces:

Lemma 9 ([59]). *Let π be a TSO execution of a program $\mathcal{P}$. If there exists an SC execution π' of $\mathcal{P}$ such that $Tr(\pi) = Tr(\pi')$, then $Tr(\pi)$ is acyclic.*

Lemma 9 and its reverse hold for the standard trace definition. However, the reverse direction is not true according to our definition. For the store buffering program given in Figure 4.7, we have the acyclic TSO trace depicted right side of the figure. Even though it is a valid TSO trace that can be obtained by delaying the writes after reads, no SC execution has this trace. In an SC trace, we cannot have both of the rf edges which requires an execution with two read operations from the initial state.

Knowing that Lemma 9 holds for the standard trace definition, we can prove it for the new trace definition:

Proof. Let π be a TSO execution and π' be an SC execution such that $Tr(\pi) = Tr(\pi')$. Since π' is an SC execution, the original result of Shasha and Snir implies that $Tr_{std}(\pi')$ is acyclic. Using the contrapositive of Lemma 8 (i.e., if there is no cyclic path in the standard trace, then there is no cyclic path in the new trace as well), we can conclude that $Tr(\pi')$ is acyclic. Then, $Tr(\pi)$ is acyclic as well. □

Our next step would be to show some properties of minimal violations. Before going any further, let us define minimal violations formally.

Definition 8 (Minimal Violation). *Let $\pi = \pi_1, \alpha, \pi_2, \beta, \pi_3$ be a TSO execution such that $\alpha = (t, isu)$ and $\beta = (t, com, *, *)$ are corresponding *isu* and *com* actions. Then,*

the distance between α and β are the number of actions by the same thread between α and β in π : $d_\pi(\alpha, \beta) = \pi_2|_t$. Number of delays in π is the sum of distances $d_\pi(\alpha, \beta)$ for all corresponding *isu* and *com* action pairs (α, β) . Then, a minimal violation is a violating execution (i.e., none of the SC executions have the same trace with it) for which the number of delays is minimum among all violating executions.

Minimal violations and TSO executions satisfy the following properties:

Lemma 10. *Let $\pi = \pi_1, \alpha, \pi_2, \beta, \pi_3$ be a minimal violation such that $\alpha = (t, \text{isu})$ and $\beta = (t, \text{com}, x, *)$ are corresponding *isu* and *com* actions stemming from the same instruction. Then, $\pi_2|_t$ is either empty or $\pi_2|_t = \pi_{21}, \gamma, \pi_{22}$ where $\gamma = (t, \text{rd}, y, *)$ action reads a different variable π_{22} contains only write actions.*

Lemma 11. *Let $\pi = \pi_1, \alpha, \pi_2, \beta, \pi_3$ and $\pi' = \pi'_1, \alpha, \pi'_2, \beta, \pi'_3$ be two TSO executions such that $\pi|_t = \pi'|_t$ for every thread t and π_2 be a subsequence of π'_2 . Then, if $\alpha \rightarrow_{hb+} \beta$ through π_2 , then $\alpha \rightarrow_{hb+} \beta$ through π'_2 .*

Lemma 12. *Let $\pi = \pi_1, \alpha, \pi_2, \beta, \pi_3$ be a minimal violation. Then, either*

1. $\alpha \rightarrow_{hb+} \beta$ through π_2 or
2. *there exists a TSO execution $\pi' = \pi_1, \pi_{21}, \beta, \alpha, \pi_{22}, \pi_3$ such that $\pi|_t = \pi'|_t$ for every thread t and π_{22} is a subsequence of π_2 .*

Proofs for lemmata 10, 11 and 12 in [13] are still valid if we use our new relaxed *hb* definition. Only difference with [13] is that we relax Lemma 12 by removing $Tr(\pi) = Tr(\pi')$ condition from the second case. However, this lemma is useful as before for the later results. The following result is shown using the previous helper lemmas and the proof in [13] is still valid:

Lemma 13. *In a minimal violation, only single thread delays its write actions.*

Now, we can prove Lemma 6 that shows soundness and completeness of witness minimal violations. To be precise, let us define a witness minimal violation:

Definition 9 (Minimal Violation Witness). *Let $\pi = \pi_1, \alpha, \pi_2, \gamma, \pi_3, \beta, \pi_4$ be a minimal violation such that $\alpha = (t, isu)$, $\gamma = (t, rd, y, *)$, $\beta = (t, com, x, *)$ is the corresponding *com* action of α and $x \neq y$. Then, π is a witness minimal violation (or witness for short) iff:*

1. *only t delays its *com* actions,*
2. *β is the first delayed *com* action and γ is the last action of t overstepped by β ,*
3. *$\gamma \rightarrow_{hb+} \beta$ through π_3 ,*
4. *π_4 only contains delayed *com* actions of writes from t that were issued before γ ,
and*
5. *all the *com* actions in π_4 writes on a different variable than y .*

Next, we prove Lemma 6.

Proof. First, we show “if” part. Assume that $\mathcal{P}$ has a witness π defined exactly as above. Consider the trace of π . We have $\alpha \rightarrow_{po+} \gamma$ and $\gamma \rightarrow_{hb+} \beta$ due to the definition of a witness. Since α and β correspond to the same node in the trace, $Tr(\pi)$ is cyclic. By Lemma 9, there is no SC execution that has the same trace with π . Thus, $\mathcal{P}$ is not robust.

For the reverse direction, we assume that $\mathcal{P}$ is not robust. Then, there exists a minimal violation π . We will show that the shape of a minimal violation fulfills the conditions 1 – 5 of a witness defined above.

1. By Lemma 13, only a single thread delays its *com* actions in π . Let us call this thread as t .
2. Let α be the first *isu* action of t of which corresponding *com* action β is delayed. By Lemma 10, there exists a read action between α and β in π . We call the last such read action as γ .

3. To reach a contradiction, assume that $\gamma \not\rightarrow_{hb+} \beta$. Then, there is a feasible TSO execution in which β is executed just before γ . This execution has one less delay than π contradicting minimality of π .
4. Consider the actions after β in π . All the actions executed from threads other than t or actions by t that are not delayed *com* actions are redundant. Executions obtained by removing those actions are still feasible minimal violations obeying condition 4 of a witness.
5. To reach a contradiction, assume that there exists a *com* action $\beta' = (t, com, y, *)$ after β in π i.e., the part of π after β is of the form $\pi_4 = \pi_{41}, \beta', \pi_{42}$. We can avoid γ until executing β' . There is a feasible TSO execution that can be obtained by removing γ from π and replacing π_4 with $\pi_{41}, \beta', \gamma, \pi_{42}$. This execution has one less delay than π which contradicts minimality of π .

□

B A Hierarchy of TSO Consistency Models

To show correctness of TSO programs, various correctness criteria are proposed. To prove safety properties of a program, one needs to check that the program never visits a bad state. Following this purpose, *state-robustness* consistency criterion is proposed. A program is state-robust iff every state reachable by a TSO execution is also reachable by an SC execution. As a result, if a program is state-robust, every safety property satisfied by SC executions of the program is also satisfied by TSO executions.

Although state-robustness is a desirable safety property for TSO programs, checking state-robustness is a high complexity problem. It is non-primitive recursive as noted previously in Section 4.7. For this reason, other consistency models that can be checked efficiently, are proposed. Other consistency models are stronger than state-robustness in the sense that a program satisfying this model is a state-robust

program. Therefore, these consistency models imply state robustness and ensure both the TSO and SC executions of the program guarantee the same safety properties. We can list widely-accepted TSO consistency models from the strongest to the weakest as *data-race freedom* ([5]), *triangular-race freedom* ([55]), *trace-robustness* ([13]) and *state-robustness*. Here, trace-robustness corresponds to trace equivalence of TSO and SC executions of a program with respect to standard trace definitions.

In this section, we show that trace-robustness with our extended trace definition is stronger than standard trace-robustness and weaker than state-robustness. This makes new trace-robustness consistency model the weakest one excluding state-robustness. As shown by the experiments, our correctness criterion can be checked to verify some state-robust benchmark programs that cannot be verified using other consistency models.

First, we show that standard robustness implies robustness.

Lemma 14. *Let $\mathcal{P}$ be a standard trace-robust program. Then, $\mathcal{P}$ is trace-robust.*

Proof. Let $\mathcal{P}$ be a standard robust program and π be a TSO execution of $\mathcal{P}$. Then, there exists an SC execution π' such that $Tr_{std}(\pi) = Tr_{std}(\pi')$. Let $Exc(\phi)$ be a set of *so* and *fr* edges of $Tr_{std}(\phi)$ where ϕ is an execution such that value fields (the last fields) of the nodes at both ends of the edges are the same. Similarly, let $Inc(\phi)$ be the set of *so* and *fr* edges that exist in $Tr(\phi)$ but not $Tr_{std}(\phi)$. Basically, $Inc(\phi)$ is the union of red *so* edges in Figure 4.6 and implied new *fr* edges. Then, $Exc(\pi) = Exc(\pi')$ and $Inc(\pi) = Inc(\pi')$. Clearly, $Tr(\pi) = Tr_{std}(\pi) \cup Inc(\pi) \setminus Exc(\pi)$ and $Tr(\pi') = Tr_{std}(\pi') \cup Inc(\pi') \setminus Exc(\pi')$. Consequently, $Tr(\pi) = Tr(\pi')$ for the SC execution π' . $\square$

To show robustness implies state-robustness, we need the following lemma.

Lemma 15. *Let $\pi = \pi_1, \beta_1, \pi_2, \beta_2, \pi_3$ be a TSO execution where $\beta_1 = (t_1, com, x, v_1)$, $\beta_2 = (t_2, com, x, v_2)$ and $v_1 \neq v_2$. Then, $\beta_1 \rightarrow_{so+} \beta_2$ through π_2 according to the new trace definition.*

Proof. Consider $\pi_2|_{x,com}$ (actions that commit on x in π_2). We consider each fungible commit group in this sequence as a single element and we apply induction on the number of fungible commit groups in this sequence.

Base Case: $\pi_2|_{x,com}$ is a sequence of fungible commits that write v_1 on x followed by another sequence of fungible commits that write v_2 on x . Then, β_1 is a member of the first fungible commit group and β_2 is the member of the second fungible commit group in π . Then, $\beta_1 \rightarrow_{so} \beta_2$ by definition of so .

Inductive Case: Assume that the claim holds if the number of fungible commit action groups in $\pi_2|_{x,com}$ are less than or equal to k . We consider the case where the number of such fungible commit action groups is $k + 1$.

Pick a commit action $\beta_3 = (t_3, com, x, v_3) \in \pi_2$ from a fungible commit action group that β_1 and β_2 are not a member of. Such β_3 exists since $k + 1 > 0$. Then, $\beta_1 \rightarrow_{so+} \beta_3$ and $\beta_3 \rightarrow_{so+} \beta_2$ by the induction hypothesis. As a result, $\beta_1 \rightarrow_{so+} \beta_2$. $\square$

Last, we show that robustness implies state-robustness.

Lemma 16. *Let $\mathcal{P}$ be a trace-robust program. Then, $\mathcal{P}$ is state-robust as well.*

Proof. Since $\mathcal{P}$ is robust, we know that for every TSO execution π , there exists an SC execution π' such that $Tr(\pi) = Tr(\pi')$. Let σ_π and $\sigma_{\pi'}$ be the last states of π and π' , respectively.

First, we show that the values of the shared variables in σ_π and $\sigma_{\pi'}$ are the same. Let x be a shared variable. The value of x in σ_π is assigned by the last commit action on x in π . Let us call this action $\beta = (t, com, x, v)$. Then, β is a maximal action if we project $Tr(\pi)$ on so edges that operate on x . β is a maximal action in projection of $Tr(\pi')$ on so edges that operate on x as well since $Tr(\pi) = Tr(\pi')$. Then, β is either a maximum element in $\pi'|_{x,com}$ or there are later commit actions that write the same value on x . In any case, $\sigma_{\pi'}$ contains the value v for x .

For the local variables (registers), neither standard nor extended trace equivalence imply equivalence of local variables since thread local actions' values are not kept in the traces. For this reason, we have the reasonable assumption that $\pi|_{t\#}$ (projection of π on non-commit actions by t) and $\pi'|_{t\#}$ are the same for all threads t .

Assuming this, we prove that the values of registers in $\vec{r}_t$ are the same both in σ_π and $\sigma_{\pi'}$ where t is a thread identifier.

Let γ be the first action by thread t that assigns a value to a local register in $\vec{r}_t$ such that the value of a register in $\vec{r}_t$ becomes different in π and π' after executing this action. To express this more formally, let $\pi|_\gamma$ be a prefix of π with γ as the last action. We aim to pick γ such that there is a register r in $\vec{r}_t$ and value of r is different in the end states of $\pi|_\gamma$ and $\pi'|_\gamma$. We will show that such a γ does not exist by reaching a contradiction.

Assume that such a γ exists. Since it is the first action that assigns different values to a register in $\vec{r}_t$, values of all the registers are the same in both π and π' just before executing γ . The value assigned by γ is an expression over constants, $\vec{r}_t$ and/or a shared variable x . Since all the registers of t are the same before executing γ , if the value assigned by γ is different in π and π' , then the value read from x is different in those executions. Let $\beta = (t_1, com, x, v_1)$ be the commit action such that $\beta \rightarrow_{rf} \gamma$ in π and $\beta' = (t_2, com, x, v_2)$ be the commit action such that $\beta' \rightarrow_{rf} \gamma$ in π' . Since γ reads different values, $v_1 \neq v_2$ and β and β' cannot be fungible. Without loss of generality, assume that β occurs before β' in π . By Lemma 15, $\beta \rightarrow_{so+} \beta'$ in π . Since $\beta \rightarrow_{rf} \gamma$, $\beta \rightarrow_{so+} \beta'$ in π and β' assigns a different value than γ reads, $\gamma \rightarrow_{fr} \beta'$. Since $Tr(\pi) = Tr(\pi')$, γ happens before β' in π' as well. So, we reach the contradictory result that $\gamma \not\rightarrow_{rf} \beta'$ in π' and we conclude that γ does not exist. $\square$

Chapter 5

CONCLUSIONS AND FUTURE WORK

In this thesis we present three studies on using refinement proofs for verifying concurrent programs. The first study is about establishing forward simulation relations to show linearizability of concurrent libraries running on SC memory model. Its main contribution is new stack and queue specifications that are equivalent to the standard specifications (in the sense that they produce the same behaviors) and it is guaranteed that there exists forward simulation relations between correct concurrent implementations and these new specifications. Using these new specifications, we have proven correctness of complicated Herlihy & Wing Queue and Time-Stamped Stack implementations.

Although this study is limited with concurrent stacks and queues, we believe that our approach can be extended for other fundamental data structures. Sets and multi-sets are first types of objects to be considered since they form the basis of other data structures. Basis for our belief is that most of the concurrent set and multi-set implementations have fixed linearization (or commit like points) for remove methods.

We favor forward simulations since they are more intuitive and they reflect the natural forward flow of programs. This natural forward flow also exists as the semantic model of the concurrent programs in automated proof tools. Hence, forward simulations can be easily integrated into existing proof assistants (like CIVL) as proof methods. If we know the forward simulation relation, we can identify the corresponding atomic actions of the implementation and the specification. Then, we can introduce a new program in which corresponding actions of both programs take their atomic steps together. Then, our forward simulation relation is valid if it is an invariant of this program. So, proof assistants that can perform assertion checking on concurrent

SC programs can be extended for proving linearizability and checking correctness of a forward simulation. In future, we aim to extend CIVL for that purpose and automate the correctness proofs of the HWQ and TSS described in the previous chapters.

The second study describes a linearizability proof of the Chase-Lev WSQ implementation. This time, the proof is not performed by finding a simulation relation but by applying a sequence of reductions and abstractions. The proof consists of several layers and granularity of atomic blocks increase after each proof layer. We reach linearizability at the last layer in which all of the method bodies become atomic. This proof is mechanized and carried on the CIVL proof assistant.

The third study extends the previous one. We use SC abstraction and reduction techniques for verifying TSO programs. Moreover, a correctness proof for the previous WSQ implementation is presented for TSO executions. This study presents write-atomicity, a reduction based SC technique, for checking TSO robustness. In addition, we propose a valid abstraction method, called read-abstractions, for verifying non-robust programs. Using new techniques proposed, we achieve to verify 33 of 34 challenging benchmark examples.

Although write-atomicity is a sound method for checking robustness (see Theorem 5), it is not complete. In future, we aim to extend write-atomicity to come up with a complete method for checking robustness. Moreover, we want to add other valid abstraction methods (like write or CAS abstractions) to enable verification of more TSO programs as the single failing benchmark example.

Another open question that needs to be addressed is the relation between different methods for proving refinement relations. We have examined two methods in this thesis for proving refinements: establishing simulation relations and increasing granularity of atomic blocks by using abstraction and reduction. These two approaches contain similarities and differences. For some examples, finding a simulation relation is easier and more natural. In others, abstraction and reduction are applied. In future, we aim to analyse connections between these two techniques and try to understand conditions in which one them is more preferable.

BIBLIOGRAPHY

- [1] Martín Abadi and Leslie Lamport. The existence of refinement mappings. *Theor. Comput. Sci.*, 82(2):253–284, 1991.
- [2] Parosh Aziz Abdulla, Mohamed Faouzi Atig, Yu-Fang Chen, Carl Leonardsson, and Ahmed Rezzine. Counter-example guided fence insertion under tso. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 204–219. Springer, 2012.
- [3] Parosh Aziz Abdulla, Mohamed Faouzi Atig, and Tuan-Phong Ngo. The best of both worlds: Trading efficiency and optimality in fence insertion for tso. In *European Symposium on Programming Languages and Systems*, pages 308–332. Springer, 2015.
- [4] Parosh Aziz Abdulla, Frédéric Haziza, Lukás Holík, Bengt Jonsson, and Ahmed Rezzine. An integrated specification and verification technique for highly concurrent data structures. In *TACAS*, pages 324–338, 2013.
- [5] Sarita V Adve and Mark D Hill. Weak ordering – a new definition. In *ACM SIGARCH Computer Architecture News*, volume 18, pages 2–14. ACM, 1990.
- [6] Sarita V. Adve and Mark D. Hill. A unified formalization of four shared-memory models. *IEEE Trans. Parallel Distrib. Syst.*, 4(6):613–624, 1993.
- [7] Jade Alglave, Daniel Kroening, Vincent Nimal, and Daniel Poetzl. Dont sit on the fence. In *International Conference on Computer Aided Verification*, pages 508–524. Springer, 2014.

- [8] Jade Alglave and Luc Maranget. Stability in weak memory models. In *Computer Aided Verification*, pages 50–66. Springer, 2011.
- [9] Rajeev Alur, Kenneth L. McMillan, and Doron Peled. Model-checking of correctness conditions for concurrent objects. *Inf. Comput.*, 160(1-2):167–188, 2000.
- [10] Daphna Amit, Noam Rinetzky, Thomas W. Reps, Mooly Sagiv, and Eran Yahav. Comparison under abstraction for verifying linearizability. In *CAV '07*, volume 4590 of *LNCS*, pages 477–490, 2007.
- [11] Nimar S Arora, Robert D Blumofe, and C Greg Plaxton. Thread scheduling for multiprogrammed multiprocessors. *Theory of Computing Systems*, 34(2):115–144, 2001.
- [12] Mohamed Faouzi Atig, Ahmed Bouajjani, Sebastian Burckhardt, and Madanlal Musuvathi. On the verification problem for weak memory models. *ACM Sigplan Notices*, 45(1):7–18, 2010.
- [13] Ahmed Bouajjani, Egor Derevenetc, and Roland Meyer. Checking and enforcing robustness against tso. In *Programming Languages and Systems*, pages 533–553. Springer, 2013.
- [14] Ahmed Bouajjani, Michael Emmi, Constantin Enea, and Jad Hamza. Verifying concurrent programs against sequential specifications. In *ESOP '13*, volume 7792 of *LNCS*, pages 290–309. Springer, 2013.
- [15] Ahmed Bouajjani, Michael Emmi, Constantin Enea, and Jad Hamza. On reducing linearizability to state reachability. In *Automata, Languages, and Programming - 42nd International Colloquium, ICALP 2015, Kyoto, Japan, July 6-10, 2015, Proceedings, Part II*, pages 95–107, 2015.
- [16] Ahmed Bouajjani, Michael Emmi, Constantin Enea, and Jad Hamza. Tractable refinement checking for concurrent objects. In *Proceedings of the 42nd Annual*

- ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2015, Mumbai, India, January 15-17, 2015*, pages 651–662, 2015.
- [17] Ahmed Bouajjani, Michael Emmi, Constantin Enea, and Suha Orhun Mutluergil. Proving linearizability using forward simulations. In *International Conference on Computer Aided Verification*, pages 542–563. Springer, 2017.
- [18] Ahmed Bouajjani, Constantin Enea, Suha Orhun Mutluergil, and Tasiran Serdar. Proving linearizability using forward simulations. In *International Conference on Computer Aided Verification*. Springer, 2018.
- [19] Ahmed Bouajjani, Roland Meyer, and Eike Möhlmann. Deciding robustness against total store ordering. In *International Colloquium on Automata, Languages, and Programming*, pages 428–440. Springer, 2011.
- [20] Sebastian Burckhardt and Madanlal Musuvathi. Effective program verification for relaxed memory models. In *International Conference on Computer Aided Verification*, pages 107–120. Springer, 2008.
- [21] Jakob Burnim, Koushik Sen, and Christos Stergiou. Sound and complete monitoring of sequential consistency for relaxed memory models. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 11–25. Springer, 2011.
- [22] David Chase and Yossi Lev. Dynamic circular work-stealing deque. In *Proceedings of the seventeenth annual ACM symposium on Parallelism in algorithms and architectures*, pages 21–28. ACM, 2005.
- [23] John Derrick, Gerhard Schellhorn, and Heike Wehrheim. *Verifying Linearisability with Potential Linearisation Points*, pages 323–337. Springer Berlin Heidelberg, Berlin, Heidelberg, 2011.

- [24] Mike Dodds, Andreas Haas, and Christoph M Kirsch. A scalable, correct time-stamped stack. In *ACM SIGPLAN Notices*, volume 50, pages 233–246. ACM, 2015.
- [25] Cezara Dragoi, Ashutosh Gupta, and Thomas A. Henzinger. Automatic linearizability proofs of concurrent objects with cooperating updates. In *CAV '13*, volume 8044 of *LNCS*, pages 174–190. Springer.
- [26] Tayfun Elmas, Shaz Qadeer, and Serdar Tasiran. A calculus of atomic actions. In *ACM Symposium on Principles of Programming Languages*, page 14. Association for Computing Machinery, Inc., January 2009.
- [27] Ivana Filipovic, Peter W. O'Hearn, Noam Rinetzkky, and Hongseok Yang. Abstraction for concurrent objects. *Theor. Comput. Sci.*, 411(51-52):4379–4398, 2010.
- [28] Matteo Frigo, Charles E Leiserson, and Keith H Randall. The implementation of the cilk-5 multithreaded language. In *ACM Sigplan Notices*, volume 33, pages 212–223. ACM, 1998.
- [29] Michael Gorelik and Danny Hendler. Brief announcement: an asymmetric flat-combining based queue algorithm. In *ACM Symposium on Principles of Distributed Computing, PODC '13, Montreal, QC, Canada, July 22-24, 2013*, pages 319–321, 2013.
- [30] Jad Hamza. On the complexity of linearizability. In *Networked Systems - Third International Conference, NETYS 2015, Agadir, Morocco, May 13-15, 2015, Revised Selected Papers*, pages 308–321, 2015.
- [31] Chris Hawblitzel, Shaz Qadeer, and Serdar Tasiran. Automated and modular refinement reasoning for concurrent programs. *Computer Aided Verification*, 2015.

- [32] Danny Hendler, Itai Incze, Nir Shavit, and Moran Tzafrir. Flat combining and the synchronization-parallelism tradeoff. In *SPAA 2010: Proceedings of the 22nd Annual ACM Symposium on Parallelism in Algorithms and Architectures, Thira, Santorini, Greece, June 13-15, 2010*, pages 355–364, 2010.
- [33] Danny Hendler, Nir Shavit, and Lena Yerushalmi. A scalable lock-free stack algorithm. In *SPAA 2004*, pages 206–215. ACM.
- [34] Thomas A. Henzinger, Ali Sezgin, and Viktor Vafeiadis. Aspect-oriented linearizability proofs. In *CONCUR*, pages 242–256, 2013.
- [35] Maurice Herlihy, Victor Luchangco, and Mark Moir. Obstruction-free synchronization: Double-ended queues as an example. In *Distributed Computing Systems, 2003. Proceedings. 23rd International Conference on*, pages 522–529. IEEE, 2003.
- [36] Maurice P Herlihy and Jeannette M Wing. Linearizability: A correctness condition for concurrent objects. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 12(3):463–492, 1990.
- [37] Moshe Hoffman, Ori Shalev, and Nir Shavit. The baskets queue. In *Principles of Distributed Systems, 11th International Conference, OPODIS 2007, Guadeloupe, French West Indies, December 17-20, 2007. Proceedings*, pages 401–414, 2007.
- [38] Majid Khorsandi Aghai. Verification of work-stealing deque implementation. 2012.
- [39] Michael Kuperstein, Martin Vechev, and Eran Yahav. Partial-coherence abstractions for relaxed memory models. In *ACM SIGPLAN Notices*, volume 46, pages 187–198. ACM, 2011.
- [40] Michael Kuperstein, Martin Vechev, and Eran Yahav. Automatic inference of memory fences. *ACM SIGACT News*, 43(2):108–123, 2012.

- [41] Leslie Lamport. How to make a multiprocessor computer that correctly executes multiprocess programs. *Computers, IEEE Transactions on*, 100(9):690–691, 1979.
- [42] Nhat Minh Lê, Antoniu Pop, Albert Cohen, and Francesco Zappa Nardelli. Correct and efficient work-stealing for weak memory models. In *ACM SIGPLAN Notices*, volume 48, pages 69–80. ACM, 2013.
- [43] Hongjin Liang and Xinyu Feng. Modular verification of linearizability with non-fixed linearization points. In *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '13, Seattle, WA, USA, June 16-19, 2013*, pages 459–470, 2013.
- [44] Richard J. Lipton. Reduction: A method of proving properties of parallel programs. *Commun. ACM*, 18(12):717–721, 1975.
- [45] Richard J Lipton. Reduction: A method of proving properties of parallel programs. *Communications of the ACM*, 18(12):717–721, 1975.
- [46] Feng Liu, Nayden Nedev, Nedyalko Prisadnikov, Martin Vechev, and Eran Yahav. Dynamic synthesis for relaxed memory models. In *ACM SIGPLAN Notices*, volume 47, pages 429–440. ACM, 2012.
- [47] Nancy A. Lynch and Frits W. Vaandrager. Forward and backward simulations: I. untimed systems. *Inf. Comput.*, 121(2):214–233, 1995.
- [48] Maged M Michael, Martin T Vechev, and Vijay A Saraswat. *Idempotent work stealing*, volume 44. ACM, 2009.
- [49] Adam Morrison and Yehuda Afek. Fast concurrent queues for x86 processors. In *ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPOPP '13, Shenzhen, China, February 23-27, 2013*, pages 103–112, 2013.

- [50] Adam Morrison and Yehuda Afek. Fence-free work stealing on bounded tso processors. *ACM SIGPLAN Notices*, 49(4):413–426, 2014.
- [51] Markus Müller-Olm, Oliver Rüthing, and Helmut Seidl. Checking herbrand equalities and beyond. In *International Workshop on Verification, Model Checking, and Abstract Interpretation*, pages 79–96. Springer, 2005.
- [52] Markus Müller-Olm and Helmut Seidl. A note on karrs algorithm. In *International Colloquium on Automata, Languages, and Programming*, pages 1016–1028. Springer, 2004.
- [53] Suha Orhun Mutluergil and Serdar Tasiran. A mechanized refinement proof of the chase-lev deque using a proof system. In *International Conference on Networked Systems*, pages 280–294. Springer, 2016.
- [54] Peter W. O’Hearn, Noam Rinetzky, Martin T. Vechev, Eran Yahav, and Greta Yorsh. Verifying linearizability with hindsight. In *PODC ’10*, pages 85–94. ACM.
- [55] Scott Owens. Reasoning about the implementation of concurrency abstractions on x86-tso. In *ECOOOP*, volume 6183, pages 478–503. Springer, 2010.
- [56] Susan Owicki and David Gries. An axiomatic proof technique for parallel programs i. *Acta informatica*, 6(4):319–340, 1976.
- [57] Gerhard Schellhorn, Heike Wehrheim, and John Derrick. How to prove algorithms linearisable. In *Computer Aided Verification*, pages 243–259. Springer, 2012.
- [58] Peter Sewell, Susmit Sarkar, Scott Owens, Francesco Zappa Nardelli, and Magnus O. Myreen. x86-tso: a rigorous and usable programmer’s model for x86 multiprocessors. *Commun. ACM*, 53(7):89–97, 2010.

- [59] Dennis Shasha and Marc Snir. Efficient and correct execution of parallel programs that share memory. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 10(2):282–312, 1988.
- [60] Oleg Travkin and Heike Wehrheim. Handling tso in mechanized linearizability proofs. In *Haifa Verification Conference*, pages 132–147. Springer, 2014.
- [61] Alan Mathison Turing. On computable numbers, with an application to the entscheidungsproblem. *Proceedings of the London mathematical society*, 2(1):230–265, 1937.
- [62] Viktor Vafeiadis. Automatically proving linearizability. In *CAV ’10*, volume 6174 of *LNCS*, pages 450–464.
- [63] Viktor Vafeiadis. Shape-value abstraction for verifying linearizability. In *VMCAI ’09: Proc. 10th Intl. Conf. on Verification, Model Checking, and Abstract Interpretation*, volume 5403 of *LNCS*, pages 335–348. Springer, 2009.
- [64] Viktor Vafeiadis, Maurice Herlihy, Tony Hoare, and Marc Shapiro. Proving correctness of highly-concurrent linearisable objects. In *PPOPP ’06*, pages 129–136. ACM.
- [65] He Zhu, Gustavo Petri, and Suresh Jagannathan. Poling: SMT aided linearizability proofs. In *Computer Aided Verification - 27th International Conference, CAV 2015, San Francisco, CA, USA, July 18-24, 2015, Proceedings, Part II*, pages 3–19, 2015.