Verifying Concurrent Programs under Weak Memory Models

Dissertation

zur Erlangung des akademischen Grades eines
Doktors der Naturwissenschaften
an der
Fakultät für Elektrotechnik, Informatik und Mathematik
der
Universität Paderborn

vorgelegt von

Oleg Travkin, M.Sc.

Paderborn, Juli 2017

Abstract

Modern multicore processors provide weak memory models like TSO, PSO or even weaker execution semantics. These memory models – due to store buffers – seemingly reorder program operations. Thus, they deviate from the commonly assumed sequential consistency (SC) semantics. Analysis and in particular verification techniques for concurrent programs consequently need to take these weak semantics into account. Linearizability, a de-facto standard correctness condition for concurrent data structures, has been defined under assumption of SC in the early 90's. For years, it was open to personal interpretation of what it means under weak memory models. Just recently, new adaptations were proposed that would define this relation formally.

In this thesis, we present a general verification approach for concurrent programs under the weak memory models TSO and PSO. The approach is based on a reduction of concurrent programs under TSO (resp. PSO) to an SC program. Thereby we enable reuse of standard verification tools, developed under SC assumption. The reduction involves two steps: a symbolic exploration of possible store buffer states, which results in a graph representation of program behavior under TSO (resp. PSO). We call it a *store buffer graph.* The latter is then transformed into a new SC program that mimics the behavior of the original program under TSO (resp. PSO). We prove both programs to be behaviorally equivalent (bisimulation). Our tool Weak2SC implements these steps as a transformation from LLVM IR programs towards Promela models for the model checker Spin and predicate logic encoding for the theorem prover KIV. Furthermore, we evaluate our reduction approach by comparing it against more common modeling techniques of weak memory semantics, which model store buffers and their behavior explicitly. To this end, we take existing approaches for verification of linearizability under SC and adapt them to the setting of weak memory models. Two of them are model checking techniques and aim at finding bugs in implementations. The third is a proof method for linearizability. We apply these approaches to a set of typical concurrent data structures and achieve promising results in terms of performance (resp. proof effort). In addition, we also discuss how the adapted verification approaches relate to the new formalisations of linearizability under weak memory models.

Zusammenfassung

Moderne Multicore Prozessoren haben schwache Speichermodelle wie TSO, PSO oder gar noch schwächere Ausführungssemantiken. Diese Speichermodelle – auf Grund von Store Buffern – sortieren scheinbar Programm-Operationen um. Damit weichen sie von der üblicherweise angenommenen Sequenziellen Konsistenz (SC) ab. Analysen und insbesondere Verifikationstechniken für nebenläufige Programme müssen daher diese schwachen Speichermodelle berücksichtigen. Linearisierbarkeit, ein de-facto Standard Korrektheitskriterium für nebenläufige Datenstrukturen, wurde unter der Annahme von SC Anfang der 90er Jahre definiert. Jahrelang überließ man die Bedeutung von Linearisierbarkeit unter schwachen Speichermodellen der eigenen Interpretation. Erst kürzlich wurden neue Varianten davon vorgeschlagen, die diesen Zusammenhang formalisieren.

In dieser Arbeit präsentieren wir einen allgemeinen Verifikationsansatz für nebenläufige Programme unter den schwachen Speichermodellen TSO und PSO. Der Ansatz basiert auf einer Reduktion von nebenläufigen Programmen unter TSO (resp. PSO) auf SC Programme. Damit ermöglichen wir die Verwendung von Standart-Verifikationswerkzeugen, die unter Annahme von SC entwickelt wurden. Die Reduktion erfolgt in zwei Schritten: Eine symbolische Exploration von möglichen Store Buffer-Zuständen, die zu einer Repräsentation des Programmverhaltens unter TSO (resp. PSO) als Graph führt. Wir nennen dies einen *Store Buffer Graphen*. Letzterer wird anschließend in ein neues SC Programm transformiert, welches das Verhalten des ursprünglichen Programms unter TSO (resp. PSO) imitiert. Wir beweisen, dass beide Programme Verhaltensäquivalent sind (Bisimulation). Unser Werkzeug Weak2SC implementiert diese Schritte als Transformation von LLVM IR Programmen hin zu Promela Modellen für den Model Checker Spin und in eine prädikatlogische Kodierung für den Theorembeweiser KIV. Außerdem, evaluieren wir den Reduktionsansatz, indem wir ihn gegen andere übliche Techniken zur Modellierung schwacher Speichersemantiken vergleichen, die Store Buffer und ihr Verhalten explizit modellieren. Dafür nehmen wir bereits existierende Verifikationsansätze für Linearisierbarkeit unter SC und adaptieren diese für schwache Speichermodelle. Zwei von ihnen sind Model Checking-Ansätze und zielen auf das Finden von Implementierungsfehlern. Der dritte Ansatz ist eine Beweismethode für Linearisierbarkeit. Wir wenden diese Ansätze auf eine Menge von typischen nebenläufigen Datenstrukturen an und erhalten vielversprechende Resultate im Sinne von Performanz (resp. Beweisaufwand). Außerdem diskutieren wir, wie die verwendeten Verifikationsverfahren mit den neuen Formalisierungen von Linearisierbarkeit unter schwachen Speichermodellen zusammenhängen.

Acknowledgments

A big "thank you" goes to Heike Wehrheim who gave me the opportunity to become a PhD student, work in in an interesting research field and eventually to write down this thesis. Thank you for all the advice that you gave me and for all your trust in me.

I would also like to thank to our research fellows Gerhard Schellhorn, Bogdan Tofan, John Derrick, Brijesh Dongol, Simon Doherty and Alasdair Armstrong for all their advice and fruitful discussions. In particular, I would like to thank Gerhard and Bogdan for their outstanding support and help to me with the theorem prover KIV.

Thanks go to our research group including former and external members: Thomas Ruhroth, Nils Timm, Dominik Steenken, Galina Besova, Steffen Ziegert, Daniel Wonisch, Sven Walther, Alexander Schremmer, Tobias Isenberg, Marie-Christine Jakobs, Steffen Behringer, Manuel Töws, Julia Krämer, Jürgen König and Elisabeth Schlatt.

Furthermore, I would like to thank to my former student assistants Annika Mütze, Thomas Haarhoff, Monika Wedel and Alexander Hetzer for helping me with the implementation of Weak2SC. Thanks go also to my former students Michael Feldmann, Katharina Dridger, Matthias Multhaup and Sven Hartwig with whom I had interesting discussions during my supervision of their theses.

Very special thanks go to my brother Dietrich, who pushed our parents to buy our first computer in Christmas 1994 and with whom I shared my first interest in computers. Since then, we kept it both until today. Of course, I also thank may parents, Helene and Fjodor, for buying the computer and for their support of both of us.

Special thanks go also to my wife Sandrina, who is desperately waiting for the day when she can buy new door plates for us. ☺

Contents

1 Introduction **1**

1.1 Concurrency on Modern Multicore Processors . . . 3

1.1.1 Weak Memory Models . . . 3

1.1.2 Towards Program Correctness . . . 6

1.2 Contributions . . . 8

1.3 Overview . . . 10

2 Memory Models **11**

2.1 Programs . . . 13

2.2 Parameterized Semantics . . . 14

2.3 Sequential Consistency . . . 18

2.4 Total Store Order . . . 19

2.5 Partial Store Order . . . 23

2.6 Relaxed Memory Order . . . 26

2.7 Related Work . . . 29

3 Reduction from Weak Semantics to Sequential Consistency **31**

3.1 Symbolic Execution with Weak Memory Semantics . . . 33

3.1.1 Store Buffer Graph . . . 35

3.1.2 Store Buffer Graphs Properties . . . 40

3.2 Transformation to a new SC Program . . . 41

3.3 Reduction is Sound and Compositional . . . 46

3.3.1 Local Bisimulation Equivalence . . . 48

3.3.2 Compositionality of the Approach . . . 52

3.3.3 Related Work and Discussion 54

4 Weak2SC – The Implementation 65
4.1 Architecture of Weak2SC 66
4.2 Case Study – Work Stealing Queue 70
4.3 From LLVM IR to a Store Buffer Graph 74
4.4 Template-based Generation of new Programs 78
4.4.1 Generating Promela Programs 80
4.4.2 Generating KIV Program Encoding 85
4.4.3 Promela Programs for Operational Memory Models 91
4.5 Discussion and Possible Future Extensions 95

5 Correctness of Concurrent Data Structures 99
5.1 Linearizability 99
5.1.1 Linearizability - Original Definition 100
5.1.2 Adaptations to Weak Memory Models 102
5.2 Discussion 113
5.2.1 Verification Methods for Linearizability 114
5.3 Other Correctness Conditions 118

6 Verifying Linearizability under Weak Memory Models 121
6.1 Model Checking under Weak Memory Models 122
6.1.1 The Idea - Abstract Atomic Specifications 123
6.1.2 Experiments 132
6.1.3 An alternative Idea - History Checking 140
6.2 Proving Linearizability under Weak Memory Models 145
6.2.1 Overview 146
6.2.2 Abstract Data Type 153
6.2.3 Concrete Data Type 154
6.2.4 Abstraction Function 160
6.2.5 Invariant 165
6.2.6 Proof Procedure and Comparison 169
6.3 Related Work and Discussion 170

7 Conclusion 175
7.1 Summary 175
7.2 Future Work 177
7.3 Design Decisions 179
7.4 Concluding Thoughts 180

A Proofs **181**
A.1 Behavioral Equivalence 181
A.2 Compositionality 195

B Code Examples **197**

Bibliography **201**

List of Figures

1.1 Programmer intuition of concurrency [AG96] 4
1.2 Illustration of the x86-architecture 5
1.3 Litmus test for reordering of writes with later reads 5
1.4 Weak2SC, a memory model-aware verification approach 8

2.1 Litmus test for reordering of writes with later reads 20
2.2 States of Process 1 of Fig. 2.1 20
2.3 Litmus test for early-reads 21
2.4 States of Process 1 of Fig. 2.3 21
2.5 Litmus test for reordering of writes with other writes 24
2.6 States of Process 1 of Fig. 2.5 24
2.7 Illustration of relaxed architectures like RMO, Power and ARM [MSS12] 27
2.8 Litmus test: Independent Reads of Independent Writes (IRIW) 28

3.1 Three steps towards verification of concurrent programs under weak memory models 32
3.2 Simple program and its infinite state space under weak memory models 34
3.3 An example program (left) containing a write-def-chain and its store buffer graph (right). The important edges and the register variable $r1$ are marked red. 36
3.4 Store buffer graph after fixing the write-def-chain (WDC) from Figure 3.3 38
3.5 Control flow graph of the new program after transforming the store buffer graph from Figure 3.4. The nodes represent the new program locations now. Edges have been replaced with the respective SC operation mimicking their effect under SC. 43

4.1 Architecture of Weak2SC – external components are excluded 67
4.2 Work Stealing Queue by Arora et al. [ABP98] 71
4.3 LLVM IR code after compilation of the code in Figure 4.2. Includes annotation for required fences. It shows only excerpt of the method *popBottom*. 73
4.4 Store buffer graph of *pushBottom* method under TSO 76
4.5 Store buffer graph of *pushBottom* method under PSO 76
4.6 Excerpt of the generated Promela model for the program in Figure 4.3. 80
4.7 Promela model generated for the TSO store buffer graph for the method *pushBottom* in Figure 4.3. 83
4.8 Excerpt of generated global (left) and local (right) state definition for Arora et al. work stealing queue[ABP98] 86
4.9 Declaration of constants, functions and predicates for the encoding of the store buffer graph as a transition system. 88
4.10 Excerpt of generated transition system using local state encoding for Arora et al. work stealing queue[ABP98] 89
4.11 Promela programs based on operational memory models as proposed in [TMW13] 92
4.12 Communication between program process and OMM process; 1. read 2. write 3. fence and 4. CAS 93
4.13 Generated program model based on operational memory models of *pushBottom* method. 94

5.1 Visualizing TSO-to-TSO linearizability as a mapping of events from a concrete TSO history to an abstract TSO history. 105
5.2 Visualizing TSO-to-SC linearizability; store buffer delays are ignored by abstract histories. 108
5.3 Visualizing TSO linearizability; last flush after response becomes the new response. 110

6.1 Linearization points in concurrent executions and the corresponding modification of the abstract state. 124
6.2 Abstract double ended queue specification in Promela. All operations are atomic. 126
6.3 Instrumented queue implementation for consistency checks against abstract data structure. Program model generated by Weak2SC based on reduction from store buffer graphs. 127

6.4 Instrumented queue implementation for consistency checks against abstract data structure. Program model generated by WEAK2SC for the use with an operational memory model. 128
6.5 Potential linearization points in an array based container data structure. 130
6.6 Overlapping of method execution and intra-process reordering. 131
6.7 Overall history checking procedure. Exploration triggers history checks via assertion whenever all processes have run to completion. 142
6.8 Stepwise checking procedure of a concurrent history. Sequential data structure and the history are input parameters to the check. 143
6.9 Visualizing non-atomic refinement and the linearization status. 148

B.1 LLVM IR code after compilation of the code in Figure 4.2. Shows variable definition and the methods *pushBottom* and *popTop*. 197
B.2 LLVM IR code after compilation of the code in Figure 4.2. Shows the method *popBottom*. 198
B.3 C code of two transactions from the transactional memory implementation, TML by [DDS+10]. Method *proc13* implements *begin, write(x,1) ,commit*; Method *proc33* implements *begin, read(x,lx), read(y,ly), commit*. The implementation also shows the required fence (*sync_synchronize*). Please note that we combined begin, read, write and commit operations into one operation, in order to be able to take the reordering across method boundaries into account. 199

List of Tables

6.1 Verification results for the transformed programs (tso2sc, pso2sc) and based on an operational memory model (tso, pso). Brackets state the memory model for which a program was fenced. 134
6.2 Verification results for full state space exploration error: #i are lines of LLVM IR instructions (“/” separated for each method); #n number of nodes in the sb-graph (“/” separated for each sb-graph); #s the number of states explored t is the time in seconds. 137
6.3 Verification results for full state space exploration error: #i are lines of LLVM IR instructions (“/” separated for each method); #n number of nodes in the sb-graph (“/” separated for each sb-graph); #s the number of states explored t is the time in seconds. 138
6.4 Number of proof steps in the theorem prover KIV for the linearizability proofs of the above case studies. 169

1 Introduction

Today most modern processors are multicore processors, i.e., a single CPU consists of multiple processor cores, which are interconnected with each other and work in parallel. In the past, CPU performance could be increased by increasing its frequency. However, a frequency increase comes along with an increase of heat emission by the CPU. For the time being, the heat problem was tackled by miniaturizing the transistors of a CPU (up to a few layers of atoms). This reduced heat emission and allowed for higher frequencies. Thus, newer generations of CPUs could improve performance over earlier generations. This development slowed down over the recent years as the physical limits were approached closer and closer. In order to further increase performance, CPU manufacturers started to combine a growing number of processor cores on a single CPU. As a result of this development, today, multicore processors are ubiquitous in most modern devices.

In order to fully use the offered performance of multicore processors, software has to be concurrent, i.e., allowing tasks to be performed in parallel as much as possible. However, often tasks or more generally processes need to agree on shared data, i.e., they need to synchronize on the state of shared data, e.g., the first element in a queue, even throughout conflicting accesses, e.g., two processes trying to remove the first element simultaneously. One way to avoid such conflicts is to use locks. Locks protect shared data by giving one process (owner of the lock) exclusive access to the data. Processes who do not own the lock must wait until they successfully acquire the lock. Thus, conflicts are avoided by serializing access to the shared data. A simple way to use locks is to acquire the lock at method invocation and release it at method return in all potentially conflicting methods. However, this is very ineffective, since usually only a few lines of code of a method are critical and thus,

serializing more than these few lines is often an unnecessary overhead. In order to minimize the time in a critical section and thereby maximizing throughput, a whole class of data structure implementations [Tre86, HW90, MS96, ABP98] relies on fine-grained synchronization. These data structures do not use locks, but single atomic instructions in order to perform critical changes.

While fine-grained concurrency increases performance of data structure implementations, it also comes at a cost. The algorithms implementing the data structures are inherently complex and bugs are easily introduced into their implementations. The latter is the case, because it is no longer relatively large (critical) sections of code that are interleaved with other sections during execution, but single instructions with other instructions. Due to the fine-grained interleaving of different processes, there are many more possible executions as each possible interleaving is a different execution. These executions have to be considered during implementation of the data structure. However, there are simply too many possible executions to consider them all during development of an algorithm. Several processes can perform different operations and each new combination of a set of processes executing a sequence of operations allows for new executions. To make things worse, infinite executions are also possible or at least must be considered, since a bug may also occur late during an execution.

Testing each possible execution one by one is not sufficient as one might never finish with this task. Even though testing is often automated to a large extend, it cannot show correctness, because the tests reveal only a fraction of the possible behavior of an implementation. Instead developers use formal verification in order to prove correctness of concurrent data structure implementations. One such method is model checking [CGP01, Cla08] where the state space of a program is systematically explored and checked for requirements. If the exploration finishes and no violation of the requirements was found, then the program is proven to be safe. Otherwise, a counterexample trace is reported that violates the requirement. However, even with model checking it is difficult to prove correctness of concurrent data structures, simply because the state space of most concurrent data structures is not finite. Thus, it cannot be checked exhaustively unless abstractions to the state space are applied, which would make the state space finite. Nevertheless, testing and model checking are useful tools for finding bugs, but when it comes to show ultimate correctness of a concurrent data structure, then a correctness proof is required. The latter formally proves that an implementation behaves correct w.r.t. a correctness property that defines its legal behavior.

Reasoning about the correctness of such fine-grained concurrent data structures can be challenging, since lots of fine-grained steps tend to make the verification verbose and complex. Several correctness properties have been proposed, e.g., Serial-

izability [Pap79], Linearizability [HW90], Quiescent Consistency [HS08] including different variations of the mentioned properties. Among these, Linearizability has established as a de-facto standard correctness criterion for concurrent data structures. Different verification techniques have been developed in order to formally prove, check or test the correctness of an implementation. Some prove a refinement between an abstract specification and an implementation (usually by proving a simulation relation) [Hes07, DSW07], some try to apply reduction techniques [EQS+10] and others check or test for trace-equivalence [LCLS09, VYY09, BDMT10].

However, all of the above mentioned correctness criteria and verification approaches have one assumption in common: they assume sequentially consistent (SC) execution semantics [Lam79], i.e., programs of concurrent processes are executed in a non-deterministically interleaved manner. This is not in line with the behavior observable on modern multicore processors. These allow for out-of-order execution, i.e., instructions appear to execute in a different order than the program order. Thus, there is a gap between the program behavior that is usually assumed by the verification approach and the program behavior that can actually occur on a multicore processor. Without further ado, this gap renders many verification results unsound.

1.1 Concurrency on Modern Multicore Processors

In the following, we elaborate about concurrency on modern multicore processors and in particular their weak execution semantics, which motivate this thesis. Furthermore, we discuss why the assumption of sequentially consistent executions semantics is a problem for existing verification techniques. Later on, we introduce the contribution of this thesis. Finally, the remaining chapters of this thesis are outlined.

1.1.1 Weak Memory Models

Modern multicore processors provide weaker execution semantics than the often assumed sequential consistency [Lam79]. Executions semantics are captured by memory models as it is the architecture of a CPU that dictates the provided semantics. Thus, semantics of multicore processors are known to be captured by *weak memory models*, weak with respect to sequential consistency (SC).

The architecture of a CPU dictates the execution semantics in the sense that it organizes how a single processor core accesses the memory in order to read from it or write to it. Figure 1.1 illustrates a sequentially consistent architecture. Several processor cores are connected to the memory via a non-deterministic switch. Only if the switch establishes a connection between a core and the memory, the core

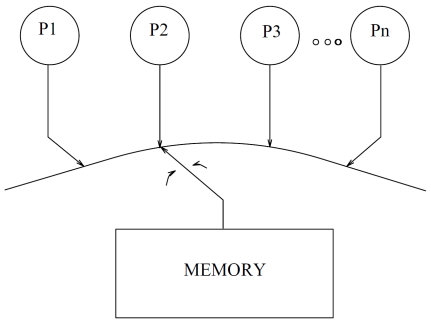

Figure 1.1: Programmer intuition of concurrency [AG96]

can execute memory instructions. Otherwise, a core has to wait. Memory access is atomic and the effect of each instruction becomes immediately visible to whoever is next after it has been performed. Since each core executes a sequential program, the resulting execution is always a non-deterministic interleaving of the sequential programs run by all processor cores. This architecture is artificial and no common multicore processor works this way, but still it is what is usually assumed when people think about concurrency or even when they verify concurrent software.

One of the most common processors is based on the x86-architecture [Int12], which is illustrated in Figure 1.2. It provides store buffers to each core, an optimization technique originally developed for single core processors. A store buffer is basically a FIFO-queue. It allows processor cores to seemingly perform a write instruction, but instead of writing the new value to the memory, the value and the location it should be written to are temporarily stored in the store buffer. The store buffer is emptied by flushing its entries to the memory at an indefinite later point in time. Meanwhile, the core can further execute its program. If it tries to read a location that was previously written and that is still present in the store buffer, then it takes the latest value from the store buffer. Otherwise, it takes the value from the memory. Please note, that the direct access to the memory (read or write) is considerably slower than the access to the store buffer of a core. This is not only the case because of slow memory hardware, but also because of potential contention with other cores trying to access the memory simultaneously.

As a consequence of store buffers and the delay it imposes on writes, instructions may appear is if they were executed out-of-order. As an example take the Litmus test in Figure 1.3. Litmus tests in the context of memory models are simple and often minimal programs revealing effects, which can be observed due to weak execution semantics. The Litmus test in Figure 1.3 detects reordering of writes with later reads. Two shared variables x and y are initially 0. The first process writes 1 to variable

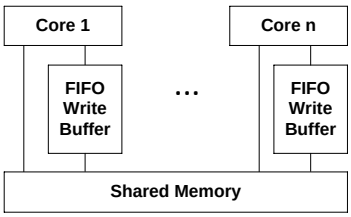

Figure 1.2: Illustration of the x86-architecture

$Initially : x = 0 \wedge y = 0$

Process 1	Process 2
$write(x, 1);$	$write(y, 1);$
$read(y, r1);$	$read(x, r2);$

$r1 = 0 \wedge r2 = 0$ possible

Figure 1.3: Litmus test for reordering of writes with later reads

x and then reads the value of y to a local register $r1$. The second process writes 1 to variable y and then reads the value of x into its local register $r2$. In all possible interleavings under SC semantics, the value of at least one register, either $r1$ or $r2$ or both, will be 1. However, due to store buffers in x86 processors, the writes can be delayed. A read instruction can execute while the previous write is still contained in the store buffer. Since it is a different location than the previously written one, it has to fetch the value from the memory. If the write of the other process was not flushed yet, the read will be observe the value 0. The same holds for the read of the other process. As a consequence of the weak execution semantics due to store buffers, the outcome ($r1 = r2 = 0$) becomes possible, which is not possible under SC semantics.

From the example, we can see that weak execution semantics can cause behavior, which cannot be explained by SC semantics and which can be observed at runtime. The fact that weak memory models do add behavior that cannot be observed under SC renders most verification approaches unsound, because they widely assume SC semantics. However, such effects are only observable in a concurrent setting. Thus, verification approaches for concurrent software must consider weak memory models instead of just assuming SC.

In the past, there has been a lot of research on weak memory models. Early work [AG96] on weak memory models tried to characterize the possible effects and provide an intuition of what kind of behavior must be expected from various types of weak memory models. Over the years, the weak memory models have been formalized [AH93, HKV98, SSO$^+$10, AFI$^+$08, MMS$^+$12, DTDW13] in several ways and their relationship has been studied [HKV98, Alg12, DTDW13]. There are four generic types of memory models, which can be characterized by the order that they preserve in interleavings of different processes. Most other models are variants of the four generic models. Sequential consistency (SC) preserves full program order, but is not implemented on actual hardware. The most common architecture (x86 [Int12]) preserves the order of writes, therefore it is known as *Total Store Order* (TSO). It allows writes to be reordered with later reads. A further relaxation of TSO

is *Partial Store Order* (PSO). In contrast to TSO, it only preserves the order of writes to each location, but allows writes to different locations to be reordered with each other. It is rarely used in practice, e.g., by the SPARC processor [SPA92]. An even more aggressive reordering is offered by the *Relaxed Memory Order* (RMO). RMO allows basically any instructions that operate on different memory locations to be reordered. The order of instructions operating on the same location is preserved. The Power [IBM15] and ARM [ARM13] architectures implement a variant of RMO.

1.1.2 Towards Program Correctness

One of the strongest motivations for the vast amount of research is to avoid execution results under weak memory that are impossible under SC. If a program does not reveal any observable behavior under a particular weak memory model, then it is robust against this model [BMM11]. Robustness of a program can be achieved by placing fence instructions all over the program. A fence instruction is a special instruction that blocks the processor from executing any further instructions until the store buffer is empty. Thus, it prevents reordering of previously executed instructions with the instructions yet to be executed. Therefore it is also known as a memory barrier. However, a fence instruction is an expensive instruction in terms of performance, because it forces the processor to wait. Thus, one wants to place as few fences as possible in order to not hurt program performance.

Several approaches [BAM07, AMSS10, KVY12, AAC+13] have been developed to identify a (minimal) fence placement for a given program that avoids non-SC executions. However, these approaches are not practical, as they require two major steps for verification. First, it must be ensured that the concurrent program is correct under SC, which can be a difficult task itself because of the state explosion problem [Val98, HKV02] and second, the program potentially enriched by a few fences must be verified to conform to its SC behavior. The latter part can be considered as the more difficult one. This is especially true, because it requires reasoning about a program in its low-level representation (single instructions) rather than the high-level language, e.g., C or C++, in which it was likely written. Memory models define the behavior of a processor and a processor executes single instructions, not high-level language statements. Thus, a high-level program must be compiled to a low-level representation in order to be able to apply the definitions from memory models and furthermore, in order to rule out potential compiler optimizations. The latter can modify the program in many different ways and taking all variants of a program into account that a compiler can produce is simply not feasible.

Anyway, it is not always necessary to add fences to a program in order to achieve robustness against weak memory models. In particular, if a program is known to be

data-race-free, then it is also robust against weak memory models, since its observable behavior in this case is known to be equivalent to the behavior under SC [CS10]. Thus, for data-race-free programs reasoning about memory models is obsolete. A data-race occurs if two concurrent programs read or write shared data and at least one of them writes to it. Data-race-freedom of a program is usually achieved by design, by adding synchronization (e.g. locks) where data races can occur. Thus, data races are avoided by ensuring that the data is owned by one process exclusively at the time of access to it. Prominent programming languages such as Java and Rust rely on data-race-freedom. Java guarantees SC semantics for Java programs if they are properly synchronized [MPA05]. Rust goes even one step further and forces developers to annotate ownership in the program. It also checks for potential data races at compile time and reports them to the developer. A more general approach to showing data race freedom is given in [CS10]. It introduces a reduction theorem, which if can be shown for a program to hold, implies that the program is data race free and thus is robust against TSO. However, many programs and in particular data structure implementations have intentional data races. Programs with intentional data-races usually rely on fine-grained synchronization such as fence instructions, atomic read-modify-write instructions or none at all. For these program, verification must consider the underlying memory model.

Considering all non-SC program behavior as harmful is stricter than necessary. Just because non-SC behavior is observable for a program does not make the program incorrect. A more intuitive correctness criterion is *Linearizability* [HW90]. The operations of linearizable data structure appear to be atomic at some point in time between invoke and return, the linearization point. Thus, the implementation behaves as if it was a sequential one. Linearizability is the de-facto standard correctness criterion for concurrent programs, but the original definition of linearizability assumes SC [HW90]. There has been quite some effort towards transforming the original definition into a definition that is aware of weak memory models [BGMY12, GMY12, BDG13, DSD14, DSGD17]. The closest definition to the original definition is the one in [GMY12], which was initially defined for TSO and then extended to other memory models [BDG13]. The reason why the original definition is no longer sufficient and why there are several new definitions of linearizability are the invokes and returns of an operation. Under SC, an invocation and a return are each a single atomic step and express the boundaries of an operation or method. Considering the delay of writes inherent to weak memory models, these boundaries are not as clear anymore. The definitions above vary mainly in their interpretation of these boundaries. However, it is still unclear which definition will establish as the new memory model aware de-facto standard and the differences between these criteria are subtle. Therefor, it should not surprise that there

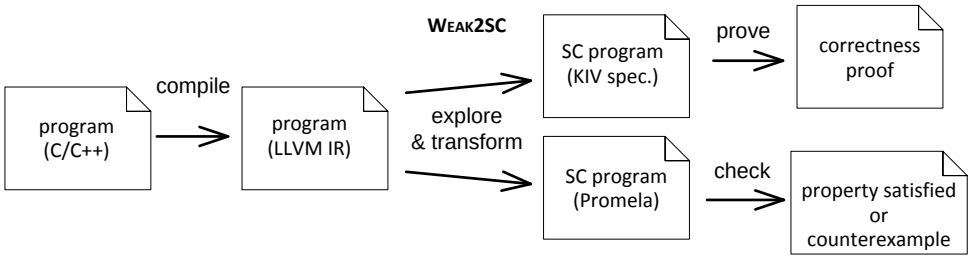

Figure 1.4: WEAK2SC, a memory model-aware verification approach

are only few attempts of verifying linearizability in the presence of weak memory models [BDMT10, TMW13, TW14].

1.2 Contributions

Concurrent software verification requires tools that are aware of the underlying memory model in order to provide sound results. Most of the available memory model-aware verification tools [PD95, BAM07, AMSS10, KVY12, AAC+12] are dedicated to a particular property or are limited in their applicability. Only few, mostly model checking tools (CBMC [CBM], Nidhugg [AAA+15]), can be considered general purpose verification tools.

This thesis introduces a general purpose approach for concurrent software verification together with an implementation of it, WEAK2SC. The key idea behind it is to transform a program combined with the effects due to weak memory models into a new program destined to execute under SC semantics. The transformation is based on a reduction that involves symbolic exploration of possible store buffer states. This information is combined with the program control flow and can be represented in terms of a graph, which eventually is encoded into a new SC program. According to our bisimulation proof, the new program is behaviorally equivalent to the original program under weak memory semantics. Thus, existing verification tools (those which assume SC) can be used for verification of the new program. Because of the behavioral equivalence, the results also hold for the original program under weak memory. The proposed approach allows developers to generally use a verification tool of their choice. However, the approach works best, if the underlying program transformation is automated. WEAK2SC implements the transformation to Promela [Hol03], the input language to the model checker SPIN, and a predicate logic encoding as is required by the theorem prover KIV [EPS+14]. An extension to other output formats and languages is possible, if these can express non-determinism.

Figure 1.4 gives an overview of the approach. First, a C or C++ program is

compiled into a low-level representation, since we later want to reason about weak memory effects. The compilation is done by the LLVM compiler framework [LA04], which produces LLVM IR code. In contrast to assembly, the LLVM IR code still contains information from the original program, e.g., variable names and types and thus, helps to understand the low-level version of the original program. Weak2SC takes the LLVM IR program and performs a symbolic exploration, in order to determine the effects of weak memory models to each process of the program. The exploration is fast, because no concurrency has to be considered within this step. Out of the thus obtained symbolic states, Weak2SC generates a new SC program that is behaviorally equivalent to the original one under weak memory models. Depending on the verification goal, the new SC program can be either the input of a model checker or a theorem prover.

We evaluate our approach on a number of case studies by extending different verification approaches [Fla04, VYY09, DSW11b] to the weak memory model setting. In particular, we experiment with model checking data structure implementations against abstract atomic specifications. The idea behind this approach is similar to [Fla04] and thus is not new, but it has never been applied in a weak memory setting before and we are the first to achieve this. In addition, we present history-based model checking, which is perhaps the most natural way of verifying correctness, since correctness criteria are usually defined via sets of correct histories. Inspired by the ideas in [VYY09], we extended the model checker Spin with capabilities to record histories and for their validation against arbitrary sequential implementations of data structures. Furthermore, we prove linearizability of a work-stealing queue [ABP98] and the Burns mutex algorithm [BL80], each twice: First by applying our reduction to them and second, by encoding the weak memory semantics explicitly into the program behavior. Finally, we compare the impact of our reduction to the proof effort. In order to achieve this, we instantiated the proof obligations from [DSW11b] for our case study. The proofs are fully mechanized in the theorem prover KIV [EPS+14]. Usually, proofs, involving weak memory effects, are done manually or are applied to trivial programs, e.g., Spinlock, which is used as a mutex algorithm in Linux [BC05]. With our proof [TMW13], we were one of first to fully mechanize such a proof to a non-trivial program like the work-stealing queue by Arora et al. [ABP98].

Our approach is inspired by existing reductions [ABP11, AKNT13], but is limited to a certain class of programs, which we discuss in Chapter 3. The limitation allows our approach to produce comparably concise new programs, whereas the existing approaches [ABP11, AKNT13] are not restricted, but produce complex new programs. The complexity stems from a *general encoding* of weak execution semantics as part of the new program. In [AKNT13], a complex abstract machine simulates the steps of the underlying memory model. In [ABP11], the new program contains a lot of

auxiliary variables in order to maintain information about potential store buffer entries. A later verification, be it a proof or state space exploration, has to consider all this additional encoding, which makes it difficult. Our approach barely requires any auxiliary variables (at most one per write in a loop of the original program) and thus limits this type of additional effort.

1.3 Overview

The thesis is structured as follows. In Chapter 2, a more detailed explanation of (weak) memory models is given. It includes various common weak memory models. For those memory models supported by the presented approach, a formal definition is given. Chapter 3 introduces the reduction from weak semantics to sequential consistency. We first explain the symbolic exploration of the low-level programs and the transformation, which generates a new program out of the explored states. Furthermore, a soundness proof for our approach is presented and discussed. Weak2SC, the implementation of the presented approach, is introduced in Chapter 4. An architecture of Weak2SC is given as an overview and technical aspects of the transformation are discussed. In particular, the chapter also discusses how Weak2SC can be extended to other output languages. In Chapter 5, variants of linearizability definitions and their sometimes subtle consequences are discussed. Furthermore, the chapter provides background on model checking and proof methods for linearizability. The previously mentioned model checking and proof methods for linearizability and their adaptations to weak memory models are introduced in Chapter 6. The requirements and the main ideas are provided, followed by experimental results and a discussion. The related work is distributed along the chapters where it fits to the context. Finally, Chapter 7 summarizes the thesis, gives conclusions and lays out possible future work.

2 Memory Models

Programs have semantics, as well as each statement of a program and each instruction, of which a statement may be composed of. Sequential program developers usually don't even have to think about semantics, as programs are simply expected to be executed sequentially, one statement of the program at a time, one instructions after the other. The result of a sequential program is the result of executing all program instructions in program order. Thus, we are used to abstract the implementation details and think of a program in its abstract meaning, e.g., a complete push operation of a stack rather than the single lines of code implementing it.

However, instructions are often executed non-atomically and they are not necessarily executed in program order because of performance optimizations. Actual processors implement various performance optimizations, which can relax program order, but do not affect the outcome of a sequential program. Thus, details from the underlying hardware are hidden from developers. In other words, a developer has an abstract view of what is actually going on in a processor and it is safe to not care about the abstracted details, at least for developers of sequential programs. The reason why it is safe is, because all common processors provide sequentially consistent semantics for sequential programs. Thus, all possible program results can be explained by sequential executions with respect to program order. However, this is not true for concurrent programs that run on multicore processors as these have different, weaker semantics. For concurrent programs, it is widely assumed that the result of an execution can be explained by an interleaving of the sequential programs run by each process. This is not true for modern multicore processors and we have given an example for this in Figure 1.3 in Section 1.1. Modern processors reveal some of their internals to program developers by seemingly reordering program

instructions. The reordering can lead to unexpected results or even program failure. It is thus crucial to understand the semantics of these processors, especially in the context of concurrent program verification.

Not long ago, the semantics of processors were given by informal descriptions, usually provided by the vendors. People soon realized that informal descriptions are often ambiguous and thus do not provide the precision needed by developers of, e.g., operating systems, programming languages or verification tools. Vendors adapted to these needs by adding Litmus tests to their descriptions [Int12] or by providing semi-formal descriptions of the semantics [Int, SPA94]. However, these can still be ambiguous [AFI+08]. Especially for concurrency verification, ambiguities in the description of semantics may render verification unsound.

Memory models formally define the semantics of a processor. There are various ways to define a memory model, but we can generally distinguish between two types of memory models: *axiomatic* or *operational*. An axiomatic memory model definition defines axioms, usually for various orders that have to be preserved in an execution, e.g., order of writes. The set of preserved orders characterizes all possible execution orders of arbitrary programs on a particular memory model. As different memory models provide different semantics, it is important to study the relationships between different memory models as it was done in [AH93, AM06, ABBM10, Alg12]. Usually, the differences between two memory models can be characterized by one or several orders that are preserved by one memory model but not by the other.

An operational memory model is basically a formal programming language consisting of a set of operations. Each operation is a formal definition of the effect of either a processor instruction, e.g, read or write or an internal step of the processor, e.g., a flush. Operations are usually modelled as atomic steps, e.g., a write enqueues a written value to the store buffer atomically. Operational memory models are practical for model checking [Cla08], because a program implicitly defines which operations are possible at a particular state and these just have to be explored. In verification, operational memory models are widely used and have been around for the last two decades [PD95, BP09, SSO+10, TMW13].

The effects of a memory model are known for processor instructions, but cannot be stated a priori for statements of high-level languages like C or C++. The reason for this is simply that there are too many ways of how a compiler can transform the original high-level program to a low-level program (usually assembly). Furthermore, one line of code in a high-level language can correspond to many instructions of a low-level program and these may depend on the code optimizations that the compiler performs throughout compilation. Since memory accessing instructions are the most expensive ones in terms of performance, compilers try to reduce the number of them and optimize them as much as possible. Memory models define the semantics of

these instructions and low-level programs state memory access explicitly. Thus, in order to reason about programs in a weak memory setting, we need to consider low-level programs.

Perhaps the most important insight from the early research is that weak memory models do not affect the observable behavior of a program if it is data-race free (DRF) [AH90, AG96]. A data-race occurs if two processes compete for a shared resource like a variable and at least one of them attempts to write to it. Obviously, sequential programs are data-race free by definition. For concurrent programs data-races can be avoided by using synchronization primitives like locks. In fact, the Java memory model [MPA05] was built around this property as it aims to guarantee sequential consistency for correctly synchronized programs. However, several flaws were identified in the implementation of the Java memory model [SA08, Sev09]. The latter shows that it is difficult to hide the effects of weak memory models. Besides that, there are plenty of concurrent data structure implementations [Tre86, MS96, HW90], which rely on fine-grained synchronization primitives and have data-races intentionally for performance reasons. Such implementations do not benefit from DRF guarantees and their implementations must take the semantics of weak memory models into account. The latter motivates our research.

In the following, we will define programs and their semantics under SC, TSO and PSO. These build the foundation of our later definitions and of our reduction in Ch. 3. Parts of it have also been used in our previous publications [WT15, TW16] and were revised for this thesis.

2.1 Programs

Reasoning about the effects of memory model requires reasoning about the semantics of single processor instructions. Thus, we have to deal with low-level programs that are the result of compiling a high-level language. In the following, we first introduce an assembly-like low-level language. We will use it in order to define the semantics for the memory models SC, TSO and PSO. In Section 3, we will also use it in order to define a reduction and carry out our proofs.

For programs, we assume a set Reg of registers local to processes and a set of variables Var, shared by processes. For simplicity, both take just integers as values. A set of labels $\mathcal{L}$ is used to denote program locations. The following definition gives the grammar of programs.

Definition 1. *A* sequential program *(or process) P is generated by the following grammar:*

$$P \quad ::= \quad \ell : read(x, r) \mid \ell : write(x, r) \mid \ell : write(x, n) \mid$$

$$\ell : r := expr \mid \ell : fence \mid \ell : skip \mid P_1;\ P_2 \mid$$
$$\ell : if\ (bexpr)\ then\ P_1\ else\ P_2\ fi \mid$$
$$\ell : while\ (bexpr)\ do\ P_1\ od \mid \ell : goto\ \ell'$$

where $x \in Var$, $n \in \mathbb{Z}$, $r \in Reg$, $\ell, \ell' \in \mathcal{L}$ and $bexpr$ is a boolean and $expr$ an arithmetic expression over Reg and $\mathbb{Z}$.

Definition 2. *A* concurrent program S *is defined as* $[P_1 \ || \ \ldots \ || \ P_n]$ *where all* $P_i, 1 \leq i \leq n$, *are sequential programs.*

The language defines statements for reading or writing to shared variables in memory. The value read from a shared variable x is stored in register r. The value written to x can be either the value of register r or a constant n. Assignments modify the value of a register r by assigning the value of an arithmetic expression $expr$ to it. The expression is evaluated over Reg and $\mathbb{Z}$. A $fence$, also known as memory barrier, is a statement that blocks until store buffers are emptied. Its purpose is to ensure that potentially delayed writes before the fence are not reordered with statements following the fence. A $skip$ is an empty statement, i.e., it does not modify any value. Furthermore, the language defines sequential composition of statements, if-$then$-$else$ statements for conditional execution, a $while$ statement for looping and a $goto$ statement for jumping to a particular program location ℓ'.

All program statements have their own unique program location ℓ. Out of the program text, we can derive a function $suc : \mathcal{L} \to \mathcal{L}$ denoting the *successor* of a label ℓ in the program. Similarly, we use functions suc_T and suc_F for the successors in if and $while$ statements (on condition being $true$, or $false$ respectively). We assume the first statement in a sequential program to have label ℓ_0.

2.2 Parameterized Semantics

Memory models differ mainly in their memory access. Some access it directly and others use store buffers in order to write to memory. Thus, we introduce a semantics that is parameterized by a store buffer type and a few predicates. These predicates capture the characteristics of the memory model, while the rest of the semantics remains equivalent among the memory models.

Processes have a local state represented by a function $reg : Reg \to \mathbb{Z}$ (registers) together with the value of a program counter, and a store buffer. Concurrent programs in addition have a shared global state represented by a function $mem : Var \to \mathbb{Z}$ (shared variables). We use the notation $mem[x \mapsto n]$ to stand for the function mem' which agrees with mem up to x which is mapped to n (and similar for other functions). A memory model is fixed by stating how the writing to and reading

from global memory takes place. Memory models use *store buffers* to cache values of global variables. Such store buffers take different forms: in case of TSO it is a sequence of pairs (variable,value); in case of PSO it is a mapping from variables to sequence of values; in case of SC the store buffer is not existing (which we model by a set which is always empty). In the semantics, the store buffer is represented by sb. Since the type of store buffer is different among memory models, we will provide it later, as a part of the parameter to the generalized semantics presented here.

Definition 3. *A* memory model $MM = (type, init, read, write, flush, fence)$ *consists of*

- *the* type *of the store buffer, and*
- *formulae for initialization, read, write, flush and fence operations ranging over* mem, sb *and* reg.

We assume all registers and variables to initially have value 0. Later, we define the semantics of programs by assigning a predicate to every statement stm according to the given memory model, i.e., we fix $[\![stm]\!]_{MM}$. We define $Ops(P)$ to be the set of all such predicates, which make up the *operations* of the program P together with an operation predicate $\ell : \mathit{flush}$ for all $\ell \in \mathcal{L}$.

We describe the semantics of program operations by logical formulae over sb, reg and mem. In this, primed variables are used to denote the state after execution of the operation. A formula like $(x = 0) \wedge (reg'(r_1) = 4)$ for instance describes the fact that currently x has to be 0 and in the next state the register r_1 has the value 4 (and all other registers stay the same). A state s for a process consists of a valuation of the variables pc (the program counter), sb and reg. We write $s \models p$ for a formula p to say that p holds true in s. For convenience, we assume all variables keep their value that are not mentioned in the predicate, e.g., if the predicate does not mention sb, reg, mem, then this implies $(sb = sb' \wedge reg = reg' \wedge mem = mem')$.

Definition 4. *The semantics of a program with respect to a given memory model MM are*

$$
\begin{aligned}
[\![\ell : read(x,r)]\!]_{MM} &\mathrel{\widehat{=}} pc = \ell \wedge read_{MM}(x,r) \wedge pc' = suc(\ell)\\
[\![\ell : write(x,r)]\!]_{MM} &\mathrel{\widehat{=}} pc = \ell \wedge write_{MM}(x,r) \wedge pc' = suc(\ell)\\
[\![\ell : write(x,n)]\!]_{MM} &\mathrel{\widehat{=}} pc = \ell \wedge write_{MM}(x,n) \wedge pc' = suc(\ell)\\
[\![\ell : fence]\!]_{MM} &\mathrel{\widehat{=}} pc = \ell \wedge fence_{MM} \wedge pc' = suc(\ell)\\
[\![\ell : \mathit{flush}]\!]_{MM} &\mathrel{\widehat{=}} \mathit{flush}_{MM}\\
[\![\ell : r := expr]\!]_{MM} &\mathrel{\widehat{=}} pc = \ell \wedge r' = expr \wedge pc' = suc(\ell)\\
[\![\ell : goto\ \ell']\!]_{MM} &\mathrel{\widehat{=}} pc = \ell \wedge pc' = \ell'
\end{aligned}
$$

$$
\begin{aligned}
[\![\ell : \mathit{skip}]\!]_{MM} &\mathrel{\widehat{=}} pc = \ell \wedge pc' = \mathit{suc}(\ell) \\
[\![\ell : \mathit{if}\ (\mathit{bexpr})\ \mathit{then}\ P_1\ \mathit{else}\ P_2\ \mathit{fi}]\!]_{MM} &\mathrel{\widehat{=}} (pc = \ell \wedge \mathit{bexpr} \wedge pc' = \mathit{suc}_T(\ell)) \\
&\quad \vee (pc = \ell \wedge \neg \mathit{bexpr} \wedge pc' = \mathit{suc}_F(\ell)) \\
[\![\ell : \mathit{while}\ (\mathit{bexpr})\ \mathit{do}\ P\ \mathit{od}]\!]_{MM} &\mathrel{\widehat{=}} (pc = \ell \wedge \mathit{bexpr} \wedge pc' = \mathit{suc}_T(\ell)) \\
&\quad \vee (pc = \ell \wedge \neg \mathit{bexpr} \wedge pc' = \mathit{suc}_F(\ell))
\end{aligned}
$$

The semantics definition encodes each program statement as a predicate. Each predicate encodes the control flow of the program as $pc = \ell \wedge pc' = \ell'$, where ℓ is the location before the actual operation and ℓ' the one after. The only exception to this is the flush_{MM} operation. A flush removes entries from the store buffer, but it is not an explicit statement of a program. It is not restricted to any particular location, but can occur whenever the store buffer is not empty. Thus, it is always an alternative to each of the other operations. The semantics provides placeholders for the characteristic predicates of the respective memory model. For instance, a $\mathit{read}(x, r)_{MM}$ represents the read semantics specific to the memory model MM. The same holds for $\mathit{write}_{MM}(x, r)$, $\mathit{write}_{MM}(x, n)$, fence_{MM}, and flush_{MM}. The possible values for MM are defined in the following sections. Besides these memory model specific operations, we also define the local operations, i.e., operations that read or modify the state of only one process, the one executing them. Local operations are assignments, $\mathit{if} - \mathit{then} - \mathit{else}$, while or goto statements.

Now that we have defined the parameterized semantics, we can define the transition system of a program. First, we define a local transition system, i.e., the transition system of a sequential program which corresponds to a single process. This local transition system abstracts from its environment (i.e., other processes running concurrently) in that it assumes arbitrary states of the global memory (which could be produced by other processes). We call this an *open* semantics. We start with a local transition system, because our later reduction is also applied locally to each method of a program. Out of the local transition systems, we can construct a global transition system as we will see.

Definition 5. *The local transition system of a sequential program P on memory model MM, $\mathit{lts}_{MM}(P) = (S, \rightarrow, S_0)$, consists of*

- *a set of states $S = \{(pc, sb, reg) \mid pc \in \mathcal{L}, sb \in \mathit{type}_{MM}, reg \in (\mathit{Reg} \to \mathbb{Z})\}$,*
- *a set of initial states $S_0 = \{s \in S \mid s \models \mathit{init}_{MM} \wedge s \models (pc = \ell_0)\}$,*
- *a set of transitions $\rightarrow \subseteq S \times \mathit{Lab} \times S$ such that for $s = (pc, sb, reg)$ and $s' = (pc', sb', reg')$, we have $s \xrightarrow{lab} s'$ iff $\exists\, op \in \mathit{Ops}(P), \exists\, mem, mem' :$*

$((s, mem), (s', mem')) \models op$ *and* $label(op) = lab$. *For such transitions, we use the notation* $s \xrightarrow[mem,mem']{lab} s'$.

Since the transition system is local, a state is represented by a tuple consisting of pc, sb, reg, i.e., the current program location represented by pc, the current store buffer content sb and the current valuation of registers reg. A set of initial states must satisfy $init_{MM}$, the memory model specific initialization condition. Furthermore, initial states start at program location ℓ_0. Transitions connect states s, s', iff an operation $op \in Ops(P)$ exists and we can provide memory values mem, mem' such that $((s, mem), (s', mem')) \models op$. Please note that we do not provide the labels $label(op)$ explicitly here. We label transitions in a particular way that helps us with our later proofs. However, at this point, the labels would rather confuse than clarify, because we have not introduced our reduction yet. We refer to Section 3.3.1 for the label function.

Processes typically run in parallel with other processes. The semantics for parallel compositions of processes is now a *closed* semantics already incorporating all relevant components. In the following, we define it for two processes. The initial global state mem_0 assigns 0 to all global variables.

Definition 6. *Let* $P_j, j \in \{1, 2\}$*, be two sequential programs and let* $(S_j, \rightarrow_j, S_{0,j})$*, be their process local (i.e., open) labelled transitions systems for memory model* MM.

The closed MM semantics *of* $P_1 \mid\mid P_2$*,* $lts_{MM}(P_1 \mid\mid P_2)$*, is the labelled transition system* $(S, \rightarrow, S_0)$ *with*

- *a set of states* $S \subseteq \{(mem, s_1, s_2) \mid s_j \in S_j, j \in \{1, 2\}\}$,
- *a set of inital states* $S_0 = \{(mem_0, s_{0,1}, s_{0,2} \mid s_{0,j} \in S_{0,j}, j \in \{1, 2\}\}$,
- *and a set of transitions* $s = (mem, s_1, s_2) \xrightarrow{lab} s' = (mem', s_1', s_2')$ *when* $(s_1 \xrightarrow[mem,mem']{lab} s_1' \wedge s_2 = s_2')$ *or* $(s_2 \xrightarrow[mem,mem']{lab} s_2' \wedge s_1 = s_1')$.

The definition defines an interleaving semantics for two processes, more precisely for their sequential programs P_1 and P_2. The transitions of the parallel composition are the local transitions of P_1 and P_2. In contrast to the open semantics for one process only, the processes now have to agree on the memory before and after (mem, mem') a transition. However, we allow only one processes to make a step at a time, i.e., only one process can modify memory and its local state, leaving the respective other local state unchanged.

A generalisation to larger numbers of components is straightforward. We just have to extend the state tuple $(mem, s_1, s_2, \ldots, s_n)$ by additional local states up to a an n as required. The interleaving is then again achieved by allowing only one of the n local states to be modified at a time.

Due to the open semantics for processes, we are thus able to give a *compositional* semantics for parallel composition. This is key to our transformation which operates on single processes and which is presented in Section 3. However, before we come to the transformation, we still need to fix the semantics of the memory models that we want to consider in this thesis. So far, we only have the parameterized semantics, where the characteristics of a memory model are provided as the parameter.

2.3 Sequential Consistency

As already mentioned, sequential consistency is our intuition of concurrency. Lamport was the first to define it [Lam79]. In a sequentially consistent memory model, all processes share a global view of the memory. The effects of reads or writes appear immediately and are visible to all processes at the same time. The sequential programs of all concurrent processes are interleaved. Thus, all possible outcomes of a concurrent program execution can be explained by an interleaving of its sequential programs.

In the following, we define the parameter to our parameterized semantics from the previous Section, which will generate sequentially consistent behavior.

Definition 7. *The memory model SC consists of*

$$\begin{aligned}
type_{SC} &\mathrel{\widehat{=}} 2^{Var} \\
init_{SC} &\mathrel{\widehat{=}} sb = \varnothing \\
write_{SC}(x, n) &\mathrel{\widehat{=}} mem' = mem[x \mapsto n] \\
write_{SC}(x, r) &\mathrel{\widehat{=}} mem' = mem[x \mapsto reg(r)] \\
read_{SC}(x, r) &\mathrel{\widehat{=}} reg' = reg[r \mapsto mem(x)] \\
fence_{SC} &\mathrel{\widehat{=}} true \\
flush_{SC} &\mathrel{\widehat{=}} false
\end{aligned}$$

We define the type of the store buffer $type_{SC}$ to be a set of variables. However, it is actually irrelevant for SC as it is not needed in order to define sequentially consistent behavior. Therefore, it will be always empty. The initialization predicate $init_{SC}$ ensures emptiness of store buffers.

Writes appear immediately in SC. Thus, $write_{SC}$ modifies the memory directly by updating the the memory location of shared variable x to a new value. The new value can be either a constant n or a value held by a register r, which is $reg(r)$. A $read_{SC}(x, r)$ takes the value of a shared variable x from the memory mem and updates the valuation of register r by updating the register function reg.

The $fence_{SC}$ predicate is defined to be $true$. Thus, the semantics is equivalent to a skip step. A fence has blocking semantics for memory models making use of their

store buffers. It blocks until the store buffer is empty. Usually, it is used in order to avoid reordering of instructions, which can be caused by the delay due to store buffers. However, in SC, store buffers are not used and the only reason why we have store buffers here is, because the generalized semantics definition (see Section 2.2) is also used for store buffer based memory models.

In the semantics definition, the $flush_{SC}$ predicate is an alternative case for all operations. Its purpose is to model the non-deterministic flushes of weak memory models. However, since SC does not use store buffers and the content of store buffers is always empty, we can define $flush_{SC}$ to be $false$. This way, we eliminate the alternative case in the disjunctions, which define the operation semantics in Definition 4. By eliminating the alternative, the semantics preserve the sequential program order in all executions.

2.4 Total Store Order

Total Store Order (TSO) is the memory model of most x86-based multicore processors manufactured by AMD and Intel. Out of the weak memory models, it can be considered as one of the stronger memory models. In contrast to SC, TSO provides store buffers for each processor core. The store buffer is used as a temporary storage of written values to a memory location. The written values and the location they are written to are store in a First-In-First-Out (FIFO) manner. The entries are flushed to memory at an indefinite later point in time, dictated by the contention on the memory.

Store buffers work similar to a cache. They can be used by reads in order to obtain a value faster than by actually accessing the memory. A read can therefore observe a previously written value before it is flushed to the memory. In contrast to caches, which adhere to cache coherence protocols, e.g., [PP84], only the core owning the store buffer can access its previously written values before they are flushed to the memory. Cache coherence protocols usually track ownership (by cores) of cache entries and invalidate entries in other caches, whenever an owner writes to a value shared by several caches. In order to achieve this, caches coordinate with each other according to the cache coherence protocol by snooping on the bus and tracking ownership. However, store buffers, in particular those of x86 processors, do not coordinate with each other, because they are physically closer to the processor core than caches. Thus, they have to be faster than caches.

TSO allow for two effects to occur, which are not possible under SC: first, reordering of reads with earlier writes and second, a read can obtain a previously written value before it is flushed to memory, also known as *early-read*. These effects

allow for inconsistencies among multiple cores. The following two litmus tests can be used to determine, whether these effects can occur on a processors.

$Initially : x = 0 \wedge y = 0$

Process 1	Process 2
$1 : write(x, 1);$	$1 : write(y, 1);$
$2 : read(y, r1);$	$2 : read(x, r2);$
$3 :$	$3 :$

result: $r1 = 0 \wedge r2 = 0$

Figure 2.1: Litmus test for reordering of writes with later reads

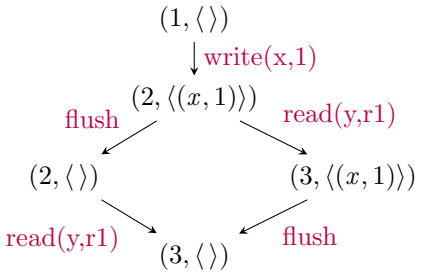

Figure 2.2: States of Process 1 of Fig. 2.1

Figure 2.1 shows a program identical to the one, we used in the introduction (see p. 5). It detects reordering of reads with earlier writes, if the result (both observed values become 0) below the program occurs. In SC, under any interleaving of the given statements, only one of the processes can observe a value 0. If a process observes value 0, then it has already processed its write (first statement) and the other process has not yet written (because we observed 0). Thus, the process that has yet to start can no longer observe value 0.

In contrast, under TSO, the write statement is not just one atomic event, but two: First, the write is enqueued into the store buffer. Second, it is flushed to the memory. In the meantime, further program statements can be processed. Figure 2.2 illustrates the states and transitions of Process 1 with respect to its program location and store buffer entries. A node consists of program location and store buffer entries. The transitions are labeled (purple) with the corresponding program statements or a flush label. After the write is processed, there is a non-deterministic choice between going on with the next program statement (the read) or flushing the entry in the store buffer. If the choice is to flush the current buffer entry (left branch), then the observable behavior is equivalent to an SC execution, in which the write appears before the consecutive read. Other processes cannot observe the non-atomicity of the write in this case and the executing Process 1 has performed both steps consecutively, which makes the write appear as if it was atomic. However, if the choice is to read the value of y (right branch), then the two statements are effectively reordered. The read obtains its value before the written value becomes visible to all other processes, i.e., the flush updates the memory with the new value.

The effect of an early-read is not an actual reordering of program statements. In fact, the litmus test in Figure 2.3 shows a program, for which both processes cannot

$Initially : x = 0 \wedge y = 0$

Process 1	Process 2
$1 : write(x, 1);$	$1 : write(y, 1);$
$2 : read(x, r1);$	$2 : read(y, r3);$
$3 : read(y, r2);$	$3 : read(x, r4);$
$4 :$	$4 :$

result: $r1 = r3 = 1 \wedge r2 = r4 = 0$

Figure 2.3: Litmus test for early-reads

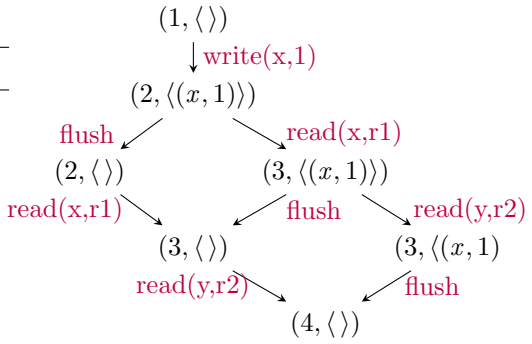

Figure 2.4: States of Process 1 of Fig. 2.3

agree on a global order of statements. It is similar to the previous program, but it is extended with a read statement between the write to variable x and the read of variable y. The new read reads the previously written variable x. A result, in which $r1 = r3 = 1 \wedge r2 = r4 = 0$ holds, reveals the effect of early-reads. The register variables $r1$ and $r3$ being 1 means that both processes have observed their own previous write. The registers $r2$ and $r4$ being 0 means that both processes have not seen the write of the respective other process. Thus, both processes observe their own write first, before the write of the respective other process.

Obviously, this outcome cannot be explained by simple interleaving of program statements as it is the case in SC. An interleaving is always a total order on the program statements of sequential programs of each process. In this example, there seem to be two orders. Again, Figure 2.4 illustrates the corresponding states of process 1 from Figure 2.3 and we will use it to explain the effects. After the write to variable x has been enqueued to the store buffer, TSO again can non-deterministically choose between flushing the entry or to continue by processing program statements. Again, if the flush transition is chosen, then the observable behavior will look equivalent to an SC run. However, if the processor continues with the next program statement (right branch), which is a read of variable x, then the read first checks the store buffer for any entries. Since in this case, the store buffer contains an entry for x, the read fetches the value from the store buffer instead of taking it from the memory, which has not been updated yet. So, process 1 observes its own written value. However, now being at program location 3, there is again the choice between flushing or processing of program statements. Please note, that the write to variable x is still pending in the store buffer and thus not visible to other processes. From this point, the example continues as in the first litmus test, i.e., the read of variable y is reordered with the pending write to x. Thus, if the process 2 has not yet flushed its write to variable y, then Process 1 can observe $y = 0$. Both processes can delay

their writes to the very end of the execution. Thus, both processes can observe a value 0 for the variable that they do not write themselves and both always observe their own write.

In order to define the memory model parameter for the semantics given in Definition 4, we have to consider two types of reads: an early-read and a normal read. We use predicates $readM$ (read from memory) and $readL$ (early-read) to distinguish between both variants of the read.

Definition 8. *The memory model TSO consists of*

$$\begin{aligned}
type_{TSO} &\widehat{=} (Var \times \mathbb{Z})^* \\
init_{TSO} &\widehat{=} sb = \langle\,\rangle \\
write_{TSO}(x,n) &\widehat{=} sb' = sb \frown \langle (x,n) \rangle \\
write_{TSO}(x,r) &\widehat{=} sb' = sb \frown \langle (x, reg(r)) \rangle \\
read_{TSO}(x,r) &\widehat{=} (x \notin sb \wedge readM_{TSO}(x,r)) \\
&\quad \vee (x \in sb \wedge readL_{TSO}(x,r)) \\
fence_{TSO} &\widehat{=} sb = \langle\,\rangle \\
\mathit{flush}_{TSO} &\widehat{=} \exists (x,n) \in (Var \times \mathbb{Z}) : sb = \langle (x,n) \rangle \frown sb' \\
&\quad \wedge mem' = mem[x \mapsto n]
\end{aligned}$$

In contrast to SC, the type of the store buffer in TSO is important. In principal the store buffer in TSO is a FIFO queue. Thus, we model the $type_{TSO}$ as a sequence of tuples, where each tuple $(Var \times \mathbb{Z})$ consists of a variable and an integer value. The initialization predicate $init_{TSO}$ states that the store buffer is initially the empty sequence. Again, we have two write predicates, $write_{TSO}(x,n)$ for writing a constant value n and $write_{TSO}(x,r)$ for writing the value of a register r, which is taken from $reg(r)$. Since a write in TSO does not write to the memory directly, both write predicates append the new entry, (x,n) or $(x, reg(r))$, at the end of the store buffer sb and thus create the new store buffer sequence sb'. The decision whether a $read_{TSO}(x,r)$ takes the read value from the store buffer or from the memory depends on the store buffer content. If there is an entry for the requested variable x in the store buffer (denoted by $x \in sb$), then it is taken from store buffer by $readL_{TSO}(x,r)$ and from the memory by $readM_{TSO}(x,r)$, otherwise. Both predicates are defined as follows:

$$\begin{aligned}
readM_{TSO}(x,r) &\widehat{=} reg' = reg[r \mapsto mem(x)] \\
readL_{TSO}(x,r) &\widehat{=} reg' = reg[r \mapsto lst_{TSO}(x, sb)] \\
lst_{TSO}(x, sb) &= n \text{ iff } \exists\, sb_{pre}, sb_{suf} \in (Var \times \mathbb{Z})^* : \\
&\qquad sb = sb_{pre} \frown \langle (x,n) \frown sb_{suf} \wedge x \notin sb_{suf}
\end{aligned}$$

where $readM_{TSO}(x,r)$ updates register entry r in the function reg with value from memory $mem(x)$ and $readL_{TSO}(x,r)$ with the last value $lst_{TSO}(x, sb)$ from the

store buffer. Please note that the semantics given by $readM_{TSO}(x, r)$ are equivalent to $read_{SC}(x, r)$. We define $lst_{TSO}(x, sb)$ via subsequences of sb, which divide sb into a prefix sb_{pre}, the entry (x, n) and a suffix sb_{suf}. The suffix sb_{suf} must not contain an entry for x. Thus, the entry (x, n) is the last or latest entry. The $fence_{TSO}$ predicate from the definition blocks until the store buffer becomes empty (denoted by $sb = \langle\rangle$). In Definition 4, the semantics of a fence $[\![fence]\!]_{MM}$ is stated to be a flush or a fence. Requiring sb to be the empty sequence for a fence, makes the disjunction a mutual exclusive disjunction, because a flush can only be performed, if there is content in the store buffer. Thus, if the store buffer is not empty, a series of flushes has to be performed before the fence becomes enabled. The $flush_{TSO}$ is again defined via subsequences of sb. Here, we divide sb into its first entry (x, n) and rest of the sequence sb', which is also the new store buffer value after the operation. The first entry (x, n) is then used to update the memory location of variable x to the value n.

2.5 Partial Store Order

Partial Store Order (PSO) is a memory model that is provided by the SPARC V8 processors [SPA92]. Since SPARC processors are rather a niche of multicore processors, PSO is less common than TSO or the more relaxed memory models ARM [ARM13] or Power [IBM15]. However, PSO is weaker than TSO and therefore can be seen as a stepping stone towards weaker memory models, which include the effects of PSO among other additional effects. Please note, that PSO allows all of the behavior that can be observed on TSO.

As the name Partial Store Order suggests, PSO, in contrast to TSO, does not preserve the order of writes. In particular, writes to different locations can be reordered with each other. However, the order of writes to the same location is preserved. There are several possible causes for the behavior, all of which can be summarized as performance optimizations by increasing throughput.

Figure 2.5 shows a litmus test that detects reordering of consecutive writes. Process 1, first, writes to shared variable x and then to y. Process 2 reads both shared variables in the opposite order, first y then x. So, in any interleaving under SC the order of writes to x and y would be preserved. Thus, one can assume that if the value of y is observed to be 1, then the value of x also has to be 1 since it must have happened before the write to y. Thus, the result $r1 = 1 \wedge r2 = 0$ is not possible under SC. It is also impossible under TSO since the TSO store buffers flush the entries in FIFO order, which is the program order.

However, under PSO the result is possible and we use Figure 2.6 in order to describe the behavior. It shows again the states of process 1 as a combination of

$Initially : x = 0 \wedge y = 0$

Process 1	Process 2
$1 : write(x, 1);$	$1 : read(y, r1);$
$2 : write(y, 1);$	$2 : read(x, r2);$
$3 :$	$3 :$

result: $r1 = 1 \wedge r2 = 0$

Figure 2.5: Litmus test for reordering of writes with other writes

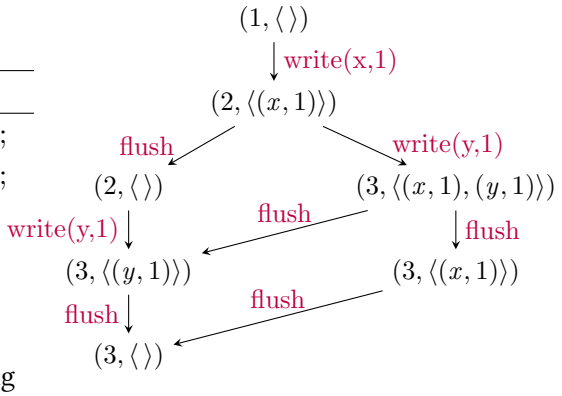

Figure 2.6: States of Process 1 of Fig. 2.5

program locations and store buffer entries. In the most left branch, the writes and flushes are alternating. Since each written value is flushed immediately after being added to the store buffer, the execution branch is equivalent to an SC execution of the two writes. If both writes add their entries to the store buffer before an entry is flushed, we get to location $(3, \langle (x, 1), (y, 1) \rangle)$. It is here, where the difference between PSO and TSO becomes visible. In TSO, all store buffer entries are flushed in FIFO order. Thus, there would be only one flush transition to $(3, \langle (y, 1) \rangle)$. PSO preserves only the order of writes to each memory location, but allows writes to different memory locations to be reordered. Thus, we get as many flush transitions as we have different memory locations in our store buffer. Therefore, we get another flush transition to $(3, \langle (x, 1) \rangle)$ in our example, which represents an execution where the older write entry to x is still pending, but the later write to y is already flushed to the memory.

In the following, we define the memory model PSO, which can be used as a parameter to Definition 4.

Definition 9. *The memory model PSO consists of*

$$
\begin{aligned}
type_{PSO} &\mathrel{\widehat{=}} (Var \to \mathbb{Z}^*) \\
init_{PSO} &\mathrel{\widehat{=}} \forall\, v \in Var : sb(v) = \langle\,\rangle \\
write_{PSO}(x, n) &\mathrel{\widehat{=}} sb' = sb[x \mapsto sb(x) \frown \langle n \rangle] \\
write_{PSO}(x, r) &\mathrel{\widehat{=}} sb' = sb[x \mapsto sb(x) \frown \langle reg(r) \rangle] \\
read_{PSO}(x, r) &\mathrel{\widehat{=}} (sb(x) = \langle\,\rangle \wedge readM_{PSO}(x, r)) \\
&\quad \vee (sb(x) \neq \varnothing \wedge readL_{PSO}(x, r)) \\
fence_{PSO} &\mathrel{\widehat{=}} \forall\, v \in Var : sb(v) = \langle\,\rangle \\
flush_{PSO} &\mathrel{\widehat{=}} \exists\, x \in Var, n \in \mathbb{Z} : sb(x) = \langle n \rangle \frown sb'(x) \\
&\quad \wedge mem' = mem[x \mapsto n]
\end{aligned}
$$

The type $type_{PSO}$ is a function mapping variables V to sequences of integers $\mathbb{Z}$. We use each sequence to model a FIFO queue per shared variable. Thus, sb is no longer a simple sequence as in TSO, but a function mapping to sequences of entries. The initialization predicate $init_{PSO}$ ensures emptiness of each such sequence. The write predicate $write_{PSO}(x, n)$ (resp. $write_{PSO}(x, r)$) takes a constant (resp. register value $reg(r)$) and appends it to the queue $sb(x)$. The updated function sb' then contains the same entries as sb for all entries except for x, for which it is mapped to $sb(x) \frown \langle n \rangle$ (resp. $sb(x) \frown \langle reg(r) \rangle$). The predicate $read_{PSO}$ has to distinguish between an early-read and a normal read from memory. The distinction is made over the store buffer contents in $sb(x)$. If the sequence $sb(x)$ is empty, the requested value for variable x is taken from the memory by $readM_{PSO}(x, r)$ and otherwise from the store buffer by $readL_{PSO}(x, r)$. The former case is again (as in TSO) equivalent to $read_{SC})(x, r)$. For the latter case, we need an additional predicate in order to determine the latest entry for a given shared variable x. Both cases and the additional predicate are given below. In contrast to the read from memory, $readL_{PSO}(x, r)$ takes the last entry from the store buffer sequence $sb(x)$, represented by $lst_{PSO}(x, b)$. The latter is n, iff we can divide the sequence $sb(x)$ into a possibly empty sequence sb_{pre} and n.

$$
\begin{aligned}
readM_{PSO} &\mathrel{\widehat{=}} reg' = reg[r \mapsto mem(x)] \\
readL_{PSO} &\mathrel{\widehat{=}} reg' = reg[r \mapsto lst_{PSO}(sb(x))] \\
lst_{PSO}(x, sb) &= n \text{ iff } \exists\, n \in \mathbb{Z}, sb_{pre} \in \mathbb{Z}^* : sb(x) = sb_{pre} \frown \langle n \rangle
\end{aligned}
$$

The PSO fence $fence_{PSO}$ blocks until the store buffers becomes empty. For PSO, this means that the sequence $sb(v)$ for each shared variable v is the empty sequence. In contrast to TSO, PSO has two kinds of fences: (1) A full fence, which blocks until the buffer becomes empty. (2) A *store-store* fence that does not block, but preserves the order of earlier writes with later writes. Thus, it restricts the flush order in a similar way as TSO.

We could extend our definition of PSO in order to formalize the second fence, but we refrain from doing this, as it would complicate the definition. One way of adding such a fence could be to put a mark entry into the store buffer sequences for each shared variable. This mark, probably indexed in order to distinguish different fences, would then have to be considered in our fence semantics. In the fence semantics, all entries before such a mark would then have to be flushed before the mark can be removed and later entries can be approached. A probably more elegant formalization of a store-store fence would require the store buffer to be a sequence of the currently

used store buffer type $type_{PSO}$. Whenever a store-store fence occurs, a new element (of type $type_{PSO}$) would be added to this sequence. All operations would only be allowed to operate on the last entry of the sequence. Flushes would have to empty this store buffer function sequence in FIFO order, thereby preserving the order imposed by the store-store fence among writes. At the same time each store buffer function would allow writes to be reordered with each other, if they are contained in the same store buffer function. However, as the explanation above suggests, adding different kinds of fences complicates the definition as a trade-off for being able to deal with different types of fences. Therefore, we consider only the blocking fence semantics.

The $flush_{PSO}$ defines the flush in PSO. It takes the first entry in the store buffer sequence $sb(x)$ and updates the memory by assigning it $mem[x \mapsto n]$, where n is the oldest value for x in the store buffer or, in other words, the first element of the sequnece $sb(x)$. Please not that the existential quantifier goes over $x \in Var$ and thus allows flushing of an entry of each variable, for which $sb(x)$ is not empty. This in particular creates as many outgoing transitions from the current state as there are different shared variable entries in the store buffer.

2.6 Relaxed Memory Order

Relaxed Memory Order (RMO) weakens the semantics of PSO even further by allowing all reads and writes to be reordered with each other, if they are addressing different memory locations. Generally speaking, RMO preserves only program order where there is data dependency between instructions, i.e., a later instruction uses the output of a previous instruction as its input. There are two popular processor architectures providing semantics that are very similar to RMO. The first is ARM [ARM13], which is often used in mobile devices and the second is Power [IBM15], which is often used in servers. In this thesis, we do not handle RMO as an execution with RMO semantics preserves barely anything from the program order. Furthermore, the semantics of RMO depend on the number of concurrent processes and thus cannot be stated in a process local way as as this section should reveal. Thus, we leave RMO open for future extension. However, in the following, we give an overview of the RMO semantics for sake of completeness.

In addition to the possible reordering of instructions, RMO also allows different processes to observe the writes of other processes at different points in time. In TSO and PSO, writes can be observed early by the writer and become visible to all other processes at the same point in time, i.e., when the flush updates the memory. In RMO, the latter is not guaranteed. Thus, a write can become visible to one process, but at the same time may be not visible to another process, yet. As a consequence, we can

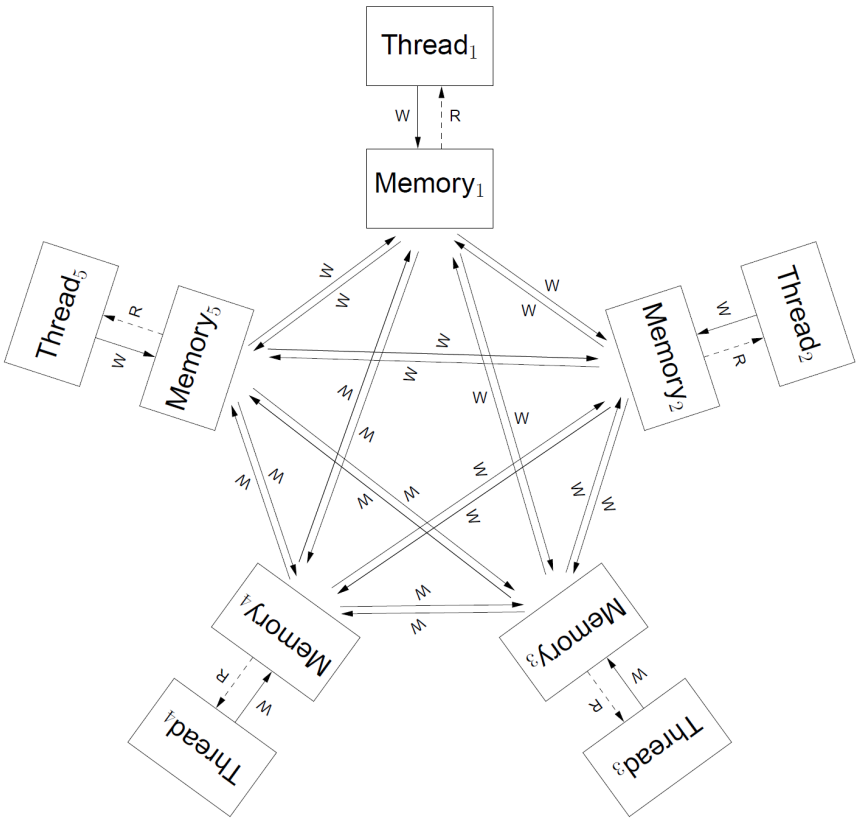

Figure 2.7: Illustration of relaxed architectures like RMO, Power and ARM [MSS12]

think of all processes to have an own view of the memory as Maranget et al. [MSS12] illustrate it. Figure 2.7 illustrates the RMO architecture, in which all threads (or processes) have their own memory. The different memories are interconnected with each other. Each memory forwards writes to other memories in separate updates rather than one atomic broadcast. This results in differently delayed updates. As each process only reads from its own memory, it can be outdated without being noticed by the process.

As an example for the weak semantics of RMO, we give another litmus test in Figure 2.8, which we took from [SSA+11]. It shows four processes, two of which write to different shared variables x and y independently. The other two processes read from x and y independently and in different orders, process 3 reads x before y and process 4 reads y before x. Process 1 and 2 can write x and y in any order, but one would expect that whatever the order may be, it is the same among all other processes. In the example, this means that if process 3 and 4, both, observe value 1 by their first reads ($r1 = 1 = r3$), then at least one of them must observe

value 1 with their second read ($r2 = 1 \wedge r4 = 1$). However, this is not always the case under RMO and the litmus detects such kind of reordering. The result can be $r1 = 1 \wedge r2 = 0 \wedge r3 = 1 \wedge r4 = 0$, in which case process 3 observed the write by process 1, but not the write by process 2. Process 4 observed the write by process 2, but not the write by process 1.

$Initially : x = 0 \wedge y = 0$

Process 1	Process 2	Process 3	Process 4
$1 : write(x, 1);$	$1 : write(y, 1);$	$1 : read(x, r1);$	$1 : read(y, r3);$
$2 :$	$2 :$	$2 : read(y, r2);$	$2 : read(x, r4);$
		$3 :$	$3 :$

result: $r1 = 1 \wedge r2 = 0 \wedge r3 = 1 \wedge r4 = 0$

Figure 2.8: Litmus test: Independent Reads of Independent Writes (IRIW)

There are two possible effects that make this result possible. The first one is a reordering of the reads of process 3 or process 4. In this case, the result becomes possible by interleaving the partially reordered programs. The second possible effect can be different delays between the processes. Thus, a new value becomes visible to one process before it becomes visible to the other. Different delays can occur for various reasons, e.g, by allowing processor cores to observe values from the store buffer of some processor cores and not allowing it for others. Asymmetric cache hierarchies or complex interconnections can also be a reason for different delays [MSS12].

From a software verification point of view, it means that the state of a concurrent program must consist of n memories for n processes. Furthermore, each memory must keep track of what writes are yet to be propagated to other memories and in particular to which memories. So, for each write, there are $(n-1)!$ permutations of propagation orders to other memories (considering that we propagate a write to each memory separately and the process propagating will always be the first in this sequence). Each order can be non-deterministically chosen. The combinatorial problem worsens, if we add more writes. With k writes to different locations, we get $(k * (n-1))!$ orders. The problem here is that the writes are not ordered at all and thus can be propagated in any order. Please note, the above number does not even consider other processes being interleaved with this propagation order. Under SC, a program that writes 10 times to 10 different shared variables would have one single execution trace with 10 events. This single trace would have to be interleaved with other concurrent processes. Under RMO, we have to consider the

number of concurrently running processes. So, for one other concurrently running process ($n = 2$), there would be $10!$ ($= 3.628$ Million) traces, which would have to be interleaved with the other process. For two processes, the number is approximately $2.4 * 10^{18}$ and hence, too big to explore the state space of such a program by most model checkers. This is also the reason, why we do not consider RMO in this thesis and refrain from giving formal definition for RMO in this section. We leave it for future work.

2.7 Related Work

Weak memory models have been studied extensively since the early 90's. Early research focused mainly on formalizing the semantics of weak memory models [AH93, PD95, AG96, CKRW99] and experimenting with their observable effects.

Most formalization can be categorized into being either axiomatic or operational. An axiomatic memory model is defined by a set of axioms which capsule the relationship of instructions (or its effect) and particularly their preserved order in executions. An axiomatic model does not need to define state internals, e.g., the store buffer, but can abstractly define the relationship between instructions. Axiomatic models are mostly used for comparison of different memory models and their relaxations [HKV98, AMSS10, SSO$^+$10, Owe10, MMS$^+$12, DTDW13]. These are often used in order to reason about small programs (litmus tests) formally. In contrast, operational memory models define the small-step behavior of an architecture and thus, allow for exploration and simulation of its behavior step-by-step. Therefore, operational memory models [PD95, YGL05, BP09, TMW13] are better suited for automated experiments with the effects of weak memory models, e.g., model checking. Our formalizations of TSO and PSO in this chapter are axiomatic, but also define the small-step semantics of the respective architecture. In that sense, they are similar to the formalization by Owens et al. [SSO$^+$10] of the x86-TSO memory model. Our definitions do not have a hardware lock as part of the architecture and also define only most important instructions like read, writes, fences. We preferred to have minimal memory model definitions for our later soundness proofs of our reduction approach in the next chapter. Otherwise, we would have to consider many more cases, which can also be expressed with the basic instruction set that our memory models provide.

This chapter introduced the most important four memory models, SC, TSO, PSO and RMO. A lot of research has been conducted on finding relationships among them and other memory models [AG96, AM06, BAM06, AMSS10, AMSS12, Alg12]. Nowadays, it is widely known that SC behavior is a subset of TSO, TSO a subset of PSO and PSO a subset of RMO. The previous sections should have helped in under-

standing this relationship as these memory models result from stepwise relaxations of orders preserved in SC. However, this behavioral inclusion does not hold for the Power and ARM processors [AG96], which are similar to RMO in terms of their relaxation, but vary in some intricate details from it, e.g., the semantics of fences.

The above memory models can be considered generalizations of currently common multicore processor architectures. For instance, store buffers are not bounded, but also other features of the highly complex multicore architectures like instruction prefetching and speculation are avoided in those models. More accurate memory models have been and are still being developed [AFI+08, SSO+10, SSA+11, MMS+12]. However, due to many details considered in their formalizations these memory models do not scale well for automated software verification, but are rather intended for determination of possible behavior under their influence as well as for their comparison with each other. The scaling problem of fine grained models is visible in the work by Abe et al. [AM14]. Their memory models include the instructions prefetching mechanism and they use their model for model checking of concurrent data structures like we do in this thesis. Their experiments with the Dekker algorithm [Dij68] under SC require magnitudes more memory and time than our approach does under TSO or PSO in a similar setup (see Chapter 6.1). We take the latter as an indicator for abstraction being necessary when dealing with weak memory models. Otherwise, these models have only limited practical use.

Recently, the C11 memory model gained popularity because of its formal foundation [ISO11b, ISO11a] and its acquire/release semantics. The latter is a generalized way to describe what parts of the program order are preserved during execution. In this sense, the C11 memory model hides different semantics available by various multicore processors and provides a unified semantics definition to program developers, similar to an interface that hides its implementations. Underneath, the compiler is responsible to ensure the C11 semantics on each respective architecture, be it an x86, Power or ARM processor. In contrast to the Java memory model [MPA05], the C11 memory model does not attempt to establish SC behavior on weak architectures. The latter remains a program developers concern, who needs to correctly place synchronization primitives or reads and writes with acquire/release semantics (instead of regular reads and writes, which do not impose any ordering constraints).

3 Reduction from Weak Semantics to Sequential Consistency

In the previous chapter, we formally defined the semantics of Sequential Consistency (SC), Total Store Order (TSO) and Partial Store Order (PSO). SC is a memory model that is widely assumed by software verification tools such as Spin, Blast, NuSMV and many more. The memory models TSO and PSO are provided by actual multicore processors such as x86 or SPARC. Their semantics are inherently weaker than SC by allowing more behavior, which can be represented by reordering of program instructions. The weaker semantics can cause unexpected results to occur during program execution, if it is a concurrent program. Thus, concurrent program verification approaches must consider weak memory models. Otherwise, such an approach may deem a program correct, although it produces erroneous behavior when executed on actual multicore processors.

In the presented approach, we overcome this gap by providing a reduction technique in the form of transformation that we first published in [WT15] and further extended in [TW16]. Instead of adapting all the tools available for concurrent software verification in order to support weak memory models, we provide the transformation. It can be applied to programs before verification and produces a new program that in addition to the original program behavior also contains behavior that is due to an underlying weak memory model such as TSO or PSO.

The reduction is a two step approach: First, a symbolic execution explores all symbolic store buffer states of each sequential program, of which the concurrent program consists. Each explored state is represented by a combination of a program location and a symbolic store buffer content. In combination, the explored states and the program transitions make up a graph, which we call the *store buffer graph*. The store buffer graph incorporates all the behavior (and effects) due to a weak

memory model. In the second step, the store buffer graph is used to generate a new program that is behaviorally equivalent to the original program under TSO or PSO. The new program assumes SC execution semantics and thus can be the input to most verification tools. However, as weak memory models are inherently non-deterministic due to flush transitions, the newly generated program is also non-deterministic. Hence, we cannot simply generate a new C program incorporating the behavior of weak memory models, because C programs are deterministic. Instead, we are restricted to programming languages that allow for non-deterministic programs, e.g., Promela, CSP. Fortunately, most verification tools do support non-deterministic programs or specifications in some way.

Figure 3.1 shows the overall approach as a three step approach, where the last step is the actual verification of correctness. While the former two steps are fully automated, the last step heavily depends on the choice of the verification tool and correctness property. We have chosen the theorem prover KIV [EPS+14] and the model checker Spin [Hol03] as verification tools and hence, generate SC programs for both tools. KIV is an interactive theorem prover and thus, needs manual guidance in many cases. It provides a proof visualization in terms of proof trees. Proof trees help as an abstract view to cases of a proof that are yet to be proved or that were proved already. Especially for big proofs, proof trees can help not to get lost in the details of a proof. Spin is a fully automated model checker. Once the correctness property (a safety or an LTL property) is specified, Spin performs a fully automated explicit state space exploration.

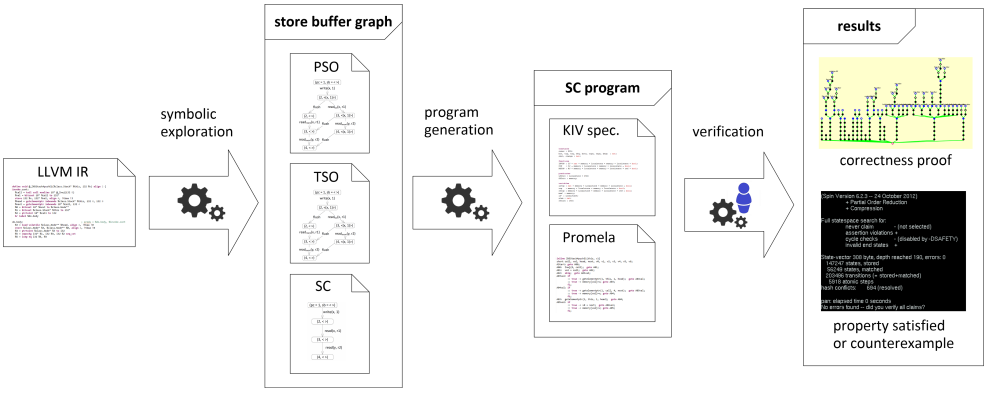

Figure 3.1: Three steps towards verification of concurrent programs under weak memory models

In the remainder of this chapter, we first elaborate more on the symbolic exploration. We introduce and formally define store buffer graphs and explain their

derivation from the LLVM IR programs. Later, we define the transformation from the store buffer graph to a new program. This includes a general transformation, which we define formally, and the transformation to Promela and KIV specifications. This is followed by a section on the soundness of the presented transformation. We prove that the original program executed with TSO (resp. PSO) semantics and a newly generated program resulting from our transformation and executed under SC semantics are behaviorally equivalent. The latter is shown via the existence of a bisimulation relation between both transition systems. The chapter concludes with a discussion of related work and the strengths and weaknesses of the presented approach.

3.1 Symbolic Execution with Weak Memory Semantics

As previously mentioned, our approach aims at generating new programs, which assume SC semantics, but incorporate the effects of a weak memory model in the program. In other words, we transform every sequential program P into a new program P' such that the labeled transitions systems $lts_{TSO}(P)$ (resp. $lts_{PSO}(P)$) are bisimilar to $lts_{SC}(P')$. See Section 2.2 for the definition of a labeled transition system lts_{MM} for memory model MM.

For this purpose, we first have to determine the effects of a weak memory model on the behavior of the program. In order to do so, we perform a symbolic execution of each sequential program (out of which a concurrent program is composed). The symbolic execution tracks - besides the operations being executed and the program locations reached - store buffer contents *only*, and only in a symbolic form. The symbolic form stores variable names together with either values of $\mathbb{Z}$ (in case a constant was used in the $write$), or register *names* (in case a register was used). A symbolic store buffer content for TSO might thus for instance look like this: $\langle (x, 3), (y, r_1), (x, r_2), (z, 5) \rangle$. The symbolic execution generates a symbolic reachability graph, which we call *store buffer graph*, and which we use as a basis for the newly generated program P'. The symbolic exploration can be performed quickly, because it ignores actual register values and uses register names instead. More importantly, each sequential program is explored separately in contrast to a concurrent composition.

However, even though we ignore register valuations and thus abstract away most of the state, the symbolic state space can become easily infinitely larger under TSO or PSO, although it might be finite under SC. The reason for it is that, in contrast to actual multicore processors, memory models generally do not limit the size of a store buffer. Thus, a simple program loop, which does nothing else but to write some value to memory in each loop iteration, creates an infinitely large state space.

$1 : write(x, 1);$
$2 : goto\ 1;$

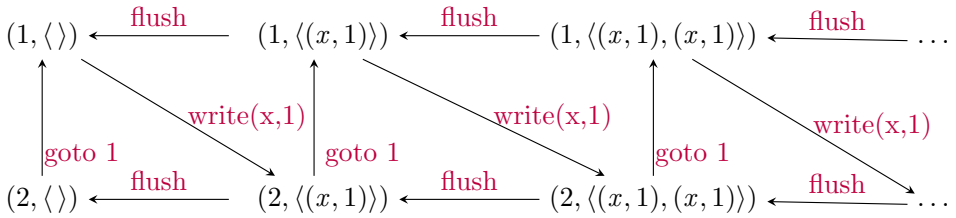

Figure 3.2: Simple program and its infinite state space under weak memory models

The non-determinism of TSO and PSO can be best described by always having the choice between doing a program step or to flush content from the store buffer (if there is any). The store buffer can choose to accumulate writes in every iteration of such a writing loop as it has no maximum number of entries. As a consequence, an exploration of such a program would find infinitely many store buffer states with a varying number of entries. Furthermore, a naive exploration of such a program's state space would never terminate. Figure 3.2 visualizes the mentioned example. It is a simple two line program. Below the program, you can see its state space illustrated as a store buffer graph. Starting from the top left state (program location 1 and an empty store buffer $\langle\,\rangle$), the written entries can be accumulated by the store buffer infinitely often. With this problem in hindsight, we state the first and most important restriction of the presented approach.

Assumption 1: We assume programs to not have unfenced writing loops, i.e., loops with a write instruction also contain at least one fence or some other synchronization instruction, which forces the store buffer to be emptied.

The restriction is necessary in order to guarantee finiteness of the store buffer graph. Store buffer graphs must be finite in our approach, because they are used to generate the new programs. The restriction is sufficient, because a fence forces the store buffer to be emptied at least once during each loop iteration. Since a loop iteration is finite and store buffer entries are symbolic values, a constant or a register name (representing the value of the register), a loop iteration can create only a finite number of store buffer states. Thus, the store buffer graph of a program, which has no unfenced writing loops is finite.

The restriction may sound like it would be limiting the applicability of our approach. It certainly does, but on a far smaller scale as it may look at first sight.

Most if not all concurrent programs have to use fences or synchronization instructions (e.g., compare-and-swap) in order to be correct under weak memory models. Synchronization is part of pretty much every concurrent algorithm, as these have to ensure at some point in time that changes made by one process are visible to other processes. This is achieved by using fences or instructions with fence-like semantics. Please note that we focus on concurrent data structure implementations throughout this theses. These can have conflicting operations and therefor must synchronize with each other. Programs, which do not have to synchronize with each other perform independent tasks and cannot conflict, i.e., they have no data races. For programs, which have no data races, weak memory models do not have to be taken into account as the observable behavior is always equivalent to what can be observed under SC semantics [AH90, AG96].

Programs can vary in many ways and some of them may even make no sense, e.g., programs writing to previously undefined variables. Since, we do not want to deal with all possible special cases, but want to have clean definitions of our reduction approach, we have to make on additional assumption:

Assumption 2: We assume that all sequential programs are in *SSA-form* (static single assignment [CFR+91]), meaning that all the registers are (statically) assigned to only once. We also assume that registers are never used before defined.

Assumption 2 is not an actual restriction of our approach, because all programs can be converted to SSA-form. Both properties, the SSA-form and no use of registers before they are defined are guaranteed by modern compilers, e.g., the LLVM-framework[1] which we use in our approach.

In the remainder of this section, we define store buffer graphs and explain how they can be derived from a program.

3.1.1 Store Buffer Graph

In principle, the *store-buffer graph* of a sequential program is its symbolic reachability graph. Nodes in this graph represent possible combinations of program locations and symbolic store buffer contents. The value of an entry in a symbolic store buffer can be either a constant $n \in \mathbb{Z}$ or a register $r \in Reg$. We use $SVal \mathrel{\widehat{=}} \mathbb{Z} \cup Reg$ as the set of possible symbolic values. Consequently, the type $stype_{MM}$ of a symbolic store buffers differs slightly from $type_{MM}$. We use $stype_{TSO} \mathrel{\widehat{=}} (Var \times SVal)^*$ and $stype_{PSO} \mathrel{\widehat{=}} (Var \rightarrow SVal^*)$, respectively as the type of a symbolic store buffer. Edges in the graph have labels, however, only symbolic ones. We refer to these symbolic labels as the *name* of an operation ($Names$). For memory model

[1] http://www.llvm.org

specific operations, this is simply the name used for the operation (e.g., *flush*) with the exception that *read* is split into *readM* (read from memory) and *readL* (read from store buffer) according to the semantics. For the other operations it is the (unevaluated) boolean condition *bexpr* or its negation (in case of *if* and *while*), or simply *goto*.

Definition 10. *A* store-buffer (or sb-)graph $G = (V, E, v_0)$ *of a memory model* MM *consists of a set of nodes* $V \subseteq \mathcal{L} \times stype_{MM}$*, edges* $E \subseteq V \times Names \times V$ *and initial node* $v_0 \in V$ *with* $v_0 = (\ell_0, sb_0)$ *and* $sb_0 \models init_{MM}$*.*

Please note that we use the predicate $init_{MM}$ in various ways for convenience. We state $sb \models init_{MM}$ to say that $init_{MM}$ holds true for sb and $s \models init_{MM}$ in order to say that $\exists\, pc \in \mathcal{L}, sb \in type_{MM}, reg \in Reg : s = (pc, sb, reg) \wedge sb \models init_{MM}$.

Before we give a formal definition of the store buffer graph, we have to consider a pattern of program instructions, which we call *write-def-chain* or short (WDC). A WDC can occur in loops. It is a sequence, in which a $write(x, r)$, the source of a WDC, is followed by a redefinition of the written value r, but there is no fence or other synchronization in between these two instructions. Thus, there is no guarantee that the flush corresponding to the write occurs before r is redefined.

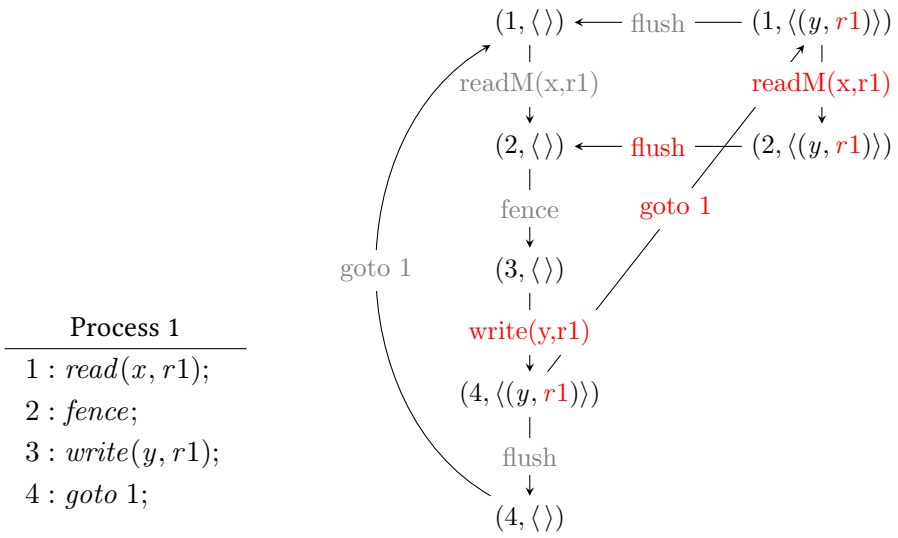

Figure 3.3: An example program (left) containing a write-def-chain and its store buffer graph (right). The important edges and the register variable $r1$ are marked red.

This pattern is important, because in the later transformation, the main idea is to transform writes into skips and flushes into writes. Basically, a flush of a symbolic

value r is replaced by a write of the register r. However, this is only possible, if the register value is still the same as it was, when it was added to the store buffer as a symbolic entry. In a WDC, this is not necessarily the case. See Figure 3.3 for an example WDC. The program is a simple loop, reading x in line 1 and writing the read value into y in line 3. The read value is stored in the register variable $r1$. Our assumptions 1 and 2 hold, i.e., it is a fenced loop (line 2), it conforms to SSA-form and register variables are defined before they are used. The write in line 3 creates a symbolic entry $r1$ in the store buffer. Following the program order, we get to state $(1, \langle(y, r1\rangle)$ via $goto$ 1 and then to state $(2, \langle(y, r1\rangle)$ via $readM(x, r1)$. When the read occurs, it redefines the value of $r1$ to a new value, which is taken from the memory. Thus, although we do not track register valuations in the store buffer graph, we know for sure that the value of $r1$ has changed at this state. Hence, we have lost the value of $r1$, which was added to the store buffer by the write in line 3. However, the flush (marked red) of the pair $(y, r1)$ has yet to occur and flushing the value of $r1$ would be incorrect.

In order to get rid of the WDC and thus to overcome this problem, we have to create a copy (an auxiliary variable) of the written value and use it instead of the original variable until it is flushed. Please note that this is very similar to what a store buffer does, since a store buffer entry is just a copy of a register value at the time of executing a write instruction. Thus, if a write is the source of a WDC, then we know that we need an auxiliary variable as a copy of the written value. On the other hand, we know for all other writes that their behavior can be mimicked by reordering instructions.

Figure 3.4 shows the store buffer graph after getting rid of the WDC in Figure 3.3. The idea here is to replace the edge name of writes that are the source of a WDC by something that we can later transform into equivalent SC behavior. Since our general transformation idea is to convert write transitions into skip transitions and the corresponding flush transitions into the actual write transitions under SC, we think it is elegant to make an exception for the write to skip conversion. Both, a write under TSO or PSO and a skip under any memory model are inherently local transitions, i.e., not visible by other processes. Thus, we can modify such a transition as long as we ensure that the observable behavior (from the point of view of other processes) does not change. In the example in 3.4, the edge $write(y, r1)$ is replaced by $r1' := r1 \wedge write(y, r1')$ where $r1'$ is a fresh variable, i.e., not used and particularly not defined by any other instruction. The assignment $r1' := r1$ ensures that we have a copy of the written value and since it is a fresh variable, it is not redefined before a fence ensures that it was flushed. By also replacing $r1$ by $r1'$ in $write(y, r1)$, we also ensure that the symbolic store buffer contains $r1'$ instead of

$r1$. Thus, when a later flush occurs, it will use $r1'$ and not the potentially redefined register variable $r1$.

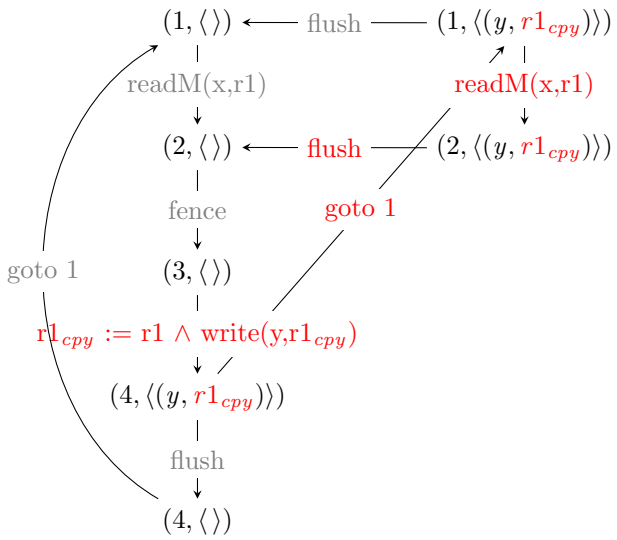

Figure 3.4: Store buffer graph after fixing the write-def-chain (WDC) from Figure 3.3

Generally, we have to find all WDCs of a program and apply this modification to all writes that are the source of a WDC. The WDCs of a program can be easily determined by a reachability analysis in the control flow graph. Starting at a write instruction, we can simply follow the control flow in order to determine, whether we can reach an instruction which redefines the written value before a fence is reached. If it is the case, then the write is the source of a WDC. If no such path can be found, then the write is not involved in WDC and we can leave it as it is. Furthermore, because we assume programs to conform to SSA-form (each variable is assigned to/defined only once), we only have to consider write instructions in loops.

The store-buffer graph for a program P is constructed via symbolic execution, executing program operations step by step without constructing the concrete states of registers. In particular, the symbolic execution follows the operations as given by the memory model MM including flush operations and tracks the reached symbolic states. If a new state is discovered, a node representing the state and an edge leading to it is added to the graph. The exploration is performed in a *depth-first* manner, but could also be achieved with other search algorithms, e.g., *breadth-first* search. The choice of the search algorithm is not important as we only take one sequential program at a time into account. Thus, it is a significantly smaller problem than the exploration of a concurrent composition of sequential programs, which we have to

deal with in the later verification.

Definition 11. *Let P be a sequential program and let $wdc(P) \subseteq Ops(P)$ be the set of write operations that are the source of a write-def-chain. The* sb-graph *of P wrt. a memory model MM, $sg_{MM}(P)$, is inductively defined as follows:*

1. *$v_0 := (pc, sb)$ with $sb \models init_{MM} \wedge pc = \ell_0$,*
2. *if $(pc, sb) \in V$, we add a node (pc', sb') and an edge $(pc, sb) \xrightarrow{name} (pc', sb')$ if $\exists\, op \in Ops(P)$ such that*
   - *$pc, pc' \models op$,*
   - *$sb, sb' \models sym_{MM}(op)$*
   - *$name = name(op)$.*

where

$$
\begin{aligned}
sym_{TSO}(op) &= (sb' = sb \frown \langle (x, r) \rangle) && \text{if } op = write_{TSO}(x, r) \wedge op \notin wdc(P)\\
sym_{TSO}(op) &= (sb' = sb \frown \langle (x, r_{cpy}) \rangle) && \text{if } op = write_{TSO}(x, r) \wedge op \in wdc(P)\\
sym_{TSO}(op) &= op && \text{else,}\\
sym_{PSO}(op) &= (sb'(x) = sb(x) \frown \langle r \rangle) && \text{if } op = write_{PSO}(x, r) \wedge op \notin wdc(P)\\
sym_{PSO}(op) &= (sb'(x) = sb(x) \frown \langle r_{cpy} \rangle) && \text{if } op = write_{PSO}(x, r) \wedge op \in wdc(P)\\
sym_{PSO}(op) &= op && \text{else}
\end{aligned}
$$

and

$$
\begin{aligned}
name(op) &= r_{cpy} := r \wedge write(x, r_{cpy}) && \text{if } op \in wdc(P) \wedge op = write_{MM}(x, r)\\
name(op) &= readM(x, r) && \text{if } op = read_{MM}(x, r) \wedge x \notin sb\\
name(op) &= readL(x, r) && \text{if } op = read_{MM}(x, r) \wedge x \in sb\\
name(op) &= op && \text{else}
\end{aligned}
$$

The definition of the sb-graph creates an edge for each possible step (corresponding to an operation op) of the program P that leads from one state (pc, sb) to another state (pc', sb'). The function sym_{MM} maps each write operation predicate op to a different predicate, which has to be satisfied by the symbolic store buffer, i.e., if a write writes a register value, then the symbolic store buffer has to contain a symbolic register entry instead of the actual value. The function works as an identity function for all other operations. The writes that are the source of a WDC ($op \in wdc(P)$) get a special treatment, i.e., the symbolic store buffer sb' must contain a fresh variable r_{cpy} instead of the original r. This results in the above three cases

per function sym_{MM}. In addition to this, the name of such write edges is modified to $r_{cpy} := r \wedge write(x, r_{cpy})$. The assignment is what will remain in the later generated SC program as we need it, in order to remember the written value. In general, writes will be represented as skip steps in the later program, but not writes involved in a WDC. This is also the reason, why we rename the sources of a WDC. Please note that none of the above modifications are necessary for writes of constants, which have an identical representation in a symbolic store buffer. Furthermore, the store buffer graph allows us to determine immediately, whether a read will take its value from the store buffer or from the memory. It can be determined just by looking at the contents of the symbolic store buffer. Read edges are renamed from $read(x, r)$ to $readM(x, r)$, if ($x \notin sb$) and $readL(x, r)$, otherwise. All other edges are named according to their operation, e.g., *flush* for flush operations, $write(x, r)$ for write operations, etc.

3.1.2 Store Buffer Graphs Properties

Please note that a store buffer graph does not differ from a control flow graph in case of SC semantics. For TSO and PSO semantics, we can guarantee finiteness of the store buffer graph because of **Assumption 1** and **Assumption 2**.

Proposition. *Let P be a finite sequential program in which every loop is fenced or write-free. Then $sg_{MM}(P)$ is finite for every $MM \in \{TSO, PSO, SC\}$.*

Proof: The proposition is trivially true for SC, because $sg_{SC}(P)$ is simply the control flow graph of P and P is finite.
Assume $sg_{TSO}(P)$ (resp. $sg_{PSO}(P)$) is infinite. Since a state is a pair of program location and symbolic store buffer and because program locations are finite, there must be a sequence of steps generating infinitely many different symbolic store buffer states. Since P is finite, such a sequence can only be a loop. According to the proposition, each loop is write-free or it contains a fence. A write-free loop does not add content to the store buffer and it is a finite sequence. Thus, the number of reachable states within a write-free loop is finite.
A loop with a fence is a finite sequence of steps, of which one step can only proceed if the store buffer is empty. Because the sequence of steps before the buffer is emptied is finite in each iteration, the only way to generate an infinite set of reachable states is to write infinitely many different entries to the store buffer. However, store buffer entries are symbolic, i.e., a constant or the name of the register variable, from which the written value is taken. Both are statically fixed with the program. Thus, the set of reachable symbolic states in a fenced loop is finite. Together, this contradicts the above assumption of $sg_{TSO}(P)$ (resp. $sg_{PSO}(P)$) being infinite. □

One key property of our later transformation is that registers remain unmodified until the corresponding symbolic store buffer entries are flushed. If this is the case, we can later replace flush transitions by write transitions, i.e., if a flush transition flushes an entry (x, r), then we replace it with a $write(x, r)$ transition. For this replacement to be sound, programs must not have WDCs as they allow for redefinition of register values before the value is flushed.

Proposition. *Let P be the program of a process in SSA form without WDCs. Let $s = (\ell, reg, sb)$ be a state of $[\![P]\!]_{MM}$ with $MM \in \{TSO, PSO\}$ such that sb contains an entry $n \in \mathbb{N}$ for a variable $x \in Var$. If this value has been put into the store buffer by an operation $write(x, r), r \in Reg$, then $reg(r) = n$.*

Proof: Since P has no WDCs and by the definition of write-def-chains: after $write(x, r)$ there is no further operation defining r before the next fence operation. Otherwise P would have a WDC and this would contradict the assumption. A fence, however, needs an empty store buffer in order to execute. □

In principle, we want not only the above property to be true for a program P, but also a similar property to hold for its store buffer graph $sg_{MM}(P)$. For sources of WDCs, there must be an auxiliary variable which holds the copied value. Creation of such variables is part of the store buffer graph construction. In our soundness proofs in Section 3.3.1, we show bisimulation equivalence of the original and the transformed programs. This proof implies that flushes in the original program and writes in the transformed program do use the same values. Therefore, we refrain from restating the above property lifted for store buffer graphs.

Now that we have a formal definition of the store buffer graph which represents all program behavior of one sequential program, we can go on with the transformation towards an SC program.

3.2 Transformation to a new SC Program

In the previous section, we described the construction of the store buffer graph for a sequential program. A store buffer graph represents *all* of the behavior that a sequential program can examine under a weak memory model such as TSO (resp. PSO). Please note that we did not differentiate whether any of the behavior is observable by other processes as it depends on the sequential programs executed by the other processes. There is no simplification or modification involved. Thus, the behavior represented by the store buffer graph will race on reads and writes of other processes, iff the original program would also conflict on the same reads and writes of another process under the respective memory model.

The general transformation idea is to generate a new SC program that mimics the effect of the semantics represented by each edge in the store buffer graph. Because the effect of each edge in the store buffer graph is preserved in the new program, the resulting behavior is equivalent to the original one with respect to conflicts with other processes and, more generally, with respect to the observable behavior of the program. Thus, a parallel composition out of the original sequential programs under weak memory model semantics and a parallel composition of transformed programs under SC will have equivalent behavior. The generation of the SC program now proceeds by defining the operations of the new program (instead of program text). Essentially, every edge in the store buffer graph gives rise to one new operation, where the nodes of the graph act as new location labels. The store buffer graph is thus the control flow graph of the new program.

Please note that the new program will most likely be non-deterministic, because flushes of writes appear non-deterministically in weak memory models. Thus, the only way a program can remain deterministic after deriving the store buffer graph and transforming it into a new SC program is, if it has no writes. Hence, the set of possible target programming languages of our transformation is restricted to the set of languages, which support non-determinism. However, our aim is to verify concurrent data structure implementations, not running them on SC processors with the effects of weak memory models. Most, if not all, concurrency verification tools and in particular their input languages do support non-determinism. This renders the above restriction to be a rather theoretical one and not a practical one.

Before we give the formal definition, we provide an example of what a transformed program looks like. In Figure 3.5, the control flow of the new program after transformation is shown. Its corresponding store buffer graph was presented in Figure 3.4 on page 38.

As you can see, the nodes in the graph are identical to those in the store buffer graph. However, here, the node labels act only as program location labels as opposed to the combination of program location and a symbolic store buffer state in a store buffer graph. The new program simply does not have a store buffer as part of its states. We keep the location labels as they are for now, since the label clearly identifies the state of the store buffer graph that is represented by each program location of the new program. We can also use numerical location labels or any other variant of labels and, in fact, do so in our implementation of the approach as we explain later in Chapter 4. The edge labels changed more or less compared to the corresponding store buffer graph. In particular, the reads that get their value from the memory are represented by SC read operations. The semantics and thus the effect of a read in this case are equivalent. The fence operation became a *skip* as there is no store buffer to be emptied under SC. The write starting at program location $(3, \langle\,\rangle)$ was the source

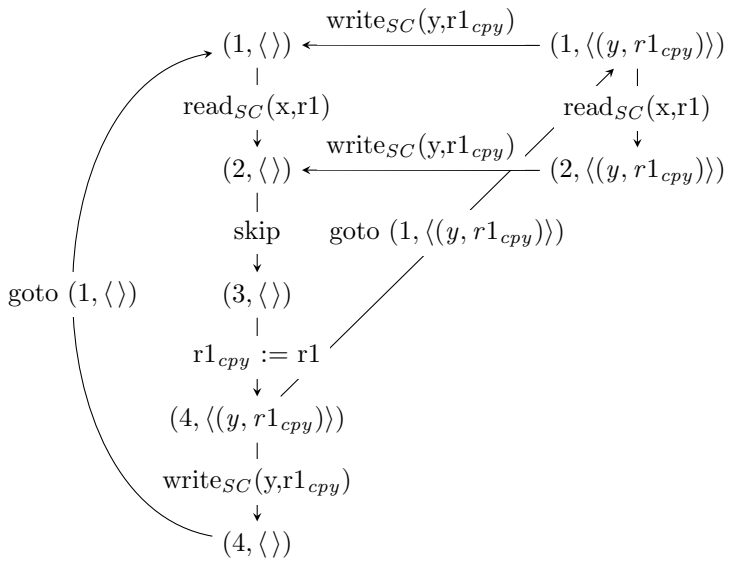

Figure 3.5: Control flow graph of the new program after transforming the store buffer graph from Figure 3.4. The nodes represent the new program locations now. Edges have been replaced with the respective SC operation mimicking their effect under SC.

of a WDC. Hence, a copy of the written value ($r1_{cpy} := r1$) is required. Otherwise, the edge would have also become a *skip* operation. The flush edges from the store buffer graph became writes in the new program. From the store buffer graph, it is obvious to see which store buffer entries can be flushed at any given state, no matter whether the underlying memory model is TSO or PSO. Hence, we know exactly which symbolic store buffer entry is flushed to the memory by an edge. In the new program, these edges become write operations with the variable and value taken from the symbolic store buffer. In the example, it is variable y with value $r1_{cpy}$ for all such edges. If the write in the example would not be the source of WDC, then the original register variable $r1$ instead of $r1_{cpy}$ would be used. Finally, the target locations of the *goto* edges are also modified, in order to distinguish between the two possible states of the store buffer that they represent, one with an empty store buffer and one with content.

In the following, all operations of the new program are also given formally in terms of predicates. The formal operations simply encode the graph in Figure 3.5. In case you are missing the *skip* and *goto* operations in the encoding, these are operations that do not alter the state except for modifying the program location pc towards pc'. The *skip* is encoded in nOp_2. The *goto* operations are encoded as

nOp_5 and the second case of the disjunction in nOp_4. The assignment $r1_{cpy} := r1$ is replaced by its predicate logic encoding $r1'_{cpy} = r1$ in nOp_3. The new sequential program P' has a set of operations $Ops(P') = \{nOp_i \mid i \in \{1, \dots 7\}\}$. The only new variant of an operation predicate, is the non-deterministic choice in nOp_4 and nOp_6, which is just a disjunction over two regular predicates encoding an SC operation. Each such predicate can also be defined as two separate predicates, and our general definition of the transformation defines two predicates in such cases. In the example, we made the non-determinism explicit for presentation purpose.

$$
\begin{array}{rlll}
nOp_1 \mathrel{\widehat{=}} & pc = (1, \langle\,\rangle) & \wedge read_{SC}(x, r1) & \wedge pc' = (2, \langle\,\rangle) \\
nOp_2 \mathrel{\widehat{=}} & pc = (2, \langle\,\rangle) & & \wedge pc' = (3, \langle\,\rangle) \\
nOp_3 \mathrel{\widehat{=}} & pc = (3, \langle\,\rangle) & \wedge r1'_{cpy} = r1 & \wedge pc' = (4, \langle (y, r1_{cpy})\rangle) \\
nOp_4 \mathrel{\widehat{=}} & pc = (4, \langle (y, r1_{cpy})\rangle) & \wedge write_{SC}(y, r1_{cpy}) & \wedge pc' = (4, \langle\,\rangle) \\
\vee & pc = (4, \langle (y, r1_{cpy})\rangle) & & \wedge pc' = (1, \langle (y, r1_{cpy})\rangle) \\
nOp_5 \mathrel{\widehat{=}} & pc = (4, \langle\,\rangle) & & \wedge pc' = (1, \langle\,\rangle) \\
nOp_6 \mathrel{\widehat{=}} & pc = (1, \langle (y, r1_{cpy})\rangle) & \wedge read_{SC}(x, r1) & \wedge pc' = (2, \langle (y, r1_{cpy})\rangle) \\
\vee & pc = (1, \langle (y, r1_{cpy})\rangle) & \wedge write_{SC}(y, r1_{cpy}) & \wedge pc' = (1, \langle\,\rangle) \\
nOp_7 \mathrel{\widehat{=}} & pc = (2, \langle (y, r1_{cpy})\rangle) & \wedge write_{SC}(y, r1_{cpy}) & \wedge pc' = (2, \langle\,\rangle)
\end{array}
$$

Note that P' is not defined syntactically here, even though Figure 3.5 can be viewed as a graphical syntax of the new program. Instead, the operations of P' are provided in terms of the SC semantics (see Section 2.1). We leave the syntax of P' undefined and also do not need it here for the general approach, because it is the semantics that defines the behavior of P'. In principle, the syntax can be anything that fits to the given semantics. It has to be provided by an implementation of our approach. The choice of the syntax dictates the tools that can be used for verification of the transformed program. Our implementation transforms programs into of Promela and KIV models as we explain in Chapter 4 in detail.

Given the above example and the idea for the transformation, we are now ready to present the definition of the transformation towards SC programs.

Definition 12. *Let $G = (V, E, v_0)$ be a store buffer graph of a sequential program P on a memory model MM. The sequential SC program P' of G, $prog(G)$, is given by the new initial location $\ell_0 := v_0$ and the set of operations $Ops(P')$ defined as follows: for every edge $(\ell, sb) \xrightarrow{name} (\ell', sb')$ we construct an operation*

$$(pc = (\ell, sb) \wedge op \wedge pc' = (\ell', sb'))$$

with

$$op = \begin{cases} true & \text{if } name \in \{fence, write(x, rn), skip, goto\ \ell'\} \\ r'_{cpy} = r & \text{if } name = (r_{cpy} := r \wedge write(x, r_{cpy})) \\ write_{SC}(x, rn) & \text{if } name = flush \wedge flushed_{MM}(x, rn, sb, sb') \\ read_{SC}(x, r) & \text{if } name = readM(x, r) \\ r' = rn & \text{if } name = readL(x, r) \wedge rn = lst_{MM}(x, sb) \\ name & else \end{cases}$$

where $rn \in SVal$*,*

$$flushed_{TSO}(x, rn, sb, sb') \mathrel{\widehat{=}} (sb = \langle (x, rn) \rangle \frown sb')$$
$$flushed_{PSO}(x, rn, sb, sb') \mathrel{\widehat{=}} (sb(x) = \langle rn \rangle \frown sb'(x)$$

and

$$lst_{TSO}(x, sb) = rn \text{ iff } \exists\, sb_{pre}, sb_{suf} \in stype_{TSO} : \\ \wedge\, sb = sb_{pre} \frown \langle (x, rn) \rangle \frown sb_{suf} \wedge x \notin sb_{suf}$$
$$lst_{PSO}(x, sb) = rn \text{ iff } \exists\, seq_{pre} \in SVal^* : sb(x) = seq_{pre} \frown \langle rn \rangle$$

The definition provides the operations of the new program P' in terms of predicates, each consisting of a program location before (pc) and after (pc') and some operational predicate op. The new program locations take the store buffer states as possible set values, i.e., $pc, pc' \in V$, where V is the set of nodes in store buffer graph G. Technically, this is not necessary, but it provides a clear link between each program location in the new program and its corresponding state in the store buffer graph. The latter can be useful during a correctness proof, when the store buffer graph acts as an overview of the program's behavior. The above definition maps each edge to an operational predicate op, which is just a $true$ value for $fence$, non-WDC $write$, $skip$ and $goto$ edges. All of these edges modify the program location only. Sources of a WDC are mapped to an assignment, which creates a copy of the written value. Flush edges are mapped to SC writes. Since flush edge labels do not contain the information, which entry is flushed from the store buffer, we need an additional predicate $flushed_{MM}(x, rn, sb, sb')$. Its definition differs for TSO and PSO, but in both cases it holds true, iff rn is the symbolic value flushed by the given edge and x is the shared variable it was written to. Reads are distinguished already in the store buffer graph by the source of the read value. A read from memory ($readM(x, r)$) is semantically equivalent to an SC read and thus is mapped to $read_{SC}(x, r)$. The reads ($readL(x, r)$) that take the read value from the store buffer are mapped to an assignment $r' = rn$, where rn is the latest entry in the store buffer for the requested

shared variable x. Again, we need two versions for the definition of the latest such entry $lst_{MM}(x, sb)$ for both memory models, TSO and PSO. In TSO, the store buffer is a FIFO queue and the latest entry for variable x may be not the last element. So, we divide the buffer into three parts: a prefix, the latest entry for x, and a suffix. The suffix must not contain an entry for x. Otherwise, it would be the latest entry. In PSO, the latest entry is easy to determine, since the store buffer has a FIFO queue for each shared variable. Thus, the latest entry for x is simply the last element in the sequence $sb(x)$. For all operations that are not explicitly mentioned in the definition, the mapping is an identity mapping.

The full transformation from a sequential program on memory model MM into its SC form (weak to SC) is finally defined as

$$w2sc(P, MM) \mathrel{\widehat{=}} prog(sg_{MM}(P))$$

We have now fully defined the reduction for programs executed under weak memory model into programs that reproduce the weak behavior under SC semantics. The transformed programs are defined in terms of their semantics. A syntactical representation of the program can be fixed for any programming or verification language that assumes SC semantics and allows for modeling non-deterministic choice of program statements. In the following section, we show soundness of the reduction and elaborate on strengths and weaknesses of the approach.

3.3 Reduction is Sound and Compositional

So far, we have shown how the store buffer graph is constructed and how we can generate new programs that reproduce the behavior of the original program under SC. What remains to be shown is that the new programs are indeed behaviorally equivalent to the original program. In other words, we have to show that the reduction is sound.

Please remember that purely sequential programs (not part of a parallel composition) do not need to consider weak memory models and thus are guaranteed to produce behavior that is also observable under SC. Our reduction applies to each sequential program separately, but our focus lies on concurrent data structures, i.e., programs that are a parallel composition of sequential programs. Even if we can show that our reduction is sound for a single sequential program, this is not necessarily true for parallel compositions. The Litmus tests in Chapter 2 are good examples for it. When a sequential program is executed on TSO or PSO, it produces SC results only and the observable behavior does not differ from SC. However, when combined in a parallel composition, the internals of TSO and PSO become observable and the non-SC results reveal these internals.

This leaves us with an additional proof obligation. We have to show that the transformed sequential programs also reproduce the behavior of the original program, when executed in a concurrent setting. In other words, we have to show compositionality of our reduction, i.e., we can replace an original program by its transformed counterpart in a parallel composition without changing its behavior. Essentially, this means that we need an equivalence relation that is also a congruence w.r.t. the possible operations in our semantics. In other words, equivalent programs must be indistinguishable from each other, even in a parallel composition with an arbitrary other program.

Before we can prove that our reduction is sound and compositional, we need to provide a notion of what we think is equivalent behavior. In particular, we are interested in observable behavior. In a concurrent setting, processes communicate with each other by modifying the memory and reading from it. Each process has a local state that is not visible to other processes. Thus, the observable behavior of a process is its access to the memory. All other operations that are not accessing memory are hidden from other processes. Hence, whatever our reduction does to a program, the resulting transformed program must be identical to the original program in terms of its memory accesses.

Ultimately, we will be interested in comparing the weak memory model semantics of one program with the SC semantics of another. We do this by comparing their transition systems. Our notion of behavioral equivalence of transition systems is based on bisimulation equivalence [Mil80], in particular strong bisimulation. Two transition systems are bisimulation equivalent, if we can provide a bisimulation relation for them. A bisimulation relation relates the states of both transition systems. The related states are branching equivalent, i.e., the outgoing transitions and branches from these states are equivalent and thus, also the remainder of the observable behavior beginning at these states. For each step of one of the transition systems, the other must be able to do the same step and vice versa. States that are reached by such a step must again be related by the bisimulation relation.

In the following, we define and prove our notion of bisimulation equivalence formally. In our definition, we take arbitrary interference with other processes into account by not making any assumption on the memory. We call it an open semantics. In a second proof, we use the results (the bisimulation relation) from the first proof, in order to show that bisimilar transition systems can safely replace each other in a parallel composition. A discussion about related work and about remaining problems of the reduction concludes this chapter.

3.3.1 Local Bisimulation Equivalence

Our definition of bisimulation compares transition systems with respect to their *labels* on transitions as well as their local *states*. In Chapter 2.2, we gave a general definition for the labelled transition system $lts_{MM}(P)$ of a program , where the memory model MM is a parameter to the definition. However, we did not introduce the labels explicitly in order to avoid confusion. Now that the idea behind the reduction was given in the previous two sections, the labels should be straightforward. We label transitions in terms of their effect to the memory, to the register and to the program location. The store buffer is ignored in the labels. In spirit, we use SC operations in order to label transitions of all labeled transition systems, including those constructed for weak memory semantics.

Definition 13. *Let $lts_{MM}(P)$ be the labelled transition system of a sequential program P on a memory model $MM \in \{TSO, PSO\}$. The labels $label_{MM}$ for $lts_{MM}(P)$ are defined as follows:*

$$
\begin{array}{lll}
label_{MM}(op) \mathrel{\widehat{=}} & skip & \text{if } op \in \{fence_{MM}, write(x,n)\} \\
label_{MM}(op) \mathrel{\widehat{=}} & skip & \text{if } op = write(x,r) \text{ and } op \notin wdc(P) \\
label_{MM}(op) \mathrel{\widehat{=}} & r_{cpy} := reg(r) & \text{if } op = write(x,r) \text{ and } op \in wdc(P) \\
label_{MM}(op) \mathrel{\widehat{=}} & write(x,n) & \text{if } op = flush_{MM} \text{ and } flushed^{conc}_{MM}(x,n,sb,sb') \\
label_{MM}(op) \mathrel{\widehat{=}} & read(x,r) & \text{if } op = readM_{MM}(x,r) \\
label_{MM}(op) \mathrel{\widehat{=}} & r := n & \text{if } op = readL_{MM}(x,r) \text{ and } n = lst^{conc}_{MM}(x,sb) \\
label_{MM}(op) \mathrel{\widehat{=}} & op & \text{else}
\end{array}
$$

Here, we must differentiate between normal writes and those which are a WDC source. Normal writes become skip steps in the new program. Writes that are sources of a WDC, are transformed to local assignments to a new register variable copy r_{cpy}. The chosen labels must reflect that, because our later bisimulation proof requires, among other properties, identical labels in both transition systems. Please note that we use the predicate $flushed^{conc}_{MM}(x,n,sb,sb')$, and the function $lst^{conc}_{MM}(x,sb)$ in a similar way to $flushed_{MM}(x,n,sb,sb')$ (resp. $lst_{MM}(x,sb)$), both of which were previously defined (see Def. 12) based on symbolic store buffers in a store buffer graph. The only difference here is that we have concrete store buffers instead of symbolic ones, since we have concrete states of the labeled transition system. Thus, store buffers do not contain symbolic entries and therefor contain only integer values. Please also note that the operation $read_{MM}(x,r)$ has two different cases in TSO and PSO, for which the label function provides different labels.

Definition 14. *Let $lts_{SC}(P)$ be the labelled transition system of a sequential program P under SC. The labels $label_{MM}$ for $lts_{SC}(P)$ are defined as follows:*

$$
\begin{array}{lll}
label_{SC}(op) \mathrel{\widehat{=}} & skip & \text{if } op = fence_{SC} \\
label_{SC}(op) \mathrel{\widehat{=}} & write(x, n) & \text{if } op = write_{SC}(x, n) \\
label_{SC}(op) \mathrel{\widehat{=}} & write(x, reg(r)) & \text{if } op = write_{SC}(x, r) \\
label_{SC}(op) \mathrel{\widehat{=}} & read(x, r) & \text{if } op = read_{SC}(x, r) \\
label_{SC}(op) \mathrel{\widehat{=}} & op & \text{else}
\end{array}
$$

From the labelling, we can already see which transitions are mapped to similar labels and thus also which transitions could potentially simulate transitions from another transition system. We still need to define when we consider two transition systems to be behaviorally equivalent. Therefor, we define a local bisimulation equivalence, which is an adaption of bisimulation equivalence by Milner [Mil80]. We call it local, because it is an equivalence of single process behavior. However, since our semantics is open, we also take interference by other processes into account. Our labeled transition system definition for sequential programs (see Chapter 2.2) creates a transition for every possible combination of s, mem, op, s', mem', s.t., $(s, mem), (s', mem') \models op$. These combinations include all possible interferences with other processes.

Definition 15. *Let $T_1 = (S, \rightarrow_1, S_0)$ be an MM_1 and $T_2 = (Q, \rightarrow_2, Q_0)$ an MM_2 transition system. Transition systems T_1 and T_2 are* locally bisimilar*, $T_1 \sim_\ell T_2$, if there is a bisimulation relation $\mathcal{R} \subseteq S \times Q$ such that the following holds:*

1. *Local state equality:*
   $\forall (s, q) \in \mathcal{R}, s = (pc_1, sb_1, reg_1), q = (pc_2, sb_2, reg_2),$
   $\forall\, r \in Reg_1 \cap Reg_2$*:* $reg_1(r) = reg_2(r)$.

2. *Matching on initial states:*

   - $\forall\, s_0 \in S_0\, \exists\, q_0 \in Q_0$ *s.t.* $(s_0, q_0) \in \mathcal{R}$*, and*
   - $\forall\, q_0 \in Q_0\, \exists\, s_0 \in S_0$ *s.t.* $(s_0, q_0) \in \mathcal{R}$.

3. *Mutual simulation of steps:*

   - *if* $(s_1, q_1) \in \mathcal{R}$ *and* $s_1 \xrightarrow{lab} s_2$*, then* $\exists\, q_2$ *such that* $q_1 \xrightarrow{lab} q_2$ *and* $(s_2, q_2) \in \mathcal{R}$*,*
   - *if* $(s_1, q_1) \in \mathcal{R}$ *and* $q_1 \xrightarrow{lab} q_2$*, then* $\exists\, s_2$ *such that* $s_1 \xrightarrow{lab} s_2$ *and* $(s_2, q_2) \in \mathcal{R}$.

The definition states three properties that must hold for a bisimulation relation $\mathcal{R}$. The first one requires equality of local states, i.e., the register valuations must be equal. However, this only has to hold for register variables that exist in both transition systems ($Reg_1 \cap Reg_2$). Since, our transformation creates auxiliary variables, this

property cannot hold without constraining it to the intersection of both register sets. Local state equality does not make any assumption on the program locations or the store buffer content. This is crucial for us, since the transformed programs in our approach do have a new set of program locations. On the other hand, a transformed program (SC) does not actually have a store buffer, because we defined it to be always empty.

The second property states that the bisimulation relations must relate the initial states of both transition systems. Each initial state must have a matching one in the other transition and vice versa, s.t., both are in $\mathcal{R}$.

The most important property of a bisimulation relation is the mutual simulation of steps. If two states (s_1, q_1) are related by $\mathcal{R}$ and one makes a step by following a transition labeled by lab, then the other can simulate this step by also following a transition with the same label. The states reached (s_2, q_2) must again be related by the bisimulation relation $\mathcal{R}$.

The definition of local bisimulation equivalence is based on the fact that the semantics is open. However, we could easily extend it to closed semantics. To do so, we only have to add mem to both sets of states as a part of the state tuple. This implicitly adds a requirement for both transition systems to agree on the memory in addition to the already stated local state equality.

This completes our definition of behavioral equivalence of local transition systems and we are ready to introduce our first theorem. It states that the transformed program produced by our approach is local bisimulation equivalent to the original program under TSO (resp. PSO).

Theorem 1. *Let P be a program with fenced or write-free loops only and with no unfenced wd-chains and $MM \in \{PSO, TSO\}$ a memory model. Then*

$$lts_{MM}(P) \sim_\ell lts_{SC}(w2sc(P, MM)) \; .$$

Proof Idea: The general proof idea is to make a case distinction over all types of transitions. We only state the bisimulation relation here. For each transition type, we prove that it simulates a transition of the respective other transition system and that the states reached are again related by the bisimulation relation. The full proof can be found in the Appendix A.1.

$$\begin{aligned} \mathcal{R}_{MM} \quad := \{(s_1, s_2) \mid \quad & s_i = (pc_i, sb_i, reg_i), i = 1, 2, \\ & \wedge\, sb_2 = \varnothing \\ & \wedge\, pc_1 = \mathit{first}(pc_2) \\ & \wedge\, conc_{MM}(reg_2, second(pc_2)) = sb_1 \\ & \wedge\, \forall\, r \in Reg_1 : reg_1(r) = reg_2(r)\} \end{aligned}$$

The relation contains pairs (s_1, s_2), both consisting of tuples (pc_i, sb_i, reg_i), $i = 1, 2$. The state s_1 can have a non-empty store buffer sb_1 as it is the state of $lts_{TSO}(P)$ or $lts_{PSO}(P)$. In contrast, s_2 (an SC state) has always an empty store buffer, i.e., $sb_2 = \varnothing$. The program counter value pc_2 is a tuple and it consists of two parts: a location of the original program and a symbolic store buffer (out of the sb-graph). For a program location $pc_2 = (pc, sb)$, we use a function $\mathit{first}(pc_2) = pc$ in order access the original program location (first tuple entry). The function $second(pc) = sb$ yields the symbolic store buffer value sb (second tuple entry). The correspondence between s_1 and s_2 consists of three major properties: First, the program locations pc_1 and $\mathit{first}(pc_2)$ have to match. Second, a concretization of the symbolic store buffer $second(pc_2)$ yields a store buffer that is identical to the one in state s_1. More formally, we define a function $conc_{MM}(reg, sb)$ taking register values reg and a symbolic store buffer sb as arguments and returning a concrete store buffer for a memory model $MM \in \{PSO, TSO\}$ as

$$
\begin{aligned}
conc_{TSO}(reg, \langle\,\rangle) &= \langle\,\rangle \\
conc_{TSO}(reg, \langle (x, n)\rangle \frown sb) &= \langle (x, n)\rangle \frown conc_{TSO}(reg, sb) \\
conc_{TSO}(reg, \langle (x, r)\rangle \frown sb) &= \langle (x, reg(r))\rangle \frown conc_{TSO}(reg, sb) \\
conc\,Var(reg, \langle\,\rangle) &= \langle\,\rangle \\
conc\,Var(reg, \langle n\rangle \frown sb) &= \langle n\rangle \frown conc\,Var(reg, sb) \\
conc\,Var(reg, \langle r\rangle \frown sb) &= \langle reg(r)\rangle \frown conc\,Var(reg, sb) \\
conc_{PSO}(reg, sb)(v) &= conc\,Var(reg, sb(v))
\end{aligned}
$$

Here, $n \in \mathbb{Z}$ and $r \in Reg$. Both concretization functions are defined inductively. For TSO, the function maps the symbolic store buffer to a sequence of pairs (shared variable and value). For PSO, we need a sequence for each shared variable. A function $conc\,Var$ maps a symbolic value sequence to a concrete sequence. The function $conc_{PSO}$ is a higher order function. It maps the register valuation reg and the symbolic store buffer sb to $type_{PSO}$, which is again a function. For each variable $v \in Var$, it yields the concrete value sequence. The third part of the bisimulation relation requires register equality. In particular, we require register values reg_1 and reg_2 to be equal for each register $r \in Reg_1$. The set Reg_1 is the set of registers from the program with weak semantics. We have $Reg_1 \subseteq Reg_2$, because Reg_2 may also contain auxiliary variables that were created during transformation. Obviously, Reg_1 is also the intersection of both sets and thus, fits well to the state equality requirement from Definition 15.

Our proof shows behavioral equivalence of sequential programs under weak memory semantics and their transformed SC form with respect to the local bisimu-

lation equivalence given in Definition 15. However, as we already pointed out, that is not enough. Every sequential program produces equivalent behavior under the memory models SC, TSO, and PSO, if it does not conflict with other concurrent programs. Concurrent data structure implementations do conflict with each other and they do it intentionally. Thus, we need to show that the transformed programs also behave equivalently in a concurrent setting. We do this by proving compositionality of the transformed program in the following section.

3.3.2 Compositionality of the Approach

In the previous section, we have shown behavioral equivalence of transformed programs under SC (using our transformation approach) and the original program under TSO or PSO with respect to local bisimulation equivalence (see Def. 15). However, it is an equivalence of sequential programs and we have not derived any implications for concurrent compositions of these sequential programs. The major question is whether we can replace sequential programs by their transformed versions in a concurrent composition and still observe the same behavior. Obviously, this is the ultimate goal and one of the major benefits of our approach. The particular benefit is that we can verify a transformed program correct and know that the verification result also applies to original program under weak memory because of their behavioral equivalence. This section shall provide a formal proof as an argument for this conclusion.

Instead of a closed semantics, we decided to use an open semantics in our definition of the labeled transition system of a program (see Section 2.2). Thus, our labeled transition systems cover all possible behavior of a program in any environment with other programs instead of just one possibility or a subset of the possible behavior. The latter could be the result of fixing an initial memory state or it could be the result of additional constraints on the changes to the memory. Our semantics definition makes no assumptions about the initial state of the memory, nor about possible steps of other processes. The latter can alter the memory in an arbitrary way. Thus, implicitly, our labeled transition systems contain every possible interference with other processes. Despite other processes possibly interfering, the transformed programs are locally bisimulation equivalent to their respective original programs.

We state a compositionality theorem for transformed programs and prove it. Since we no longer have arbitrary interference with other processes, but particularly with those processes that are part of the parallel composition, we need a closed semantics. The difference between these two variants is that in a closed semantics all processes have to agree on a memory mem while the open semantics simply

considers all values of mem (see also Def. 6 in Section 2.2). For the closed semantics, we define a global bisimulation equivalence $\sim_g$ between two transition systems, which in addition to local bisimulation equivalence requires equality of shared memory mem among the transition systems of the programs that are in the parallel composition. For completeness, we define global bisimulation equivalence in the following definition:

Definition 16. *Let $T_1 = (S, \rightarrow_1, S_0)$ be an MM_1 and $T_2 = (Q, \rightarrow_2, Q_0)$ an MM_2 transition system. Transition systems T_1 and T_2 are* globally bisimilar, *$T_1 \sim_g T_2$, if there is a bisimulation relation $\mathcal{R} \subseteq S \times Q$ such that the following holds:*

1. *Global state equality:*
   $\forall (s, q) \in \mathcal{R}, s = (mem, ls_1, ls_2), q = (mem, lq_1, lq_2),$
   $ls_i = (pc_{i,1}, sb_{i,1}, reg_{i,1}), lq_i = (pc_{i,2}, sb_{i,2}, reg_{i,2}), i \in \{1, 2\},$
   $\forall\, r \in Reg_{i,1} \cap Reg_{i,2}$: $reg_{i,1}(r) = reg_{i,2}(r)$.

2. *Matching on initial states:*

   - $\forall\, s_0 \in S_0\, \exists\, q_0 \in Q_0$ *s.t.* $(s_0, q_0) \in \mathcal{R}$, *and*
   - $\forall\, q_0 \in Q_0\, \exists\, s_0 \in S_0$ *s.t.* $(s_0, q_0) \in \mathcal{R}$.

3. *Mutual simulation of steps:*

   - *if* $(s_1, q_1) \in \mathcal{R}$ *and* $s_1 \xrightarrow{lab} s_2$, *then* $\exists\, q_2$ *such that* $q_1 \xrightarrow{lab} q_2$ *and* $(s_2, q_2) \in \mathcal{R}$,
   - *if* $(s_1, q_1) \in \mathcal{R}$ *and* $q_1 \xrightarrow{lab} q_2$, *then* $\exists\, s_2$ *such that* $s_1 \xrightarrow{lab} s_2$ *and* $(s_2, q_2) \in \mathcal{R}$.

The global bisimulation equivalence requires global state equality, similarly to the local state equality in the local bisimulation. Since we are dealing with parallel compositions, the state tuples in both transition systems consist of a memory mem (the same in both transition systems) and local states ls_i, lq_i, which encode the state of each single process. Each local state of the first transition system ls_i has to have a locally equal state lq_i in the other transition system and vice versa. The matching of initial states and the mutual simulation of steps are defined identically to the local bisimulation. There are no further important differences to the local bisimulation equivalence. The definition can be extended to an arbitrary number of processes in a parallel composition by adding more local states to the state tuples in S, Q. The extension is straightforward and would be similar to an extension of our definition of the parallel composition (see Def. 6 in Section 2.2).

In the following theorem, we state our desired compositionality result: Local bisimilarity of processes implies global bisimilarity of their parallel compositions.

Theorem 2. *Let* P_1, P_1', P_2, P_2' *be sequential programs such that* $lts_{MM_1}(P_j) \sim_\ell lts_{MM_2}(P_j')$ *,* $j \in \{1,2\}$*. Then*

$$lts_{MM_1}(P_1 \mid\mid P_2) \sim_g lts_{MM_2}(P_1' \mid\mid P_2') \ .$$

Proof idea: Again, we only state the bisimulation relation and provide the full proof in the Appendix A.2. The proof proceeds by showing that from a state, in which the bisimulation relation holds, we can only reach states that are also related by the bisimulation relation. By our assumption in the theorem, the processes of the parallel compositions are locally bisimulation equivalent. Thus, we can reuse this property in our global bisimulation relation. Let $\mathcal{R}_j$, $j \in \{1,2\}$, be the relations showing local bisimilarity of $lts_{MM_1}(P_j)$ and $lts_{MM_2}(P_j')$. Out of this, we construct the following global bisimulation relation $\mathcal{R}$:

$$\mathcal{R} := \{((mem, ls_1, ls_2), (mem, lq_1, lq_2)) \mid (ls_j, lq_j) \in \mathcal{R}_j, j \in \{1,2\}\}$$

With this proof, we show that our transformation from programs running under TSO and PSO towards equivalent SC programs is sound. In particular, we have shown that the transformed programs behave equivalently to original programs not only in an isolated environment, but also in parallel compositions with other processes. It is particularly this result that enables verification of transformed programs and, if verified correct, to assume correctness of the original programs under TSO (resp. PSO) semantics. In the following section, we discuss related work as well as strength and weakness of the reduction presented in this chapter.

3.3.3 Related Work and Discussion

In the last years, several approaches were proposed in order to deal with software verification under the influence of weak memory models, ranging from theoretical results to practical techniques. In the following, we cover only a fraction of them as our main focus lies on different variants of modeling behavior of programs under weak memory. However, we start with some theoretical aspects and a property called robustness. The latter states that programs reveal SC behavior only, even though they are executed under weak memory models.

Theoretical Results

Atig et al. [ABBM10] have shown that the reachability problem for programs in a TSO or PSO environment is decidable. Their decidability result was obtained via a reduction to lossy channel machines. However, for other more relaxed memory models like RMO, which allow reads to be reordered with later reads or writes, the

problem is undecidable. The latter was shown by Atig et al. via a reduction to the post correspondence problem [Pos46], which is known to be undecidable. Their decidability results fit to our experience with our attempts of reducing program behavior under weak memory to an equivalent SC program. It is intrinsically more difficult to provide an equivalent SC program for a program with RMO semantics than for a program with the stronger TSO or PSO semantics. TSO and PSO preserve program order at least partially, i.e., reads are not reordered with later instructions. In RMO, program order is only preserved among instructions that are control or data dependent. Essentially, this leaves us with a more or less unordered set of instructions that corresponds to a sequential program. The state space of a parallel composition of seemingly unordered sets of instructions grows quickly with the number of instructions. That is because for each process, all permutations of its instruction set must be considered, except for those violating control or data dependencies. This is also one of the reasons for us to exclude RMO from our reduction technique. Although we think it is generally possible to represent symbolic states of a program under RMO in a store buffer graph (would have to be extended to represent parallel compositions), we do not think it would be practical. Even for rather small and simple programs, the store buffer graph could grow large quickly since each possible execution corresponds to a particular path in the store buffer graph.

Bouajjani et al. [BMM11] determined the complexity (PSpace) of deciding robustness of programs against TSO. A program is considered robust against a weak memory model, if its SC behavior and its observable behavior under the weak memory model are equivalent. In principle, robustness opens up an alternative for the verification of programs under weak memory models. Instead of verifying program behavior under weak memory directly, robustness enables a two step verification: The first step is to show correctness of a program executed under SC, which can be the result of reasoning about a program in its non-compiled representation. In a second step, the program is shown to be robust against a particular weak memory model. By separating these two concerns, the overall effort can potentially be less than tackling both concerns at once. It is particularly the reasoning about correctness of a low-level program that can be difficult and complex and that is avoided by this approach.

In [Der15], Derevenetc shows PSpace-completeness of deciding robustness also for the weaker memory models PSO and RMO. He also developed algorithms in order to determine, whether a program is robust against a particular weak memory model. Because of its separation of concerns, robustness can be a practical property. However, the assumption that a program executing under weak memory is correct, iff it shows SC behavior only is not necessarily true. Some algorithms like the Spinlock (used in Linux-Kernel) are considered correct, even though they allow for non-SC

behavior [GMY12]. For these highly optimized algorithms, robustness imposes unnecessary restrictions on their behavior. Thus, reasoning about correctness and the effects of weak memory semantics at the same time cannot always be avoided. In such cases, complexity is probably the biggest challenge due to the programs being low-level and weak memory models allowing instructions to be reordered in many ways. Memory models are usually given as a set of axioms that define the possible execution orders or as an operational model that generates all possible executions. In both cases, the memory model adds a whole layer of complexity to reasoning about correctness. The latter is due to unfolding of possible next steps during the reasoning process. Our reduction to an SC program avoids particularly this complexity by mimicking the memory model in its effects in the generated program. Although this comes at the cost of having a larger low-level program than originally, the reasoning about whether a particular execution or reordering is possible or not becomes obsolete. The overall behavior to be verified is an SC interleaving of the processes, each executing a transformed sequential program.

Model Checking

An approach for stateless model checking has been proposed in [AAA$^+$15]. The authors introduce chronological traces in order to represent TSO and PSO executions. A chronological trace represents a class of equivalent traces under weak memory. Based on chronological traces, the authors apply dynamic partial order reduction techniques. A program is deemed correct, if all of its chronological traces do not violate the property to be fulfilled. The authors compute the set of all chronological traces incrementally, starting with a randomly scheduled execution. New traces are the result of permuting pairs of events in a trace. The computation continues until all chronological traces are found or the correctness property is violated. The approach by [AAA$^+$15] is a pure model checking approach and for some experiments, it achieves better results than our reduction (see Chapter 6). However, it is also an under-approximating technique, e.g., the approach replaces program loops by a read and an assume statement. Thereby, the authors eliminate parts of the behavior, which comes due to weak memory semantics in order to speed up the state space exploration. Our reduction produces bisimulation equivalent SC programs without any simplification of the observable behavior. In order to apply similar techniques in our approach, we would require a weaker notion of bisimulation equivalence for soundness. It is not yet clear what this notion would be and we leave this for future work. However, we also see potential for program simplifications, which do not affect observable behavior of a program. Programs often have a write followed by several process local instructions, which do not involve a memory access. For

instance, a write followed by two process local instructions generates three possible execution paths in a store buffer graph: First, the write can be flushed immediately. Second, the write can be flushed after the first local instruction and third, it can be flushed after the second local instruction. All three possible executions generate a different path in a store buffer graph, but do not vary in their observable behavior, because local steps are hidden from observers, i.e., other processes. These executions form an equivalence class and it is sufficient to check one of them for correctness since the results will not vary with the choice of the execution. Thus, the store buffer graph can be reduced to one path instead of three in such a case. This suggestion does not change the observable behavior and thus would remain sound with respect to some weaker bisimulation equivalence than the one proposed. In [AAA+15], the observable program behavior is modified, but the modifications are justified only informally. The approach by [AAA+15] cannot be applied in a correctness proof without formalizing the trace computation and reasoning about the set of all chronological traces. The latter was avoided by the authors and it would not be practical due the many traces. Our reduction is a general approach (w.r.t. our assumptions about the programs) and thus, can help with model checking and correctness proofs, e.g., in a theorem prover.

A recent approach of checking robustness has come up in [BCDM15]. The authors propose a lazy model checking algorithm that uses a robustness-based oracle. The check is applied to an automata encoding of parallel programs. It first checks a program with pure SC semantics and reports if the correctness condition was violated (state reachability). Otherwise, the oracle is asked for a path that leads to a non-robust state, which can only exist due to store buffers. If such a path exist, the program is extended by the path and checked again for the correctness condition. The latter happens incrementally until either a counterexample was found or until no more new non-robust states and their corresponding execution paths are reported by the oracle. The path that is added to the SC program behavior simulates TSO behavior under SC. Thereby, the overall behavior is a mixture of SC and TSO behavior. In contrast to our reduction, the proposed approach creates auxiliary variables for each entry in a store buffer (address-value pairs) in order to simulate store buffer behavior under SC. Because of loop-unwinding and because the approach in [BCDM15] is not restricted to programs like our approach (see Assumption 1), the number of auxiliary variables required can grow quickly, especially in loops. Our approach requires at most one auxiliary variable per write in a loop, which we can guarantee because of our restriction to programs without unfenced writing loops. If we would drop Assumption 1, we would also require auxiliary variables for every entry in a store buffer. Please note that this decision is a trade-off between having an approach that is as general as possible and an approach that is practical,

but restricted to a certain class of programs. In addition, the loop-unwinding by the approach always sets an upper bound on the iterations of loops, which is not required in our reduction due to the restricted class of programs considered. Another major difference is that our store buffer graphs are process local and compositional while the oracle in the approach takes parallel compositions as input. Furthermore, the correctness check in [BCDM15] is performed by incrementally extending program behavior lazily with non-SC execution paths, which are returned by the oracle. From a practical point of view, this involves two state space explorations in each iteration. The first is a correctness checks of the SC behavior of the program including the so far added paths leading to non-robust states. The second exploration checks for new non-robust states that were not considered yet. Our reduction does not require iterative checking procedures, because store buffer graphs represent already all of the behavior that a sequential program can generate under TSO (resp. PSO). Thus, a parallel composition of our transformed programs incorporates already all of the possible behavior and the state space must be explored only once.

Reductions

Cohen and Schirmer proposed a reduction theorem [CS10], which aims at avoiding reasoning about TSO behavior. In particular, they suggest a programming discipline, which guarantees sequentially consistent executions only, even though the program is executed on TSO. The latter is shown via reduction from *store buffer machines* to SC computations. In their approach, they introduce auxiliary variables for each memory location in order to keep track of the state and other attributes of a location, i.e., ownership, sharing, whether it is a read-only location and process local dirty flags for all locations. The authors define access policies to memory locations with respect to their current state. Adherence to the access policy can be shown via a program invariant while reasoning. The access policies mainly try to avoid reordering of memory accesses to shared variables, but allow reordering of memory accesses if a local variable is read or written. This is sufficient, because local variables are never seen by other processes. Instruction reordering becomes visible, when concurrent processes access a shared variables. This is particularly where the proposed programming discipline requests use of volatile read and writes. For writes, an ownership of the written location is also required. Volatile reads and writes differ in their semantics from regular reads and writes by ensuring program order, e.g., by placing a fence after a write in order to flush the store buffer before the program continues or by placing a fence before a read, s.t., it cannot read from the store buffer early. Thus, the proposed discipline systematically avoids weak memory effects on shared locations, which is where they could be observed otherwise. In

other words, the programming discipline constructs robust programs against TSO by design. Hence, it can be practical for developers, who do not want to reason about correctness in the presence of weak memory semantics and who are willing to pay the price for it in terms of performance. Otherwise, there is no way to circumvent the weak memory semantics, which requires techniques like a robustness check at least or a reduction technique like the one proposed in this thesis.

A very general reduction approach was introduced by Alglave et al. [AKNT13]. They define an abstract machine that simulates weak memory models (TSO, PSO, RMO and Power) under SC and proved it to be equivalent to their framework of axiomatic memory models [AMSS10]. The abstract machine is used to construct an abstract event graph for a given concurrent program, where the abstract events are all reads and writes of a program. The edges in such an abstract event graph are dictated by program order and external communication, e.g., if one process could potentially read a value written by another process. The abstract event graph is then transformed into an SC encoding in order to determine potentially critical cycles among the events. A cycle indicates potential non-SC behavior. Where a cycle is found, the SC program encoding is instrumented, in order to simulate weak memory behavior, e.g., the delay of writes. The instrumented encoding can then be checked for correctness conditions using SC verification tools. In contrast to our reduction, Alglave et al. start their analysis with a parallel composition, which requires them to restart their transformation with any new process that is added or removed. Our reduction is process local and compositional. Thus, once a sequential program is transformed into its SC representation, it can be reused in different parallel compositions. The latter often correspond to different scenarios for different properties under test. Furthermore, the approach by Alglave et al. is limited to model checking only, because transformed parallel compositions are everything but readable and therefore must be processed automatically. Since our reduction takes each sequential program (each process) separately into account, the generated SC programs (comparably concise models) can also be used for proving correctness, e.g., in a theorem prover.

Linden and Wolper [LW10] proposed a reduction for store buffers in TSO to finite automata and exploit it as a partial order reduction technique [God96] for model checking. They use automata to represent symbolic store buffer contents and introduce conditions under which symbolic store buffers can represent each other. These conditions can be met, if programs write the same content repeatedly to the store buffer as it can be the case in an unfenced writing loop. The proposed technique certainly can reduce exploration time of a model checker. Our Assumption 1 allows us to ignore the problem of unfenced writing loops among other benefits like having at most one auxiliary variable per write in loops. If we would drop Assumption 1, then

store buffer graphs could become infinitely large in our approach. An adaption of the approach by Linden and Wolper to store buffer graphs could guarantee finiteness even for programs with unfenced writing loops. We leave such an adaption for future work. However, it is worth mentioning that the authors express their own doubt that their reduction will be helpful with verification of large programs, because the finite representation of buffers can still be very large in practice.

The approach closest to us is the one by Atig et al. [ABP11]. Similar to us, they provide a translation from a TSO program to an equivalent SC program, but assume an age bound k. The bound k stems from the observation that store buffer entries can stay for at most k steps in the store buffer until they are eventually flushed to the memory. Their approach is to model the store buffer behavior as part of the new SC program by introducing k vectors of shared variable copies as part of the local state. Hence, rather than getting rid of the complexity of store buffers, store buffers are replaced with auxiliary vectors in the SC program. The bound results in some sort of bounded verification; if the program exceeds the bound (e.g., in case of loops without fences), the bound needs to be increased and verification restarted. In our approach (auxiliary) variable copies are only used if they are indeed required, i.e., when the symbolic store buffer entry corresponding to the source of a write-def-chain can be redefined between write and flush. We have at most one new variable per register in the program. This is enough since we consider a restricted class of programs (Assumption 1) for which we can then carry out a (non-bounded) verification. In summary, our approach works for a restricted class of programs, but for this carries out a full verification, whereas Atig et al.'s technique works for all programs, however, sometimes only with an under-approximating analysis. For the class of programs with fenced-loops our approach furthermore generates fewer auxiliary variables, and may speed up verification (see experiments in Chapter 6). We thus see our approach as an excellent alternative to Atig et al.'s, in case the program falls into our category of fenced-loop programs.

Another interesting approach by Dan et al.[DMVY13] developed predicate abstraction for weak memory models. Their starting point is a program that was verified under SC. Given the predicates or the invariant that was required to verify the program correct under SC, they describe a way to extrapolate the invariant to TSO and PSO semantics. The extrapolation involves adding auxiliary variables representing store buffer entries under weak memory semantics. Similar to the previous approach, this step is bounded by a finite store buffer length. If in case of an unfenced writing loop, the store buffer size can grow infinitely, then authors impose an artificial constant bound on it. The extrapolation aims to preserve essential information about variables from the SC proof into a TSO or PSO setting by adding predicates involving the auxiliary variables. The result of the extrapolation is a new SC program

incorporating weak memory behavior. In a final step, the program together with its inferred invariant have to be verified by a model checker. In principle, the idea is similar to the previous approach as it also is a transformation, but it is applied to the program and an SC invariant instead of the program only. Furthermore, it also differs in treatment of buffer sizes, which may result in under-approximation of the possible behavior due to introduction of artificial bounds. The authors are able to verify infinite state programs, but may fail to do so due to the abstractions they introduce while extrapolating the invariant.

Limitations and Future Extensions

In the following, we elaborate on some limitations of our transformation approach that haven't been discussed in the related work above. Along the limitations, we also discuss possible future extensions that could address the shortcomings of our approach.

An open task for future extensions is the minimization of store buffer graphs as it could reduce exploration effort in a model checker. Much of the non-determinism due to store buffers is not observable by other processes and thus, bears the potential to be removed from store buffer graphs without affecting verification results. One idea towards that direction is to make sequences of process local steps simply atomic. If a program reaches such a sequence of local steps and still contains entries in the store buffer, then the resulting store buffer graph will contain all possible flush sequences within that sequence of local steps. However, a flush commutes with local steps, i.e., the reached state will be the same in all of the sequences. Thus, it is sufficient to consider only one such sequence and drop the others. Another idea could be to reduce the number of variables that are introduced by the compiler in order to compute temporary results. A compiler often introduces temporary variables that are necessary to perform computations or evaluations of high-level expressions based on the instruction set of a processor. Most verification tools support high-level expressions and thus do not need these temporary variables to compute the result of such an expression. Fewer variables result in a smaller state vector. A smaller state vector enables model checkers to explore more states before they run out of memory. Besides being beneficial for model checking, store buffer graph minimization could reduce complexity of programs and thereby also help with correctness proofs, e.g., by having more concise invariants. However, as already mentioned before, reducing the size of store buffer graphs also requires a weaker bisimulation equivalence than the one used in Theorem 1. Otherwise, the soundness of the approach would be no longer guaranteed, which is why we refrain from further modification of transformed programs throughout this thesis.

A probably less obvious restriction to our approach is that it implicitly assumes fences at method invocation and return. By unfolding the behavior of a sequential program, the symbolic execution starts with an empty store buffer, which is similar to the program having just passed a fence. The symbolic execution stops once, it reaches an empty store buffer while being at the program location of a return statement, which is similar to the return statement being a fence. This restriction does not represent actual weak memory semantics, but it is there for a practical reason: we had to define a starting and end point of the symbolic execution. More importantly, it is not an actual limitation of our approach. Method calls that occur in a program can be inlined in the surrounding method. Thus, reorderings that would occur around method boundaries can be represented in store buffer graphs, but need some extra effort in that sense. However, because it is a static approach and recursion ends are determined dynamically, the approach is restricted to non-recursive functions and methods only. The assumption of an initially empty store buffer is due to us not knowing a priori what the context of a program can be. An extension that would allow for specification of initial content of store buffers prior to the symbolic execution is straightforward and thus, not a limitation of the approach. An extension that would inline methods into each other whenever a method call occurs in the program code would have to consider potential namespace conflicts, but should also be straightforward.

Our reduction is applied to program code. Thus, it can only consider statically available information of a program which leaves some possible challenges for verification. One such challenge is pointer aliasing, i.e., different variables or even expressions in a program can refer to the same memory location. By construction of a store buffer graph, we symbolically execute a sequential program in order to determine possible store buffer states and in order to distinguish whether a read can be early or not. That distinction is made by the names of pointer variables, which represent memory locations. Our approach does not incorporate alias analysis. Thus, the store buffer graph construction fully relies on the alias detection of the LLVM framework [LA04], which we use in order to compile programs into their low-level representation. If an alias remains undetected, it could falsify a constructed store buffer graph, since possible early reads may be missed and in PSO, writes may be enqueued in different buffer queues corresponding to different memory locations although they should be in the same queue. We are aware that the compiler does not fully rule out pointer aliases and thus, we suggest adding an additional alias analysis as a preprocessing step to the our proposed approach in future extensions. Alias analysis is a challenging problem on its own it goes beyond the focus of this thesis. For an extensive overview of state-of-the-art alias analyses techniques that could be applied, we refer to [CNW13].

As a last point, it should be mentioned that our reduction does not support RMO, nor ARM or Power (the realworld counterparts of RMO). Although programs under RMO can be transformed into equivalent SC programs, we do not think that a reduction as the one proposed in this thesis (in case it would be extended to RMO) would result in programs of manageable size. In other words, we do not think the approach would be practical. The strength of our reduction stems from the following fact: we transform programs, in which each process seemingly has its own view of the memory due to store buffers, into SC programs, in which processes share one global view of the memory. Parts of a program that previously allowed processes to observe values that differ from values observable by other processes are transformed to local behavior of the new SC program. However, a reduction that achieves that for programs under RMO is not possible. In RMO, processes can potentially observe a write of another process early by taking it from their store buffer. A process cannot read early from all store buffers of a multicore processor, but only from a subset. Otherwise, a reduction would be simple since all processes would always observe the most recent values, just like in SC. Since the process scheduling is usually not known a priori, RMO behavior must be over-approximated by allowing each process to potentially, but not necessarily, observe values early from all other processes. Because the store buffers in RMO are shared among different processes, we cannot transform the behavior related to the store buffer into local behavior of a process as we did for TSO and PSO. Instead, it changes with the number of processes in a parallel composition and thus, is non-compositional. Hence, if we were to adapt our reduction to the RMO setting, we would have to construct store buffer graphs for the full parallel composition. It would also require many auxiliary variables in order to track which writes in a store buffer are observable by different processes. All this information would have to be encoded in a program state. Adding reordering of instructions, an SC representation of a program under RMO explodes in its size unless under-approximating techniques are applied to it. Therefore, other approaches like [AMSS10, AKNT13] should be preferred in this case.

4 Weak2SC – The Implementation

In the previous chapter, we introduced the reduction of programs under TSO or PSO to equivalent programs under SC. The reduction is generic up to the point, where a store buffer graph is created. However, the transformation from a store buffer graph to a particular encoding of the store buffer graph using SC semantics, the new SC program, highly depends on the choice of the verification tool and its input language. We have chosen the model checker Spin [Hol03] and the theorem prover KIV [EPS+14] as our verification tools, the model checker as bug-finding tool and the theorem prover for correctness proofs. We implemented all of the steps of our approach from parsing an LLVM IR program to the transformed programs in a tool called Weak2SC [Tra16].

In this chapter, we introduce the implementation of Weak2SC. Along the lines, we also introduce an example that is prone to errors due to weak memory models and which we will use in order to exemplify the application of our approach throughout the thesis. The example is a work-stealing queue implementation by Arora et al. [ABP98]. In this chapter, we will use it to showcase our program transformation to an SC program incorporating weak semantics. In Chapter 6, we will use it as a running example for verification of linearizability under weak memory models. In the following section, we start with the architecture of the tool and discuss the choice of existing tools that helped us implement the approach. Section 4.2 introduces our running example. In Section 4.3, we start with the LLVM IR parser that produces a model of the program code and thereby allows us to implement the remaining steps of the approach in a model-driven way. We go on by providing details about the symbolic execution of programs, which is used to explore store buffer states and to construct the store buffer graph. Out of the store buffer graph, we generate the new

SC programs using templates. The template-based program generation is explained in Section 4.4 for both verification tools, Promela for Spin and a predicate logic encoding for KIV. We conclude this chapter with a discussion of the implementation and possible extensions.

4.1 Architecture of Weak2SC

Weak2SC was implemented as an extension to the Eclipse[1] Integrated Development Environment (IDE). Eclipse is a widely used open source developer tool. It is common among web, desktop and mobile application developers, embedded systems developers and many other types of domain specific developers. One of the reasons for its success and why we chose Eclipse as the foundation of Weak2SC is its ability for customization. It is designed to be extended with features, such that program developers can customize their IDE to their needs. By making Weak2SC an extension to Eclipse, we enable program developers to integrate Weak2SC in their development process within Eclipse without any considerable effort.

The architecture of Weak2SC is visualized in Figure 4.1. It consists of a set of plugins, each implementing different functionalities in the programming language Java. Plugins in Eclipse are software components. Their intention is to accomplish separation of concerns. Unlike UML component diagrams [Obj15b], Figure 4.1 does not visualize the provided or required interfaces of each component. Instead, edges were added in case a plugin "uses" or "extends" another plugin. Plugins in Eclipse require or provide Java packages and thus, are not shown in the figure as they would clutter the tool structure.

The plugins shown in Figure 4.1 can be grouped into four main feature implementations of Weak2SC: 1. the LLVM IR parser, 2. the store buffer graph construction, 3. the store buffer graph visualization and 4. the transformation of store buffer graphs into new program encodings. Each of the plugins visualized in the figure belongs to one of the four features. The features were implemented based on other tools that are commonly used in Eclipse. As such, we used EMF[2] for the design of our meta models, Xtext[3] for generating an LLVM IR parser, GMF[4] for creation of a graphical interface to the store buffer graphs and Acceleo[5] in order to define a model-to-text transformation that generates the new SC programs.

The Eclipse Modeling Framework (EMF) is an implementation of the Meta-Object Facility (MOF) standard [Obj15a] for model-driven software engineering by the OMG.

[1] www.eclipse.org
[2] www.eclipse.org/emf
[3] www.eclipse.org/xtext
[4] www.eclipse.org/gmf
[5] www.eclipse.org/acceleo

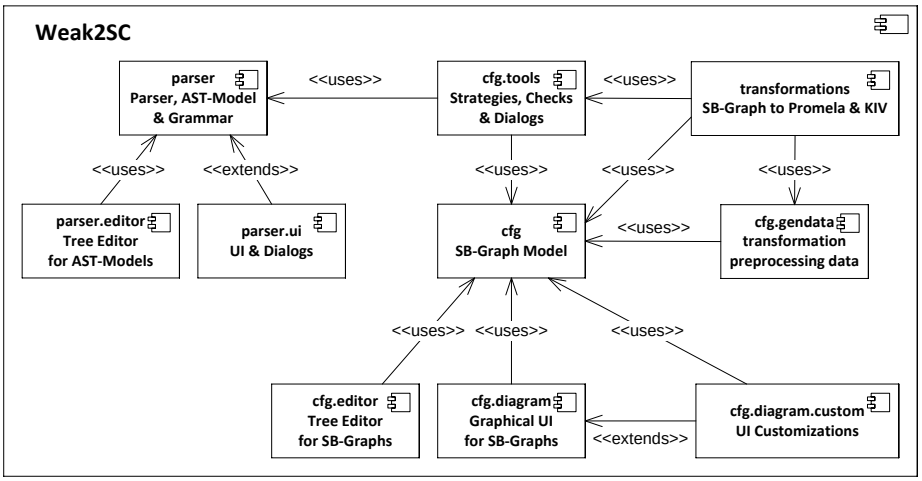

Figure 4.1: Architecture of WEAK2SC – external components are excluded

In EMF, meta models are described via Ecore models. Ecore models can be used to define all sorts of meta models, e.g. LLVM IR language, store buffer graphs, Petri nets or UML. In contrast to Ecore models, UML Class Diagrams are limited to object oriented software systems. One of the reasons why we use EMF (and why it is a popular Eclipse tool) is because it generates Java code for the instantiation of the Ecore models. The generated code contains built-in features like object listeners and model serialization to disk. Besides this, EMF can generate tree editors for the instantiation and modification of models. The parser generator Xtext makes use of EMF by deriving the abstract syntax tree from language grammars and generating an Ecore model for it. The latter is instantiated by the generated parser. The Graphical Modeling Framework (GMF) can be used to generate graphical editors based on the meta models that are provided as Ecore models. Acceleo is an implementation of the MOF Model To Text Transformation Language standard [Obj08].

In the following, we will go over each plugin and explain its purpose as well as how it is tight to the other plugins.

parser This plugin implements the LLVM IR parser and contains the meta model defining the abstract syntax tree (AST) of the parsed language. The latter is instantiated by the parser. Most parts of the plugin are generated by the Xtext parser generator based on a grammar that was provided for the LLVM IR language. The provided grammar is derived from the partially documented LLVM IR grammar that was available from the source code implementing

the LLVM framework. Other parts were derived from the LLVM language reference manual[6].

parser.editor This plugin implements a tree editor for the models that are instantiated by the *parser* plugin. The plugin enables inspection and modification of parse results before they are processed (before the store buffer graphs is created) and thereby helped us debugging WEAK2SC. It was generated from the Ecore model that was derived from the LLVM IR grammar.

parser.ui This plugin contains the text editor for the LLVM IR language. Together with the parser, it is automatically generated based on the grammar that we had to provide in the *parser* plugin. Key features of this editor are text highlighting of key words, inline parse error reports and a tree-based structural outline (similar to *parser.ui*) of the parsed code. The latter is parsed on the fly.

cfg This plugin contains the Ecore model for the store buffer graphs. Furthermore, it contains code that implements store buffer graphs (EMF generated). The name of this and the following plugins stems from our early development, when we just thought of representing the control flow of a program with respect to weak memory models. Although control flow graphs and store buffer graphs under SC are equivalent, they differ for weak memory models. In order to emphasize this difference, we renamed the constructed graphs to store buffer graphs, but did not adapt the plugin names for technical reasons.

cfg.tools The plugin contains most of our manual implementation. Most importantly, it contains the different symbolic exploration strategies that are required for the memory models SC, TSO and PSO. Besides this, the plugin contains several sanity checks that are performed before the store buffer graph of a program is explored, e.g., in order to make sure that the program does not have unfenced writing loops. Furthermore, we implemented dialogs that help developers in Eclipse to go through the transformation process as we intended it (see Figure 3.1 on page 32).

cfg.editor Similar to *parser.editor*, this plugin is an EMF generated tree-editor for store buffer graphs. It was used for debugging purposes.

cfg.diagram The graph visualization of store buffer graphs is implemented in this plugin. It is a GMF generated graph editor, which we use as a viewer only. A GMF generated editor contains many features that would be otherwise difficult to implement, e.g., automated graphical layouts. However, the input that we

[6] www.llvm.org

had to provide to GMF in order to generate the editor lies in the *cfg* plugin together with the Ecore model of store buffer graphs. GMF basically needs a definition of graphical elements and a mapping of the graphical elements to model elements in order to generate an editor.

cfg.diagram.custom This plugin contains manually implemented customizations of the graphical visualization in *cfg.diagram*, e.g., label encoding of store buffer values. A drawback of generated code is that its customization and maintenance is difficult. The plugin *cfg.diagram* alone contains more than 7000 lines of generated code. Furthermore, the code is regenerated with every change or extension in our meta models, which is why we externalized the manual customizations to a separate plugin.

cfg.gendata This plugin contains a meta model for data that is computed before the transformation is performed. It exists only for convenience and represents technically redundant information like different sets of program variables, e.g., local, global or all. It also contains mappings of variable or function names to potentially new names. This information could also be derived during transformation, but it would complicate the transformation definition with algorithmic computations. We want the transformation definition to be simple and easy to extend.

transformations This plugin contains templates that we defined in Acceleo, in order to transform our models to the textual representation of the new SC programs. It also contains the implementation of the transformation pre-processing (instantiating the *gendata* model) and dialogs for triggering the transformation with different modes. For Promela, we have two modes: 1. store buffer graph encoding based on the proposed transformation in Chapter 3 and 2. a Promela model of the program including an operational encoding of the chosen memory model semantics. For KIV, the modes allow to choose between a local state encoding and a global state encoding. The local variant defines the state of a process separately from the shared state, while the global variant defines the program state as a single tuple that represents the shared state and the state of all processes. In addition, one can choose to have natural number or integer encoding. Sometimes the latter cannot be avoided due to negative numbers used by the implementation. The former have better support in KIV in terms of term simplifications that are applied automatically to a lemma during a proof. The choice between global and local state encoding highly depends on the proof obligations which are to be proven correct. These are discussed in Chapter 6.2.

At the time of writing this thesis, the overall implementation of Weak2SC has ca. 160k lines of code (LOC) of which only ca. 10k LOC had to be manually implemented. It is available on a git repository[7] on Github. The complete repository, our later case studies for the experiments and an update site for the installation in Eclipse is made available on the DVD attached to this thesis. A short tutorial on how to use Weak2SC is part of the tool and can be found in Eclipse Help after installation of Weak2SC. In the following sections, we will go through the transformation process step-by-step and in detail. For this purpose, we introduce a running example, which is an implementation of a work-stealing queue by Arora et al. [ABP98]. In later Chapters, we will also use it as a verification example as it is not correct under TSO and PSO.

4.2 Case Study – Work Stealing Queue

As a running example throughout this thesis, we will use a queue implementation by Arora et al. [ABP98]. It is an interesting case study for several reasons. It is correct under SC semantics, but it needs fences for correctness under TSO and PSO. By correctness, we mean linearizability [HW90], but we will come back to different correctness criteria later in Chapter 5. Furthermore, the queue implementation is a work stealing queue. In particular, different processes have different roles, i.e., processes can be the owner of the queue or a stealer of queue elements. In addition, the implementation also exemplifies how a double compare-and-swap operation can be implemented using a normal compare-and-swap (CAS) operation.

Following the pseudo code provided by Arora et al. in [ABP98], we implemented the queue in C. Figure 4.2 shows the complete source code of our implementation. The implementation uses an array pointed to by the variable *deq*. Furthermore, it uses two shared variables *bot* and *age*. The variable *bot* is the index value of the array that represents the end of the queue (precisely, the first free slot in the array). The variable *age* is an encoding of two values in one 32 bit integer. The first 16 bits encode a value *tag*, which is used to avoid the ABA problem. The latter 16 bits encode the value *top*, which is the index value representing the start of the queue in the *deq* array. The encoding of two variables within one enables the algorithm to perform a double compare-and-swap operation using regular a CAS instruction. Processes can be the owner of the queue or they are a stealer. An owner process adds and removes elements using the operations *pushBottom* and *popBottom*. Both operate on the bottom end of the queue. Stealer processes can try to remove an element from the top end of the queue. By having stealer and owner processes operate on different ends of the queue, the algorithm avoids contention.

[7] http://www.github.com/oleg82upb

```
1  unsigned int *bot;
2  int *deq, *age;
3
4  int popTop(){
5    int oAge = *age;
6    unsigned int locBot = *bot;
7    if (locBot <= (oAge >> 16))
8    {
9      return NULL;
10   }
11   int elem = deq[oAge >> 16] ;
12   int nAge = oAge;
13   nAge = (((nAge >> 16) + 1) << 16)
14        | (nAge & 0xFFFF);
15
16   if (CAS(age, oAge, nAge))
17   {
18     return elem;
19   }
20   return ABORT;
21 }
22
23 void pushBottom(int elem){
24   unsigned int locBot = *bot;
25   deq[locBot] = elem;
26   locBot++;
27   *bot = locBot;
28 }
29 int popBottom(){
30   unsigned int locBot = *bot;
31   if( locBot == 0)
32   {
33     return NULL;
34   }
35   locBot--;
36   *bot = locBot;
37   int elem = deq[locBot];
38   int oAge = *age;
39   if ( locBot > (oAge >> 16))
40   {
41     return elem;
42   }
43   *bot = 0;
44   int nAge = (0 << 16)
45        | ((oAge & 0xFFFF) + 1);
46   if (locBot == (oAge >> 16))
47   {
48     if (CAS(age, oAge, nAge))
49     {
50       return elem;
51     }
52   }
53   *age = nAge;
54   return NULL;
55 }
```

Figure 4.2: Work Stealing Queue by Arora et al. [ABP98]

A *pushBottom* operation adds an element to the queue by simply writing to the first free slot in the array, which is indexed by *bot*. By incrementing *bot*, the new element is made visible to other processes. A *popBottom* operation removes an element from the bottom end and if necessary, resets the array by setting *top* and *bot* to zero. Otherwise, the queue would move through the array with ever increasing values of *top* and *bot* until the array length is reached. The algorithm assumes that the array length is chosen sufficiently high, such that it is never reached by *top* and *bot*. If the *popBottom* operation detects that the queue is empty, ($top < bot$ and $bot \neq 0$), then it resets the values *top* and *bot* to zero (lines 43-53). Before the queue is reset, the operation has to ensure that the last element is not stolen by another process (lines 46-52). The latter can happen if $top = bot$. The CAS in line 48 resets the *top* value by replacing the *age* value with a new value *nAge*. The latter contains an incremented *tag* value and a value $age = 0$. If the CAS succeeds, the owner process also becomes the owner of the removed element *elem*. Otherwise, a stealer process must have taken it already and the *popBottom* operation returns *NULL*. In that case, the owner process can safely write the new value of *age* in line

53, because all other processes will see an empty queue and the owner is the only process who can add elements to it. A stealer process starts with reading *age* and *bot* in lines 5 and 6. If it observes $top \geq bot$, then it observes an empty queue and returns *NULL* (lines 7-10). If this is not the case, then stealer needs to make sure that it does not conflict with other processes. A stealer process modifies only the *top* end of the queue. In order to do so, it creates a new value of *age* (lines 13-14) with *top* being incremented and *tag* being equal to the previous value. The new value *nAge* replaces the old value of *age*, if the CAS in line 16 succeeds. Then, the stealer has successfully stolen the element from the queue. The CAS can only fail if another process has either stolen the element or if the element was the last and the owner process has successfully reset the queue. If the CAS in line 16 fails, then the stealer process aborts without retry.

In order to reason about the weak memory model effects to a program, we need its low-level representation, because it reveals the position of all memory instructions. For that purpose the code in Figure 4.2 was compiled into the intermediate representation provided by LLVM. The resulting code is shown in Figure 4.3. The figure shows all three operation implementations and the variable declarations. In addition, line 13 and 56 indicate where a fence is required. For the *popBottom* operation, the figure shows only an excerpt of the code. The complete implementation can be seen in the Appendix B.

Methods in LLVM IR are defined by a return type, a name that becomes a global variable and a list of typed parameters. The code is structured in terms of labeled control flow blocks. Each control flow block ends with either a return (*ret*) or a goto (*br*) to another block. The latter can also be conditional. Each line represents a single instruction, e.g., *load* for a read, *store* for a write or *getelementptr* for a pointer computation. To be more accurate, the code is in Single Static Assignment (SSA) form [CFR+91]. Variables in LLVM IR are prefixed by either "@" or "%". Global variables have the prefix "@" while local variables (corresponding to registers) are prefixed by "%". Variable names are preserved after the compilation, e.g., *deq*, *bot* and *age*. However, the code also contains several auxiliary variables that are introduced by the compiler. Most of the auxiliary variables are enumerated (%0, %1 and so on). In addition to the names, the code also contains type annotations for most variables, e.g., whether a variable is 32 bit integer (*i32*) or a pointer value (**i32*) to a 32 bit integer. At this stage, the types are technically no longer necessary as all values are essentially register values, but they help understanding the code.

After compilation of the C program, the code obviously became longer in terms of lines of code as all high level C program constructs are compiled into single instructions implementing them. Some of the instructions may need some additional explanation. The *pushBottom* method starts with three *load* instructions where the

```
1  @bot = common global i32* null, align 4
2  @deq = common global i32* null, align 4
3  @age = common global i32* null, align 4
4
5  define void @pushBottom(i32 %elem) nounwind optsize {
6  entry:
7    %0 = load i32** @bot, align 4, !tbaa !0
8    %1 = load i32* %0, align 4, !tbaa !3
9    %2 = load i32** @deq, align 4, !tbaa !0
10   %idx = getelementptr inbounds i32* %2, i32 %1
11   store i32 %elem, i32* %idx, align 4, !tbaa !3
12   %inc = add i32 %1, 1
13   <<< fence required for PSO >>>
14   store i32 %inc, i32* %0, align 4, !tbaa !3
15   ret void
16 }
17
18 define i32 @popTop() nounwind optsize {
19 entry:
20   %0 = load i32** @age, align 4, !tbaa !0
21   %1 = load i32* %0, align 4, !tbaa !3
22   %2 = load i32** @bot, align 4, !tbaa !0
23   %3 = load i32* %2, align 4, !tbaa !3
24   %shr = ashr i32 %1, 16
25   %cmp = icmp ugt i32 %3, %shr
26   br i1 %cmp, label %if.end, label %return
27
28 if.end:
29   %4 = load i32** @deq, align 4, !tbaa !0
30   %idx = getelementptr inbounds i32* %4, i32 %shr
31   %5 = load i32* %idx, align 4, !tbaa !3
32   %add5 = add i32 %1, 65536
33   %6 = cmpxchg i32* %0, i32 %1, i32 %add5 seq_cst
34   %7 = icmp eq i32 %6, %1
35   %. = select i1 %7, i32 %5, i32 -2
36   br label %return
37
38 return:
39   %retval.0 = phi i32 [ -1, %entry ], [ %., %if.end ]
40   ret i32 %retval.0
41 }
42 define i32 @popBottom() nounwind {
43 entry:
44   %0 = load i32** @bot
45   %1 = load i32* %0
46   %cmp = icmp eq i32 %1, 0
47   br i1 %cmp, label %return, label %if.end
48
49 if.end:
50   %dec = add i32 %1, -1
51   store i32 %dec, i32* %0
52   %2 = load i32** @deq
53   %idx = getelementptr inbounds i32* %2, i32 %dec
54   %3 = load i32* %idx
55   %4 = load i32** @age
56   <<< fence required for TSO >>>
57   %5 = load i32* %4
58   %shr = ashr i32 %5, 16
59   %cmp1 = icmp ugt i32 %dec, %shr
60   br i1 %cmp1, label %return, label %if.end3
61
62 if.end:
63   %dec = add i32 %1, -1
64   store i32 %dec, i32* %0, align 4, !tbaa !3
65   %2 = load i32** @deq, align 4, !tbaa !0
66   %idx = getelementptr inbounds i32* %2, i32 %dec
67   %3 = load i32* %idx, align 4, !tbaa !3
68   %4 = load i32** @age, align 4, !tbaa !0
69   %5 = load i32* %4, align 4, !tbaa !3
70   %shr = ashr i32 %5, 16
71   %cmp1 = icmp ugt i32 %dec, %shr
72   br i1 %cmp1, label %return, label %if.end3
73
74 if.end3:
75 ...
76
77 return:
78   %retval.0 = phi i32 [ -1, %entry ], ...
79   ret i32 %retval.0
80 }
```

Figure 4.3: LLVM IR code after compilation of the code in Figure 4.2. Includes annotation for required fences. It shows only excerpt of the method *popBottom*.

C code starts with an assignment. The reason for it is that the shared variables *bot* and *deq* are pointers variables. Thus, the first *load* of *bot* at line 7 fetches the pointer value. The second *load* fetches the value pointed to by the pointer. It is similar for the *deq* variable. However, since the access to the array is a writing one, only one *load* is required to fetch the pointer value. A *getelementptr* computes the memory location that is relative to the fetched pointer value. That is also the location that the consecutive *store* writes to in order to complete the single line assignment "*deq[locBot] = elem;*" from the C code (line 25). The other methods proceed in a similar way and we will not go into details here as it is just the LLVM IR representation of the C code in Figure 4.2. Other noteworthy instructions are *ashr*, which shifts (in our case 16) bits of an integer value to the right, *icmp*, which is an integer compare operation, *select*, which assigns a value to a variable based on a boolean condition, *phi*, which assigns a value to a variable based on which was the previous control flow block before the program jumped to the block containing the *phi* instruction, and *cmpxchg* which is a compare-and-swap operation.

This completes our introduction of the running example. In the following Section, we will use it in order to demonstrate the most important steps involved in our tool implementation of the presented approach, Weak2SC.

4.3 From LLVM IR to a Store Buffer Graph

Weak2SC contains a built-in LLVM IR parser. The parser parses the textual input and creates a model out of it. This greatly helps with processing the program input, because our algorithms can be implemented in terms of walks over a tree model instead of performing many string comparisons for every line of code. We did not use the LLVM IR parser provided by the LLVM framework, because it is implemented in C++ and therefore is difficult to integrate into the Eclipse IDE, which uses Java as its programming language. Instead, we developed a parser from scratch using the Xtext framework. Xtext is parser generator framework for Eclipse, which generates parsers from grammars together with a textual editor for the given language that uses the generated parser. All of the features generated by Xtext come as Eclipse plugins and thus are integrated in the Eclipse IDE. The only thing we had to provide is a grammar for the LLVM IR language.

The LLVM IR parser implementation in the LLVM framework is partially documented with excerpts of a grammar for the LLVM IR language. We used it as a starting point for our grammar and refined it whenever or wherever we felt, it was necessary or where it was outdated. On the other hand, we were able to leave out parts that we do not require for our implementation of Weak2SC, e.g., the structure of generated code annotations.

The parse result is a model representation of the parsed program, the abstract syntax tree (AST). Such models also have a meta model. The latter defines all possible model instances or in other words, all valid program inputs that can be represented by an AST. The meta-model is derived from the LLVM IR grammar that we provided to Xtext and it is a starting point for all further computations like the exploration of the store buffer graph.

Before Weak2SC continues with the exploration of the store buffer graphs for weak memory, two checks are performed for the program. In particular, Weak2SC checks for writing loops without fences and whether the program contains instructions that the tool does not support, e.g., vector based operations that are non-atomic. In order to perform these checks, the control flow of the program is required. Thus, an SC-based exploration is performed at beginning in order to obtain the store buffer graph with SC semantics, which is just the control flow graph of the program. The checks generate warnings that are given as feedback to the user or even prevent the user from applying Weak2SC to a program, e.g., when a writing loop without fences is found.

If the program passes the checks, the user can select the memory model for which he or she wants to explore the store buffer graphs for. Figure 4.4 and 4.5 show the store buffer graphs for the *pushBottom* operation of our case study, the Arora et al. [ABP98] (see Figure 4.3 for the LLVM IR code). The different semantics corresponding to the memory models TSO and PSO can be observed immediately from both figures. Note that under SC the store buffer graph is just a simple sequence of states. Therefore, we refrain from showing it here.

The graphical representation of the editor in Weak2SC deviates from what we used in Chapter 3, but should be self-explaining for the most part. The nodes are labeled by the program location and symbolic store buffer contents. However, if the buffer is empty, only the program location is used as a label. Nodes that represent the program state before the first statement (resp. after the *ret* statement) are highlighted in blue. Thereby, the begin and end of the behavior represented by the store buffer graph can be seen immediately. This helps especially with larger store buffer graphs and when the built-in automatic layout fails to organize the elements from top to bottom. The editor provided with Weak2SC also allows for later adjustment of the graph layout. The edges in the store buffer graph editor are labeled and colored with respect to their semantics. The example in the figure does not contain all types of transitions that differ visually. Anyway, for the sake of completeness, we provide a list of the colors that we use in order to highlight particular transitions:

gray transitions represent local statements. These statements cannot be observed by other processes as only the local state of the executing process is affected

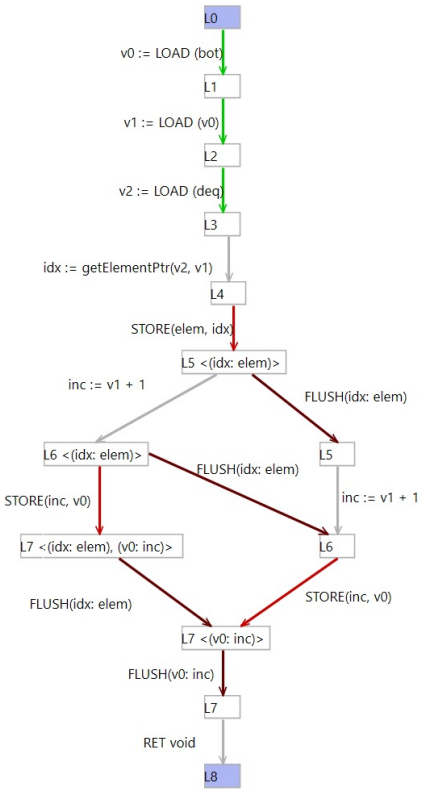

Figure 4.4: Store buffer graph of *pushBottom* method under TSO

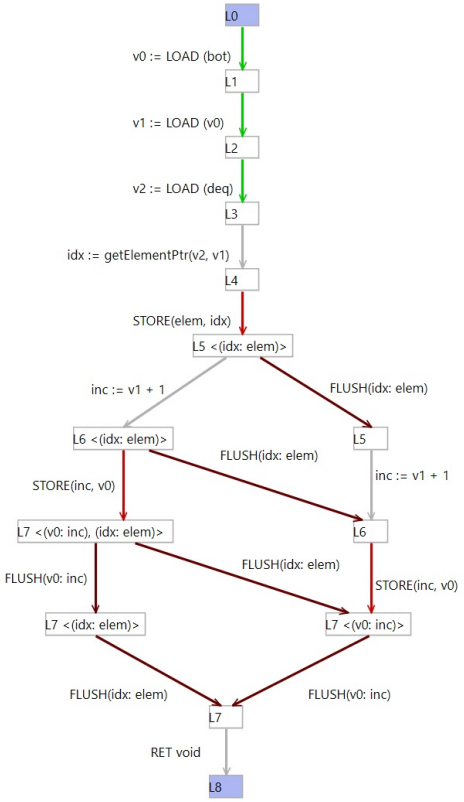

Figure 4.5: Store buffer graph of *pushBottom* method under PSO

by them.

green transitions represent reads (*load*) from memory. Transitions that represent early reads have their labels replaced by an assignment, but remain green, in order to visualize that they correspond to a read in the original program.

red transitions represent writes (*store*) to memory. However, these transitions do not actually write to the memory, but add a memory location-value pair to the store buffer. Thus, they affect only the local state of the executing process.

brown transitions represent flush transitions. A flush transition removes a memory location-value pair from the store buffer and updates the shared memory at the given location with the given value.

blue transitions have fence-like semantics, i.e., they block execution until the store buffer is emptied. Compare-and-swap and fence are two examples for such instructions.

yellow transitions represent memory space allocating instructions.

The exploration of the store buffer graph is essentially a breadth-first-search (BFS) algorithm. Starting at an initial location *L0* with an empty store buffer, it explores state after state of a process in BFS manner and constructs the store buffer graph as it explores the state space. We implemented three exploration strategies corresponding to the memory models SC, TSO, PSO. Essentially, the exploration strategies vary only in two aspects: 1. representation of the store buffer in a node and 2. computation of the next (outgoing) transitions of a state. The strategy for SC exploration is the basic strategy and is extended by the TSO exploration strategy. Similarly, the strategy for PSO extends the TSO exploration strategy.

In SC, the store buffer is ignored as it is always empty and flush transitions do not exist. Thus, the exploration simply explores the control flow of the program. The other strategies are based on it.

In TSO, the store buffer is a single FIFO queue for all memory locations and value pairs. In contrast to the SC exploration, the store buffer entries have to be considered during exploration. Flush transitions extend the set of outgoing transitions that are computed by the SC exploration. In addition, early reads must be taken care of. The SC exploration strategy creates already read transitions. All we had to adapt for TSO besides adding flush transition is to refine the construction of read transitions in the store buffer graph. In TSO, every read is checked whether it is an early read or not. If it is an early read, an assignment is constructed that assigns the latest symbolic value in the store buffer for the requested memory location to the target register variable. Otherwise, a regular read transition is constructed.

In PSO, the store buffer contains FIFO queues of values for each memory location. In order to avoid different implementations of symbolic store buffers and algorithms e.g., for checking equivalence of buffer contents, we use one representation for both TSO, and PSO. In order to achieve this, we represent a store buffer as a list of pairs, where the pairs are pairs of memory location and again a list of values. Thereby, the list of pairs models the FIFO queue under TSO and is ignored in PSO. A pair in TSO contains always just one value. In PSO it contains a list of values to represent the per location FIFO queues. In addition, the exploration strategy for PSO refines the constructions of flush transitions. In particular, when the store buffer contains entries for different memory locations, then PSO allows for a flush of the oldest value to each of the different memory locations. This can also be seen in Figure 4.5 at the node *L7 <(v0: inc), (idx: elem)>* as it has two outgoing flush transitions. Note that by flushing the value *inc* to location *v0*, the order of writes is changed with respect to program order. For the algorithm, it means that the new element becomes visible to other processes before the element value *elem* is written to the memory. It

is particularly this behavior of the *pushBottom* operation that must be avoided under PSO by adding a fence between the writes.

In TSO and PSO, write-def-chains (WDC) are detected before the actual TSO or PSO specific exploration. In order to find writes that are involved in a WDC, a breadth-first-search is performed starting at every write in the program that is part of a loop. If a variable used by the write is redefined before it is guaranteed to be flushed, then we consider the write to be part of a WDC and mark it as such. The strategies for TSO and PSO, both check for each write whether it is marked before a write transition is constructed. If a write is marked, then a copy of the involved variable (the one potentially redefined before flush) is created, unless it exists already. According to our transformation in Chapter 3, we modify the transition to be an assignment and a write. The assignment copies the value of the copied variable at the state before writing. In the write, the variable that is copied is replaced by the copy. Consequently, the symbolic store buffer in the state that is reached by the modified write transition will also contain the variable copy instead of the original variable.

This closes our explanation of the strategies for exploration of store buffer graphs with SC, TSO and PSO semantics. For further details, such as examples, models, and the full implementation of Weak2SC, we refer to our Github repository[8]. In the following section, we will elaborate on the generated programs that Weak2SC produces and which are the input to either the model checker Spin [Hol03] or the theorem prover KIV [EPS+14].

4.4 Template-based Generation of new Programs

Having explored the store buffer graph of a program, we can now go on with the transformation by generating the new SC program that is meant to be the input to either a model checker or a theorem prover. For this purpose, we defined several model-to-text transformations in Weak2SC. The transformations were defined within the Acceleo framework, which implements the OMG standard [Obj08] for model-to-text transformations in model-driven development.

Weak2SC generates two types of Promela program models. The first is a store buffer graph encoding in Promela as proposed in our approach in Chapter 3.2. The second is based on different operational memory models for SC, TSO and PSO which we defined in Promela. The transformation based on operational memory models follows the approach in [TMW13] that we extended for PSO in [TW16]. Weak2SC automates both types of transformation. Both types of generated Promela models

[8] www.github.com/oleg82upb

can be used in order to explore program behavior under weak memory model. The models based on operational memory models are a good alternative if our program assumptions are not met (see Sec. 3.1). Essentially, we assume programs to have no unfenced writing loops and to be in SSA form.

Weak2SC also defines two types of program encodings for the theorem prover KIV. These differ in the way a program state is represented. One variant captures the whole state (shared and local) in one tuple, while the other variant defines separate tuples for shared and local states. We refer to the former variant as a global encoding and as a local encoding to the latter. Since linearizability [HW90] is the correctness criterion we focus on, we need to consider the available proof obligations and theory for linearizability in KIV. These can be categorized into global [DSW07] and local proof obligations [DSW11b]. The global (resp. local) proof obligations require a program behavior definition that is based on a global (resp. local) state encoding. While the global proof obligations are more general, the local proof obligations can reduce the proof effort [DSW11b, TWS12, TTSW14] significantly. Weak2SC can generate both types of encodings from a store buffer graph. The proof obligations are discussed in Sec. 6.2 while in this chapter, we focus only on providing an encoding of the program behavior.

The transformations are defined in terms of templates. A transformation template defines how an element in a model, e.g., a store buffer graph, is translated to the target language. A template divides the target language representation of a model element into a universally identical part (e.g., function definition with brackets for parameters and statements) and a part that is model element specific (e.g., the function name). In addition, templates help to structure the transformation into smaller parts that are easy to understand or handle by defining separate templates and their composition, e.g., by defining a template for a function and one for each type of statement that could be used in a function. A template-based transformation essentially defines a walk over the model (based on the template structure), where each model element is handled by its respective transformation template. Each template results in a piece of text, which is embedded in the textual result of a surrounding template. Thereby, the outermost template defines the complete transformation of a model to its textual representation in the target language.

Due to the size of our meta-models (more than 100 classes) and the fact that we defined several transformation variants (all together over 2k LOC), we refrain from showing the exact transformation templates. Instead, we will explain the structure of the transformed new programs in the respective target language. This will be based on our case study, the work-stealing queue by Arora et al. [ABP98] (see Section 4.2).

```
1  #define MEM_SIZE 15 //size of memory
2  #define null 0
3
4  short memory[MEM_SIZE];
5  short memUse = 1;    //next free cell
6
7  //pointer computation
8  inline getelementptr(type, instance,
9      offset, targetRegister)
10 {
11   atomic{
12   assert(offset <= type);
13   targetRegister = instance + offset;
14   }
15 }
16
17 //compare-and-swap
18 inline cas(adr, old, new, result)
19 {
20   atomic{
21   result = memory[adr];
22   if
23     :: memory[adr] == old
24       -> memory[adr] = new;
25     :: else -> skip;
26   fi;
27   }
28 }
29 //memory allocation
30 inline alloca(type, targetRegister)
31 {
32   atomic{
33   targetRegister = memUse;
34   memUse = memUse + type;
35   assert(memUse < MEM_SIZE);
36   }
37 }
38
39 //Stubs
40 proctype process1(){
41   //TODO: empty stub
42 }
43
44 proctype process2(){
45   //TODO: empty stub
46 }
47
48 init{
49 atomic{
50   alloca(1, bot);
51   alloca(1, deq);
52   alloca(1, age);
53   run process1();
54   run process2();
55   }
56 }
```

Figure 4.6: Excerpt of the generated Promela model for the program in Figure 4.3.

4.4.1 Generating Promela Programs

Our Promela programs generated from store buffer graphs can be divided into two major parts. The first defines parts of a Promela model that for the most part remains identical among different programs, i.e., memory definition, pointer computation, memory allocation, empty process stubs and an initial state definition. The second part defines the behavior that corresponds to store buffer graphs in terms of inline statements. This way, an inline statement can be used in a similar style to method calls in regular programming languages like C or C++. In the following, we explain both parts using our case study, the Arora et al. work stealing queue (see Figure 4.3 for the LLVM IR code and Figure 4.4 for the store buffer graph of the method *pushBottom* that is used here as an example).

Memory Representation and Access

Figure 4.6 contains an excerpt of the generated Promela code for our case study. The foundation of our generated Promela models is our representation of the memory.

We use an array *memory* whose length can be adjusted to specific verification needs, but has to be defined statically. Each cell in the *memory* represents one value, be it boolean, byte, short or integer. Thus, we have no built-in support for non-atomic reads or writes, e.g., a write of a 64 bit value on 32 bit (word length) architecture would need at least two atomic writes. In order to keep the size of the state vector small and thereby being able to explore more states, we use *short* (16 bit) integers by default in our generated models. This is sufficient in most cases, since high integer values are rarely used by programs that are explored in a model checker.

Instead of modeling complex garbage collection and memory allocation mechanisms, we decided to allow processes to allocate memory, but never release it. Our models capture this by using a counter *memUse* (initially $memUse = 1$) that points to the next free cell of the *memory*. The memory allocation (lines 29-37) then simply returns value of *memUse*, if a pointer to a free memory cell is required. At the same time, the counter *memUse* is incremented by the amount of free memory cells that were requested. An assertion ensures that an error is generated by the model checker, if the chosen *memory* length is not sufficient.

Objects, structs and arrays require pointer computations in order to access an attribute of an object (resp. struct) or an element at a particular index of an array. In LLVM IR, the *getelementptr* instruction performs these computations locally. All of the data types are represented by consecutive cells in the memory. A pointer to the object (resp. struct or array) always points to the first cell of the object (resp. struct or array). Thus, an attribute or index can be accessed by computing an offset relative to the pointer and thereby getting a new pointer pointing exactly to the requested memory cell. Our generated models simulate this computation in an inline statement *getelementptr*. An assertion (line 12) checks whether the accessed cell is still in bounds of the object (resp. struct or array). This check is only possible because LLVM IR contains type information. Our transformation uses the type information in order to derive the size of an object in terms of the number of memory cells it requires.

In order to avoid clutter in the program behavior, the model also contains an inline statement for a compare-and-swap (CAS) instruction. Another reason for the separate definition is that the result of a CAS in LLVM IR is in fact a tuple, which contains the read value (obtained from *memory[adr]*) and a success bit. In most of the LLVM IR code we have seen, the result was used as if it was just the read value. However, sometimes the CAS result is used as if it is only the success bit. By outsourcing the CAS definition to an inline statement, it can be adapted at a single point in the code.

Test Scenarios

Before the state space of a program can be explored, we need to fix the scenario that is to be explored. A scenario defines the number of concurrent processes, their behavior and an initial state. Processes in Promela are defined as *proctype*s. Weak2SC generates empty *proctype* stubs. These can be used in order to define particular scenarios by filling them with calls of the inline statements that we generate for each store buffer graph (resp. method). An initial state is defined in an *init* statement. The generated *init* statement allocates memory for all variables that are globally defined in a program. In addition, it starts both previously defined processes. Of course, more processes can be defined and added in the same way. The reader may notice that the variable *deq* should be a pointer to an array and thus should allocate more memory cells than only one. Given the LLVM IR code in Figure 4.3, there is no simple way of differentiating this from a regular pointer to an integer and thus, the allocated amount must be checked manually afterwards. In fact, our implementation allocated memory for the *deq* array in the main function, which we removed before transformation as it is not part of the actual queue implementation. Weak2SC does not aim at defining complete scenarios, but tries to help with providing definition templates which reduce manual effort. However, some manual effort has to be spent in order to define a scenario.

Program Behavior and Variable Declaration

The second part of our generated Promela models defines the program behavior in terms of inline statements. It also declares variables that are declared globally in the LLVM IR code of a program. Figure 4.7 shows an excerpt of the generated Promela model of the Arora et al. queue. In particular, it contains global variable declarations and the inline statement that was generated for the *pushBottom* operation and its respective store buffer graph under TSO (see Figure 4.4). For the global variables *bot*, *deq* and *age*, memory space is allocated in the init statement as previously mentioned. Each store buffer graph is represented by an inline statement. Each inline statement has a number of parameters, which coincide with parameters of the underlying method of the store buffer graph. Since inline statements in Promela do not return results, we add a *result* parameter if the underlying method returns a result value. We also need to ensure that the *result* variable is assigned the returned value before the control flow of the inline statement is left.

In LLVM IR local variables are declared by their use. In contrast, Promela requires explicit variable declarations. Thus, inline statements for a store buffer graph declare local variables first. The local variables are derived from the original program by collecting all variables that are used in a method and which are not declared globally.

```
1  short bot = null;
2  short deq = null;
3  short age = null;
4
5  inline pushBottom(elem){
6  short v0, v1, v2, idx, inc;
7  AStart: goto A00;
8  A00: v0 = memory[bot]; goto A01;
9  A01: v1 = memory[v0]; goto A02;
10 A02: v2 = memory[deq]; goto A03;
11 A03: getelementptr(1, v2, v1, idx); goto A04;
12 A04: goto A05idx;
13 A05idx:
14   if
15   :: inc = v1 + 1; goto A06idx;
16   :: memory[idx] = elem; goto A05;
17   fi;
18 A06idx:
19   if
20   :: goto A07idxv0;
21   :: memory[idx] = elem; goto A06;
22   fi;
23 A05: inc = v1 + 1; goto A06;
24 A07idxv0: memory[idx] = elem; goto A07v0;
25 A06: goto A07v0;
26 A07v0: memory[v0] = inc; goto A07;
27 A07: goto AEnd;
28
29 AEnd: skip;
30 }
```

Figure 4.7: Promela model generated for the TSO store buffer graph for the method *pushBottom* in Figure 4.3.

This collection is then of course extended by the variables that we introduced due to WDCs. All variables are treated as integer variables. This also includes boolean variables, where we treat 0 as false and all other values as true.

After local variable declaration, the program behavior is defined. All nodes of a store buffer graph have a unique label. For each label (corresponding to a node), the possible outgoing transitions are defined. Because it is a graph encoding, we cannot rely on the order of the statements in the generated Promela model. Thus, each transition is followed by a *goto* statement that leads to the next label corresponding to the target node in the store buffer graph. In principle, all transition have the form *source label: transitions effect; goto target label;*. The effect of each transition is encoded straightforwardly. Access to *memory* is atomic as it is assumed for all SC programs. See line 8 for a read of the shared variable *bot* from memory and line 16 for a write to the memory location represented by variable *idx*.

For nodes with several outgoing edges, the choice of the next transition can be deterministic, non-deterministic or mix of both. Deterministic choices are introduced by conditional branches. A non-deterministic choice to flush store buffer content is added to a node, if its store buffer is non-empty. A mix of both conditions can occur, if a branch is conditional and the store buffer is non-empty. In Promela, we model all of the above mentioned cases as an *if* statement. An *if* in Promela is a non-deterministic choice between statements that are not blocking. A blocking statement in Promela is an expression that evaluates to false. Thus, for conditional branches we can simply use the branching conditions to make the *if* statement deterministic (see *CAS* definition in Figure 4.6 for an example). Non-deterministic choices are achieved by simply stating no condition or one that always evaluates to true. The code excerpt in Figure 4.7 uses the former variant at labels *A05idx* (line 13) and *A06idx* (line 18). We can also mix both types of choices. Thereby, we can combine choices that are always enabled with choices that are enabled conditionally.

Each inline statement corresponding to a store buffer graph has two additional labels *AStart* and *AEnd*, for which we ensure that they are at the beginning (resp. end) of the inline statement. At the label *AStart*, the program simply jumps to the label corresponding to the first node of the store buffer graph. The label is necessary, because the order in which store buffer graph nodes are processed by Acceleo during transformation varies. Thus, it does not always begin with the first node. The label *AEnd* is required, because programs can have several return statements. Inline statements in Promela do not have a return statement as they define only a piece of control flow. Thus, we add a label *AEnd*, which marks the control flow end of the inline statement. The label can be targeted by transitions corresponding to return statements.

All labels are unique, but differ from what we used in Chapter 3.2. We use a label prefix starting with "A" for the first method, "B" for the second and so on. Besides this, labels are numbered identically to the numbering in the store buffer graph. A label suffix contains variable names if the corresponding node in the store buffer graph contains an entry. In most cases, using the variables that represent the memory locations of the store buffer entries is sufficient in order to distinguish labels uniquely. However, in some cases the entries differ only by their values. In these cases, we add the values to the suffixes. Thus, labels are kept short where it is sufficient and extended where it is necessary.

This concludes our store buffer graph encoding in Promela. In the following Section, we introduce its encoding in the theorem prover KIV. In Section 4.4.3, we also explain how Weak2SC can be used to generate program models in Promela based on an operational memory model as proposed in [TMW13].

4.4.2 Generating KIV Program Encoding

KIV [EPS+14] is an interactive theorem prover. The reason why we chose it as the tool for proving an algorithm correct is its built-in proof assistance. It allows for proof automation by implementing different proof heuristics, e.g., automated quantifier instantiations or case splits. Furthermore, it has a powerful proof visualization that helps with understanding proof goals and why they sometimes or rather the majority of the time fail to close. KIV is distributed with a large library that contains various data structures, fully formalized by axioms and additional lemmas that help with developing user specific formalizations and proof obligations.

KIV supports *Higher-Order Logic* and *Dynamic Logic*. The proof calculus in KIV is sequent calculus. In sequent calculus, a lemma is simplified and split into different lemmas based on a set of rules until an axiom is reached and thus, we know the lemma is correct. A lemma can be a proof goal that is split into minor proof goals. If all minor proof goals are correct, then the original proof goal is also correct. Because of this, proofs have a tree-like structure, which is where the proof visualization in KIV provides a lot of useful information, e.g., all proof steps, closed and remaining proof goals.

Our focus lies on verification of concurrent data structure implementations, where linearizability [HW90] is a quasi standard correctness criterion. Proof obligations for linearizability have been developed and published in [DSW11a, DSW11b, SWD12]. The theory behind the proof obligations for linearizability was formalized and proven sound in a KIV project. The latter is available online[9] and can be used for verification of other case studies. Our transformation to program encodings in KIV targets specifically these proof obligations, but is not limited to them. Other properties can also be verified as our transformation provides the program behavior only, and not the correctness condition. Chapter 5 and Chapter 6 provide more insights into the actual linearizability theory. For now, we will focus only on the ingredients that must be generated, in order to encode a store buffer graph into an SC program in KIV. The KIV input is divided into two major parts, the definition of program state and the encoding of all possible program steps, which will be the concrete operations. In the following, we first introduce the program state encoding and outline the encoding of concrete operations later.

Program State Representation

Similar to the generated Promela models, we define the shared memory first. It is a function that maps memory locations to values. For simplicity, we use the positive natural numbers $\mathbb{N}$ as the domain of memory locations. Instead of defining

[9] http://swt.informatik.uni-augsburg.de/swt/projects/

a special number value as being *null*, we allow values taken from the memory to be *null*. The set of all values ref is either $null \cup \mathbb{N}$ or $null \cup \mathbb{Z}$. Weak2SC allows to choose which of the ref variants shall be taken for the KIV program encoding. Using positive natural numbers is usually more practical, but sometimes not sufficient, i.e., when a program requires negative numbers. By defining the memory function as $memory : \mathbb{N} \to ref$, any value taken from memory can be *null*. However, by enabling memory access only for $\mathbb{N}$ values, proofs have to establish that the value is not negative and not a *null* value. Otherwise, the access is undefined and the proof cannot succeed.

Weak2SC provides two variants of program state representation: a global state and a local state representation. While the global state representation is the general form, the local state representation can be more practical as it can be used for proofs, where the correctness arguments are local to each process. The benefit of local proof arguments comes from usually simpler properties that have to be considered, because they are stated per process in contrast to properties involving all processes in a global state representation.

data specification
using PC, genProc, natref-memory
$CS := mkcs($ . .$mem : memory;$
. .$pc : Proc \to PC;$
. .$elem : Proc \to nat;$
. .$inc : Proc \to nat;$
. .$idx : Proc \to ref;$
. .$cmp : Proc \to nat;$
. . .
$);$
variables
$cs, cs' : CS;$
$pcf, pcf' : Proc \to PC;$
$natf, natf' : Proc \to nat;$
$reff, reff' : Proc \to ref;$
end data specification

data specification
using natref, PC, Proc
$Localstate := mkls($ . .$pid : Proc;$
. .$pc : PC;$
. .$elem : nat;$
. .$inc : nat;$
. .$idx : ref;$
. .$cmp : nat;$
. .$add : nat;$
. .$retval\_0 : nat;$
. .$v0 : ref;$
. . .
$);$
variables
$ls, ls' : Localstate;$
end data specification

Figure 4.8: Excerpt of generated global (left) and local (right) state definition for Arora et al. work stealing queue[ABP98]

In Figure 4.8, we show an excerpt of both variants of state definitions that Weak2SC can generate for our case study. Both define the respective state as a tuple. CS models the global state. Thus, it includes the shared state (the memory) while for

each local variable, it contains a function that maps from a process identifier $Proc$ to a value. Each tuple entry has its own access function that applied to a tuple yields the respective value, e.g., $cs.mem$ yields the current $memory$ function value. Local variables are accessed via a parameter $p \in Proc$, e.g., $cs.pc(p)$ yields the program location value of process p. The set of all program location is PC, which is generated separately and we refrain from showing its specification as it just defines a set of disjoint label values. Local states $Localstate$ are defined similarly to the global states, but they model only state of one process. The entry $ls.pid \in Proc$ models the owner process of a local state $ls \in LS$. As each local state has its owner process, accesses to local variables do not need to provide the owner explicitly as a parameter, e.g., $ls.pc$ yields the program location of the owner process of ls. For the local state encoding, a separate shared state is required but not generated. The $memory$ definition is available as part of our library[10] and it only needs to be instantiated as the shared state of the program. Please note that the value of the global variables bot, deq and age must be represented by an entry in the memory. Thus, we define bot, deq and age as constants in a later part of our generated specifications. The constant value can then be used in order to access the respective memory entry, e.g., $mem[bot]$ yields the shared value of bot where mem is a variable representing the memory function.

As the reader may have noticed by now, some of the variables are represented by nat[11] and others by ref values. The reason for this is that some of the variables are used for memory access or are the result of it, e.g., a pointer. Thus, they could be undefined and have the value $null$. An invariant usually has to rule out that case if it is necessary for the proof. However, other variables are the result of an arithmetic operation, for which we can safely assume that they are a numerical values. Weak2SC determines the type of each variable based on its use in an instruction and the provided type annotations in LLVM. If a variable is used in different methods and the type check for the variables results in ref for at least one of its uses, then its type becomes ref and nat, otherwise. We do not introduce new variables in such a case, because the program is already in a low-level representation at this point and we want to keep the encoding of it as concise as possible.

Program Behavior

The program behavior is defined in a separate specification module. Each atomic step of the program is represented by a predicate that encodes its effect. We also refer to these steps as concrete operations, since in our later proofs we will also have

[10] All our KIV projects and libraries are available on the DVD attached to this thesis.

[11] nat (resp. int) is the KIV specification for positive (resp. and negative) natural numbers.

enrich natref-memory, localstate, cindex with
constants
$bot : ref;$
$deq : ref;$
$age : ref;$
functions
$INVOP : IJ \to nat \times memory \times Localstate \times memory \times Localstate \to bool;$
$COP : CJ \to memory \times Localstate \times memory \times Localstate \to bool;$
$RETOP : RJ \to memory \times Localstate \times memory \times Localstate \times nat \to bool;$
predicates
$LSInit : Localstate \times Proc;$
$GSInit : memory;$
$\ldots$

Figure 4.9: Declaration of constants, functions and predicates for the encoding of the store buffer graph as a transition system.

to deal with abstract operations. The latter usually encodes the effect of a method as a single operation in contrast to multiple concrete operations implementing it.

We differentiate between concrete operations corresponding to method invocation, method return and all other steps of a method implementation. Many correctness criteria define correct behavior abstractly in terms of histories of invokes and responses. Encoding these steps of a program as a special kind of concrete operation simplifies later proofs, because it makes additional predicates obsolete that would otherwise define whether a step is an invocation (resp. a response) or not. Figure 4.9 shows the definition of constants, functions and predicates, which are essential for the definition of the operation encoding. The constants correspond to the global variable definitions in the LLVM IR code. They can be thought of as constant identifiers for memory locations, e.g., to the array pointed by variable deq in the algorithm. The constants are followed by three function declarations $INVOP$, COP and $RETOP$, one for each type of concrete operation (invoke, return and all other). Each of the functions maps labels (IJ, CJ, RJ) to a predicate encoding the respective operation. The predicate defines the precondition and effect of an operation. Invoke (resp. response) operations have an additional parameter for the input (resp. output) of a method. The encoding in Figure 4.9 is local. Thus, the predicates are defined over two pairs of shared state and a local state, ($memory \times Localstate$). One pair represents the program state before and one after the effect of a concrete operation. For the global encoding, we would replace $memory \times Localstate$ by CS.

In addition, predicates for the definition of initial states are declared ($LSInit$ and $GSInit$ for local encoding and a $CSInit$ for the global encoding).

axioms
$LSInit$:
$\models LSInit(ls, p) \Leftrightarrow ls.pc = N \wedge ls.pid = p;$

;; pushBottom corresponds to method @pushBottom
pushBottomini :
$\models INVOP(pushBottomini)(inp, mem, ls, mem', ls') \Leftrightarrow$
$ls.pc = N \wedge ls' = (ls.pc := A00\ .elem := inp) \wedge mem' = mem;$

*;; %0 = load i32** @bot, align 4, !tbaa !0*
pushBottom1 :
$\models COP(pushBottom1)(mem, ls, mem', ls') \Leftrightarrow$
$ls.pc = A00 \wedge ls' = (ls.pc := A01\ .v0 := mem[bot.v]) \wedge mem' = mem;$

;; FlushTransition
pushBottom9 :
$\models COP(pushBottom9)(mem, ls, mem', ls') \Leftrightarrow$
$ls.pc = A06idx \wedge ls' = ls.pc := A06 \wedge mem' = mem[ls.idx.v, \ulcorner ls.elem \urcorner];$

;; %inc = add i32 %1, 1
pushBottom10 :
$\models COP(pushBottom10)(mem, ls, mem', ls') \Leftrightarrow$
$ls.pc = A05 \wedge ls' = (ls.pc := A06\ .inc := (ls.v1 + 1)) \wedge mem' = mem;$

;; ret void
pushBottom14ret :
$\models RETOP(pushBottom14ret)(mem, ls, mem', ls', return) \Leftrightarrow$
$ls.pc = A07 \wedge ls' = ls.pc := N \wedge mem' = mem;$
...
end enrich

Figure 4.10: Excerpt of generated transition system using local state encoding for Arora et al. work stealing queue[ABP98]

In order to properly define the program behavior, axioms must be defined. Figure 4.10 shows an excerpt of the axioms that Weak2SC generates for the work-stealing queue case study. In fact, Figure 4.9 and Figure 4.10 are excerpts of the same

output file. The initialization predicate $LSInit$ binds a process p to a local state by $ls.pid = p$ and defines that a process is initially idle $ls.pc = N$, where N is the program location of idle processes.

The remaining axioms define concrete operations. The figure depicts exemplary an invoke, a return operation and other typical concrete operations such as a read, a flush and a local computation. Each operation is defined in terms of a comment, an axiom name, a sequent definition and a statement of whether it should be used as a simplifier rule by the theorem prover (omitted in the figure). Comments are generated in order to help with orientation in the sometimes lengthy output. The comments name the LLVM IR instruction corresponding to the concrete operation or if there is none (e.g., if it is a flush), its corresponding edge in the store buffer graph. Invoke operations mention the original name of a method in their comment, because WEAK2SC allows method names to be changed and the generated labels use the chosen method name as a label prefix.

Throughout the thesis, we will use unprimed variables as representative for the state before the effect of an operation and primed variables as representatives of the state after. In the figure, an invoke operation is defined for the *pushBottom* method. The operation named as $pushBottomini$ defines that a process must be idle ($ls.pc = N$) before invocation of *pushBottom*. The pc value of the local state is updated to $A00$ and the parameter inp is assigned to the local variable $elem$ by the expression $(ls' = ls.pc := A00\ .elem : inp)$. The latter expression is a short notation for stating that all values of the tuple ls' are equal to the values in ls, except for the updated ones. The predicate part $mem' = mem$ states that the memory is not modified by the operation.

The operation $pushBottom1$ is neither an invoke nor a return operation and therefore it is defined as a COP operation. As the comment states, it encodes the semantics of a load/read edge from the store buffer graph. It updates the program location pc and since it is a read, it assigns the value taken from the memory mem at location $bot.v$ to the local variable $ls.v0$. As the type of bot is ref, it can be either $null$ or a natural number. A natural number v can be lifted to a ref by $\ulcorner v \urcorner$. The natural number of a ref value r can be accessed by $r.v$. However, in a proof one would need to establish that r is not $null$. Otherwise, the access is undefined.

The operation $pushBottom9$ corresponds to a flush edge, which according to our transformation becomes a write in our SC program. Thus, it updates the memory mem. The statement $mem' = mem[ls.idx.v, \ulcorner ls.elem \urcorner]$ defines mem' to be equivalent to mem, except for the location $ls.idx.v$, which is updated to the new value $\ulcorner ls.elem \urcorner]$. Operation $pushBottom10$ is an addition and thus, a simple local operation as it modifies the local state only. It assigns the value $ls.v1 + 1$ to the local variable $ls.inc$. Since, the variable $ls.v1$ has the type nat, it does not have to

be lifted.

The remaining operation $pushBottom14ret$ encodes a return edge. Since the method *pushBottom* does not return a value, the operation only updates the program location value $ls.pc$ to the idle value N. Otherwise, the predicate would also state a property for the variable $return$ that is part of predicate signature.

The complete encoding for the queue implementation together with its linearizability proofs can be found on the DVD attached to this thesis.

4.4.3 Promela Programs for Operational Memory Models

Besides the implementation of our reduction approach from Chapter 3, Weak2SC also provides a generation of Promela program models based on operational memory models. The intention behind the implementation and automation of this type of program model generation was to provide a fall back solution for programs that do not adhere to our assumptions 1 and 2 (see Ch. 3.1). In our later experiments (see Ch. 6), we compare our reduction approach against the more common approach of using an operational memory model for verification. An operational memory model is essentially a semantics definition that is operational. It defines the semantics in terms of the behavior of its parts. In our case, this is the program control flow and the store buffer. While the behavior of the control flow is straightforward, the behavior of the store buffer is the interesting part as the operational memory model must mimic all steps of an actual store buffer. Operational memory models have been used for quite a while in the research of weak memory behavior [PD95]. However, to the best of our knowledge, we were the first to implement an operational memory model for TSO in Promela and the model checker Spin [TMW13]. Later, we extended our work with an operational memory model for PSO and an automated program model generation, which has become part of Weak2SC and was published in [TW16]. In the following, we will introduce this type of generated models.

The Operational Memory Model

The underlying approach separates the program model from the operational memory model. Figure 4.11 shows the idea behind the separation. Each process in a concurrent setup is represented by a pair of processes. The first one is the program process. It models the program control flow or simply the sequence of program instructions. The second process implements the semantics of instructions in Promela according to the memory model (SC, TSO or PSO). When a program process issues an instruction that needs access to the memory, e.g., a read, then the request is forwarded via handshake communication to the operational memory model (OMM) process. It is the latter process that contains a write buffer (in case of TSO or PSO) and which

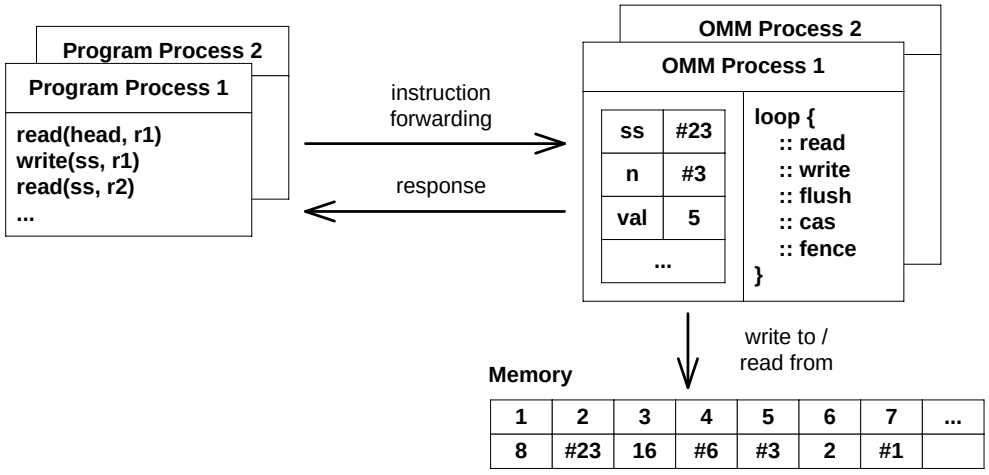

Figure 4.11: Promela programs based on operational memory models as proposed in [TMW13]

simulates the possible effects of a particular memory model, e.g., the delay of a write due to the later appearance of a flush. The OMM process essentially manages all access to the memory and therefore can simulate atomic access in case of SC or the weak memory effects. Since the OMM process is a separate process, the non-determinism of a weak memory model is captured implicitly in the possible interleavings of all processes.

The OMM processes for TSO and PSO are modelled by a loop over a non-deterministic choice between five possible events. Four out of the five events are the handshake communications with the program process. Thus, they are only enabled if the program process issues such an event. The corresponding events are a program process issuing a read, a write, a CAS or a fence. Figure 4.12 shows the communication between a program process and an OMM process in case of these events, but also how a request is executed by the OMM process. The fifth event is a flush and it does not require handshake communication. Instead, it is enabled whenever the store buffer is not empty. An OMM process for TSO and PSO has an array that models the store buffer. Thus, if a program process issues a write, then the OMM process takes it via handshake communication and stores it into its store buffer. This also enables flushing it, either immediately after or at a later point in time. When a program process issues a read, then the OMM process looks up for the requested value in its store buffer. If it is present, then value from the store buffer is given in return to the program process, so that it can proceed with its computation. Otherwise, the value is taken from the memory. A read issued by

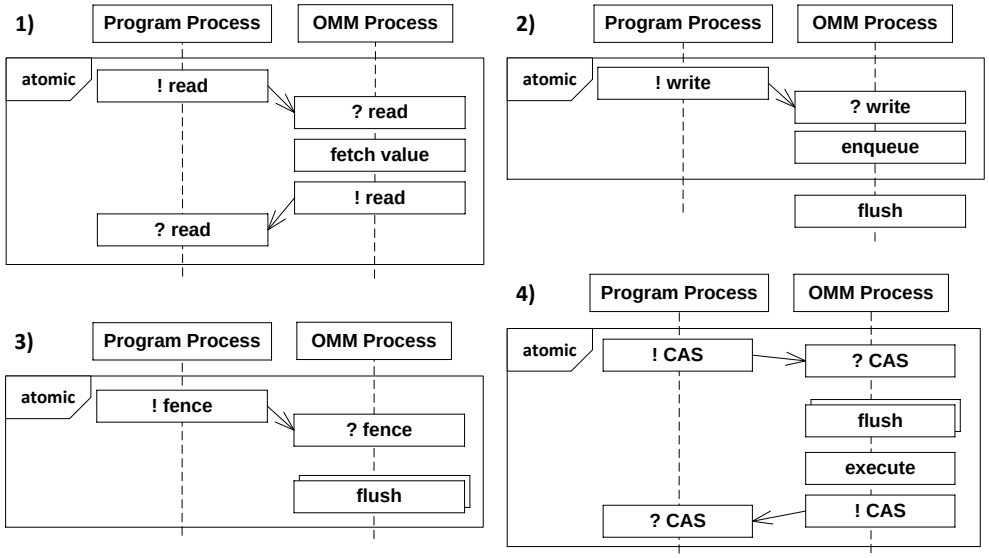

Figure 4.12: Communication between program process and OMM process; 1. read 2. write 3. fence and 4. CAS

the program process is blocking it until it gets a response from the OMM process. This way, a read remains atomic, even though two processes are involved. An issued fence causes the OMM process to empty its store buffer, which it does by flushing its entries all at once. Performing all flushes at once is safe in this case, even though a fence does not guarantee that all entries are flushed atomically. Stepwise flushing of store buffer entries is already covered by regular non-deterministic flushes. A CAS operates directly on the memory, but before it can be performed, the store buffer must be emptied. The latter is achieved similarly to fence.

For SC, we do not need a separate OMM process. Instead, all requests are instantly forwarded to the memory and performed atomically, because Promela and Spin provide already SC semantics.

As mentioned earlier, the operational memory models have been developed earlier [TMW13, TW16]. However, they are also part of the output that Weak2SC generates for a program, when an operational encoding is requested. All we need now is to generate program processes that issue the handshake communication to the OMM processes, whenever they would otherwise attempt to access and perhaps modify the memory. The latter is achieved by simple inline statements, which issue the communication for read, write, fence and CAS as depicted in Figure 4.12. They are also used by the generated programs in our program processes. We provide them

as part of operational memory encoding, because they differ for SC, which does not need OMM processes.

Generating Program Processes

```
1  #define MEM_SIZE 10    //memory size
2  #define BUFF_SIZE 3    //buffer size
3  #define null 0
4  short memUse = 1;      //next free memory cell
5  //#include "sc.pml"
6  #include "tso.pml"
7  //#include "pso.pml"
8
9  chan channelT1 = [0] of {mtype, short, short, short};
10 chan channelT2 = [0] of {mtype, short, short, short};
11
12 short bot = null;
13 short deq = null;
14 short age = null;
15
16 inline pushBottom(elem){
17 short v0, v1, v2, idx, inc;
18 skip;
19 entry:
20  read(bot, v0);
21  read(v0, v1);
22  read(deq, v2);
23  getelementptr(1, v2, v1, idx);
24  write(idx, elem);
25  inc = v1 + 1;
26  write(v0, inc);
27  goto ret;
28
29 ret: skip;
30 }
```

Figure 4.13: Generated program model based on operational memory models of *pushBottom* method.

For the sake of completeness, we show an excerpt of the generated program model for the Arora et al. queue in Figure 4.13. The memory is represented as a fixed but adjustable size array, similar to our previously introduced Promela models in Section 4.4.1. The same holds for the write buffer that is carried by the OMM processes. We have to fix these sizes, because Promela does not support dynamic data structures. Memory allocation and pointer computations also work in the same way as previously introduced. The program representation is significantly shorter as it does not represent non-determinism due to weak memory semantics explicitly.

The latter is hidden in a separate operational memory model, which is simulated by OMM processes. In summary, the program model is close to the original LLVM IR code as the control flow blocks and statements remain in program order and even the block labels correspond to those in the original code (see Fig. 4.3 on p. 73). The only exception to the block labels is the return statement, which is modeled as a separate labeled block (label *ret*). It is the last program location within the inline statement and the target jump location of statements corresponding to return instructions (particularly those which are not the last program statement of a method).

For all memory accessing instructions, inline statements are defined, which perform the handshake communication with the respective OMM process. Each pair of processes gets its own channel, which is unbuffered (synchronized communication). Whether SC, TSO or PSO semantics are used is determined by the include statement in lines 5 to 7. The OMM process definitions are part of the generated output for a given program and can be exchanged one for another with minor model adjustments. We refrain from showing them here and refer to our publication [TMW13], where we initially presented the approach. The complete operational memory models encoded in Promela can also be found in our Github repository[Tra16] or the DVD attached to this thesis, where they are part of generated program models for various case studies.

4.5 Discussion and Possible Future Extensions

From the beginning of the implementation, it was clear that Weak2SC would be a tool that we would extend incrementally with more and more features. Thus, we focused on following the OMG standards for model-driven software development [Obj15b, Obj15a, Obj08], which, besides other objectives, aim for minimizing effort in software maintenance and future extensions. With an ever growing list of implemented features, there is also an ever growing list of possible extensions. In the following, we want to discuss the ones we think would be the most promising extensions.

Alias Analysis As already pointed out in Chapter 3.3.3, our implementation of the reduction approach in Weak2SC lacks a pointer alias analysis. We fully rely on the pointer alias detection by the LLVM compiler framework. However, pointer variables do not necessarily represent disjoint memory locations. In Weak2SC, we treat them as if they were disjoint, e.g., when we detect early reads. As a consequence, a program generated by Weak2SC based on a store buffer graph could miss possible behavior due to pointer aliases. Therefore, we suggest an extension towards consideration of potential pointer aliases in future work in order to improve the quality of the results achieved with Weak2SC. In particular, such an extension would determine whether two pointer variables can alias each other and consider this information in the store

buffer graph construction, e.g., by distinguishing the case in which both variables represent the same memory location from the case where they do not.

Target Languages and Tools So far, Weak2SC produces output for two verification tools, which have no built-in support for weak memory models. Thus, one direction for future extensions is obviously the support for more verification tools in order to widen the range of possible choices by users of Weak2SC. We expect the potential users of Weak2SC to be verification experts, but even experts are usually experts in few verification tools. Thus, it is important to offer input to tools they are familiar with. For model checking concurrent software, the tools Z3 [dMB08] and NuSMV [CCG+02] would be our first choice, as the former is an SMT solver and the latter a symbolic model checker. Both would contrast the explicit state model checker Spin. In terms of theorem proving, Isabelle [NPW02] and Coq [BC04] are widely used and therefore would be good candidates, even though they offer a less extensive graphical user interface compared to KIV [EPS+14].

Invariant Checking The overall approach proposed in this thesis for verification of concurrent data structure implementation still involves quite a bit of manual effort. We think, the model checking branch is well automated in terms of program model generation and checking a program property. Although Weak2SC also helps with generating an encoding of the program behavior for the theorem prover KIV, it does not automate the actual proof in any sense. A correctness proof is usually the most time consuming part of program verification, even with the help of a theorem prover. The difficulty lies in finding the correctness arguments like an invariant and particularly get them right. It is an iterative process of trying to prove program correctness. If the proof fails, the correctness arguments must be revised and the proof attempted again until the proof either succeeds or an argument is found why the program is incorrect. Weak2SC generates already input to model checkers. So, it could be extended towards checking parts of the correctness arguments like the invariant that are required for a proof. An automated check could reduce effort spend on proof attempts with an insufficient invariant. At the time of writing, we plan and develop an extension of Weak2SC towards this direction in the near future.

Store Buffer Graph Representation We mentioned already that store buffer graphs can also help with understanding how an algorithm works and that this is crucial in the verification process. However, a store buffer graph can grow quickly with the size of the corresponding program, particularly, if it has only few fences that would limit possible reorderings. One idea, we have not followed yet, is to change the visual representation of store buffer graphs towards a hierarchical state representation. Sometimes parts of a store buffer graph (the control flow) repeat themselves combined with different store buffer states. Therefore, we suggest evaluation of a hierarchical store buffer graph representation for future work. It would

group nodes with the same store buffer state. Each such group node would again contain nodes and edges, essentially representing the control flow of the program that does not affect the store buffer state. Currently, every node with a non-empty symbolic store buffer has a separate outgoing flush transition to some other node (same program location, but flushed value removed from store buffer). With group nodes for each store buffer state, several flush transitions could be represented by a single edge between the group nodes. Edges corresponding to writes would mark the start of the control flow within such a group node. The proposed store buffer graph representation would avoid redundant flush transitions and help with structuring a graph into meaningful parts. Consequently, we think it has the potential to significantly reduce visual clutter for larger programs.

Fences and other Instructions Weak2SC supports only a subset of the possible LLVM IR instruction set. In particular, it supports what we found would be used most often in concurrent data structure implementations. Also, Weak2SC supports only full fences so far. For TSO this is fully sufficient. However, under PSO, there is also a weaker type of fence, which Weak2SC does not take into account, yet. The weak fence can prevent writes to be reordered with each other without imposing an order on reads. The weak fence does not block until the buffer is emptied, but it allows for a processor to continue its execution (including early reads) while ensuring that writes before the weak fence are flushed before the writes after the fence. For the sake of completeness, this type of fence should be added to the considered semantics of PSO in the future work. An extension of the store buffer graph construction should be straightforward in this case. Most of the other instructions that are not supported by Weak2SC are essentially multi-word instructions, i.e., the respective memory access (read or write) may involve several reads (resp. writes) per instruction. Although we think it would be desirable to support the complete instructions set of LLVM IR, we do not give it a high priority in our future work as software verification without multi-word semantics is challenging enough already.

The discussion concludes the chapter on the implementation of the proposed reduction from Chapter 3 in terms of the tool Weak2SC. In the following Chapter, we provide some background on concurrent correctness criteria, particularly on linearizability, and on the verification techniques used to check or prove it for a given program. In Chapter 6, we present our own experiments with verification of linearizability and explain how Weak2SC allowed us to perform them.

5 Correctness of Concurrent Data Structures

Throughout this thesis, we are ultimately interested in verification of concurrent data structures. For concurrent data structures, there are many correctness conditions [Pap79, HW90, HS08, SK09, HKP+13] that define when an implementation is considered correct. One of the oldest is *serializability* [Pap79]. It was also one of the first to capture the idea that concurrent behavior should be explainable by sequential behavior. In other words, for any concurrent execution of a program there should be an equivalent sequential execution. Many other correctness conditions evolved from serializability. Some correctness conditions relax serializability, e.g., *snapshot isolation* [BBG+95] for data bases, others strengthen it by tying it to a certain context and the definition of additional properties that have to hold, e.g., *opacity* [GK08] for transactional memory implementations [HM93, ST97]. All of them stick to the fundamental idea of defining correctness via equivalence to sequential behavior.

5.1 Linearizability

For concurrent data structures, *linearizability* [HW90] was developed by Herlihy and Wing and since then has become the quasi standard correctness condition. It is popular for particularly two reasons: first, it ties the definition of correctness of a concrete data type to an arbitrary abstract and sequential data type, e.g., an atomic stack specification for a stack implementation. Abstract data types represent the correct behavior of concrete data structures and are essentially missing in serializability [Pap79]. The latter requires a concurrent execution of a set of transactions to be equivalent to some sequential execution of the same set. Sequential executions of an abstract data type justify concurrent behavior, which also makes it an intuitive

correctness condition.

The second reason for the popularity of linearizability is the refinement guarantees that it provides. At some point between invocation and response, the *linearization point* (LP), an implementations appears to take effect just like its abstract specification. The refinement between a linearizable implementation and its abstract specification means that the implementation can be replaced by its abstract specification without changing the observable behavior. The latter is also possible, because linearizability is compositional [Wei89], i.e., objects that are composed out of linearizable objects will be linearizable again. Compositionality combined allows us to investigate and verify supposedly linearizable implementations separately from larger programs in which they are meant to be embedded. This also helps with the verification of programs embedding linearizable implementations, since the linearizable code parts can be replaced by abstract atomic (and thus usually simpler) specifications.

With the rise of weak memory models in modern multicore processors, it quickly became obvious that their is a semantic gap. Not only do most verification techniques assume sequential consistent (SC) memory models [Lam79], but also the original definition of linearizability [HW90]. Recently, several adaptations have been proposed. In this section, we would like to provide an overview of the original linearizability definition and how it was adapted to the setting of weak memory models. Our intention is to clarify the different meanings and consequences of using a particular linearizability definition. This also lays the ground for the next Chapter 6, where we present approaches for the verification of linearizability under weak memory models. A recent survey on different linearizability definitions was also given in [DD15].

5.1.1 Linearizability - Original Definition

All versions of linearizability are defined by relating histories of a concrete data type with histories of their abstract data type. Therefore, we first have to define what a history is before we can deal with the actual linearizability definition. This part is a brief rewriting of definitions in [HW90]. The definitions are modified slightly as parts of them were given informally in the original paper.

In [HW90], histories are defined as finite sequences of invoke and response events. Here, we use P for the set of all processes, M is set of all operations (or implemented methods). INP and OUT define the input and output domain.

$$E \mathrel{\widehat{=}} inv(P \times M \times INP) \mid ret(P \times M \times OUT)$$
$$H \mathrel{\widehat{=}} seq(E)$$

Each event $e \in E$ is issued by a process $e.p \in P$, corresponds to an operation of the data type $e.m \in M$ and has input values $e.i \in INP$ in case of an invoke and output values in case of a response $e.o \in OUT$. Furthermore, we use predicates $inv?(e)$ (resp. $ret?(e)$) which are true iff e is an invoke event (resp. a return event). A history $h \in H$ is of the form $e_1 \frown e_2 \frown \ldots \frown e_n$, if $n = \#h$ is the length of h. Furthermore, the events are totally ordered by the order of their appearance in h, i.e., $\forall\, i, j \in \mathbb{N} \bullet i < j \Rightarrow e_i < e_j$.

Additional terminology from [HW90] defines when events match and when an invocation is pending. A response event e_j is said to match an invoke event e_i, $match(e_i, e_j)$, if $inv?(e_i) \wedge ret?(e_j) \wedge e_j.p = e_i.p \wedge e_j.m = e_i.m$. An invocation e_i is considered $pending(e_i)$ if no matching response follows the invocation in the history. [HW90] defines a history h to be sequential, $sequential(h)$, if

1. the first event in h is an invoke event
   $\exists\, e \in E \bullet \mathit{first}(h) = e \wedge inv?(e)$, and

2. every invoke is immediately followed by its matching response. Only the last invoke event is allowed to not have an immediately following matching response. Responses are followed by invokes.
   $\forall\, n \in \mathbb{N} \bullet 0 < n < \#h - 1 \wedge inv?(e_{n-1}) \Rightarrow ret?(e_n)$

A history that is not sequential is concurrent. Also, since linearizability relates different histories, we need a notion of equivalence. In [HW90], this is defined via subhistories. A subhistory $h|_p$ of h is the sequence of events that remains, if we remove all events that do not belong to process p. Two histories h, h' are equivalent if the subhistories of all processes are identical, i.e., $\forall\, p \in P \bullet h|_p = h'|_p$. Furthermore, a history is defined to be *well-formed* if the subhistories of all processes are sequential, i.e.,

$$wf(h) \mathrel{\widehat{=}} \forall\, p \in P \bullet sequential(h|_p)$$

Herlihy and Wing assume all histories to be well-formed. The latter includes particularly concurrent histories, which is important, because this assumption does not necessarily hold under weak memory models.

A history h also induces an irreflexive partial order $<_h$ on operations, which is better known as real-time order:

$$\forall\, i, j \in \mathbb{N} \bullet ret?(e_i) < inv?(e_j) \Rightarrow e_i <_h e_j$$

which is the order between each response and its following invoke events. The above definitions orders operations according to their real-time occurrence. Operations that overlap in h are not ordered by $<_h$.

A history h is $legal(h)$ if it belongs to a sequential specification of a data structure. The idea behind legal is that it captures the semantics of a data structure, e.g., making it illegal to pop an element from an empty stack. In addition, all legal histories are sequential. Abstract atomic data types always provide legal executions and thus are a simple representation of the correct behavior of a data structure. The original definition is informal and we leave it as informal as it is.

Linearizability relates concurrent histories with sequential histories. Concurrent histories can have multiple operations executing at the same time without having reached their response, i.e., such histories have several invokes without matching response. These histories do not have a direct sequential counterpart, but must be completed before compared to a sequential history. A history h' is created from h, which can be compared to sequential histories instead of h. Essentially h' results from removing invoke events of operations, which have not taken effect yet (not reached their linearization point yet) and appending response events for those operations which have taken effect. The function $complete(h)$ does the former. The latter can be achieved by appending a sequence of responses. With this in mind, the definition of linearizability [HW90] by Herlihy and Wing is as follows:

Definition 17. *A history h is linearizable if it can be extended (by appending zero or more response events for pending invokes) to some history h' such that:*

L1 *$complete(h')$ is equivalent to some legal sequential history s and*

L2 *$<_h \subseteq <_s$.*

Furthermore, a linearizable data structure (or object originally) is one whose concurrent histories are linearizable with respect to some sequential specification. In other words, linearizable data structures behave as if their operations were atomic and thus like a sequential data structure.

5.1.2 Adaptations to Weak Memory Models

As already mentioned, the original linearizability definition assumes a sequential consistent memory model and thus, there is a gap between its assumptions and the semantics that we find on our latest multicore processors. Under SC, the execution of an operation begins with its invocation and ends with its response. The effect of an operation must become globally visible within the interval between invoke and response. Although both events are also present under weak memory models, they do not necessarily represent the begin and end of an operation, because parts of it can be delayed. Thus, the interval in which the effect of an operation must become visible is up to interpretation. Depending on the choice of the execution interval we

end up with different consequences for practicality and the guarantees provided by an adapted linearizability definition.

In this section, we want to discuss three different adaptations of linearizability towards weak memory models [BGMY12, GMY12, DSD14]. All of them were initially stated for the TSO memory model. However, so far, only the adaptation by Gotsman et al. [GMY12] was also extended to more relaxed memory models [BDG13]. All of the linearizability adaptations come with different foundations, particularly with their own weak memory semantics. The memory model is crucial, because it defines the set of possible executions or traces out of which the set of possible concurrent histories can be derived. When dealing with linearizability definitions, we are mostly dealing with histories, but we also have to understand what an event in a history tells us about the execution from which it was derived. We refrain from giving the complete semantics definitions (as used by the resp. definition), but instead provide an informal description of the history events. The latter should suffice to present the crucial differences for each of the adapted linearizability definitions.

TSO-to-TSO Linearizability

To to best of our knowledge, the first adaptation of linearizability towards weak memory models came from Burckhardt et al. [BGMY12]. It targets the TSO memory model. A major focus by Burckhardt et al. was to define linearizability as a refinement relation and thus, to be able to replace an implementation by its specification without a change in the observable behavior from a clients perspective. A client observes all interactions with the implementation via its interface, but can also experience delays, e.g., through results obtained by other processes.

Under TSO, method execution is not necessarily finished, when a client receives a response, because writes can be delayed due to store buffers. Thus, a later method may be executing already while writes of the previous operation are still waiting to be flushed to memory. Consecutive operations of a single process can overlap in their execution and thus, can be considered as internally concurrent in such a case. Overlapping execution within a process violates the well-formedness assumption by Herlihy and Wing, which assumes sequential execution for each process. The latter means that the linearization point of an operation must be somewhere between its invoke and response. The linearization point (LP) is a step at which the effect of an operations becomes visible to all processes. Under TSO, the LP of an operation can be delayed and thus can occur after its response.

The key question here is whether overlapping of operations within a process should be generally allowed for linearizable data structures or whether it should be considered incorrect. If allowed, consecutive operations can take effect in a different

order than their program order. The latter is possible under TSO, if the later operation linearizes on a read while the earlier operation linearizes with a write and both are reordered due store buffer delay. The adaptation by Burckhardt et al. [BGMY12] allows overlapping of operations within a process. However, the linearizability definition by Burckhardt et al. [BGMY12] allows this only if the abstract specification also allows it. The latter is only possible, if the abstract specification can examine TSO behavior itself. This is also the reason why Gotsman et al. [GMY12] called this version of linearizability *TSO-to-TSO linearizability* as it relates TSO histories to TSO histories.

A history in [BGMY12] consists of four events:

$$E_{tso2tso} \mathrel{\widehat{=}} inv(P \times M \times INP) \mid ret(P \times M \times OUT) \mid \mathit{flush}(inv) \mid \mathit{flush}(ret)$$
$$H_{tso2tso} \mathrel{\widehat{=}} seq(E_{tso2tso})$$

where the invoke and response events are identical to the previous definition. In addition, histories contain $\mathit{flush}(inv)$ and $\mathit{flush}(ret)$ events. By adding these events, a history represents two layers of execution: First, the interface to a client, who calls an operation (invoke) and obtains a result (response). Second, the delayed execution interval of writes to memory due to the store buffer, which are represented by the two flush events. These events mark the point in time at which the store buffer switches from flushing writes of one operation to the writes of the next operation. In the TSO semantics by Burckhardt et al. [BGMY12] invoke and response markers are added to the store buffer when an operation is invoked and when it returns. Later, these markers are flushed just like other writes in the store buffer, but without any other effect than being removed from the buffer. The moment when the markers are flushed is recorded as $\mathit{flush}(inv)$ and $\mathit{flush}(ret)$ events in a history. Similar, to $inv?$ and $ret?$, we will use $\mathit{finv}?$ and $\mathit{fret}?$ as a predicates for events which are true iff the event is a $\mathit{flush}(inv)$ (resp. $\mathit{flush}(ret)$).

Given the above type of histories, the TSO-to-TSO linearizability definition by Burckhardt et al. [BGMY12] is as follows:

Definition 18. *The **TSO-to-TSO linearizability relation** is a binary relation $\sqsubseteq$ on histories defined as follows: $h \sqsubseteq h'$ if $\forall\, p \in P \bullet h|_p = h'|_p$ and there is a bijection $\pi : \{1, \ldots, \#h\} \to \{1, \ldots, \#h'\}$ such that $\forall\, i \in h_i = h'_{\pi(i)}$ and*

$$\forall\, i, j \bullet (i < j) \land (ret?(h_i) \lor \mathit{fret}?(h_i)) \land (inv?(h'_j) \lor \mathit{finv}?(h'_j)) \Rightarrow \pi(i) < \pi(j)$$

As in the original definition, an implementation can be considered TSO-to-TSO linearizable if all its histories are TSO-to-TSO linearizable with respect to an

abstract specification. The crucial difference is that the abstract specification does not necessarily have to be atomic, but produces TSO histories itself.

Similar to the original definition, it relates histories by maintaining real-time order of certain events. In contrast to original definition, this definition deals with more different types of events and in particular maintains the order between responses or their flushes with later invokes or their flushes. If the flush events are ignored or if we use an atomic abstract specification which places the $\mathit{flush}(inv)$ and $\mathit{flush}(ret)$ events immediately after their respective invoke and response events, then TSO-to-TSO linearizability provides identical guarantees to the original definition of linearizability.

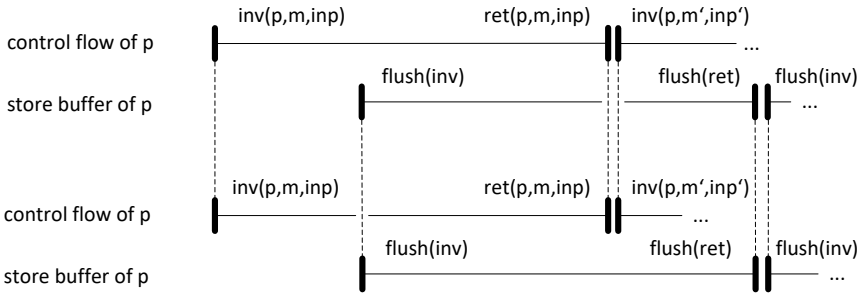

Figure 5.1: Visualizing TSO-to-TSO linearizability as a mapping of events from a concrete TSO history to an abstract TSO history.

What is new about TSO-to-TSO linearizability is that it defines a refinement relation between TSO histories which incorporate flush events. TSO-to-TSO linearizable data structures can be implementations at different granularity levels up to an atomic specification. However, if a concrete data structure can have its LP delayed after the response, then TSO-to-TSO linearizability requires this also to be possible in the abstract data structure. Otherwise, the concrete data structure is not TSO-to-TSO linearizable w.r.t. the abstract data structure. Figure 5.1 shall help with the visualization of the TSO-to-TSO linearizability relation. It shows two TSO histories of a single process p executing an operation m followed by m'. The dashed lines connect identical events in both histories. The overlapping execution of both operations due delayed flushes should be obvious. For both TSO histories, control flow and the delayed store buffer events are shown in separate time lines. The bottom history is supposed to refine the top history. The intervals formed by the dashed lines must be present in both histories. Their real-time order must be respected by a permutation. However, they do not have to be exactly above each other as is shown in the figure. Essentially, any operation has three fundamental intervals: 1. the over-

lap with a previous operation ($inv(\ldots) \ldots \mathit{flush}(inv)$), where it can linearize before the previous operation. 2. its core interval ($\mathit{flush}(inv) \ldots ret(\ldots)$), in which it does not overlap with any other operation of the same process. And 3. the overlap with a later operation $ret(\ldots) \ldots \mathit{flush}(ret)$, in which the later operation can linearize earlier as in (1.). Please note that these intervals can also be empty or shifted in way that there is no core interval, e.g., if the previous operation has pending writes and these remain in the store buffer until after the response of the current operation. The latter example also allows multiple consecutive operations of a process to overlap. The top history is a permutation of the events from bottom history and must respect the real-time order (not only for p but on all processes). Furthermore, the events of each process must not be reordered in a permutation, $\forall\, p \in P \bullet h|_p = h'|_p$. Just to clarify, that does not mean that the effect of an operation must appear in program order of a process. Instead, the effect can appear in any order that is legal w.r.t. the semantics of the abstract data type that provides the abstract TSO history.

Burckhardt et al. also prove an abstraction theorem (soundness), which states that a TSO-to-TSO linearizable implementation can be replaced by its abstract implementation, while assuming the *most general client*. The latter can be best viewed as arbitrary code calling methods of the data structure, but not knowing or using any variables that are also used by the data structure implementation internally.

TSO-to-TSO linearizability is probably the least restrictive adaptation of linearizability [HW90] to a weak memory model setting. However, it also imposes a burden for reasoning about correctness, because abstract data structures are not necessarily atomic. For linearizability, sequential histories are derived from usually simple data structures with atomic operations. The latter simplifies the definition of correct behavior. For non-atomic abstract data structures as in TSO-to-TSO linearizability, defining what is correct can be unintuitive and more importantly error prone due to the concurrency involved.

TSO-to-SC Linearizability

In the previous section, we posed an important question about what should be considered correct under weak memory models. The question is whether to allow data structure implementations to take effect outside of an operation's invoke-response interval. TSO-to-TSO linearizability [BGMY12] adapts linearizability to TSO memory models and allows this type of behavior. In this section, we will discuss TSO-to-SC linearizability [GMY12] by Gotsman et al. which prohibits this type of behavior.

As in our previous sections, we start with the histories. Under the TSO-to-SC case, no additional events like $\mathit{flush}(inv)$ or $\mathit{flush}(ret)$ are required, because the

definition in a sense ignores the delays of writes due to store buffers. Histories consist of invoke and response events E, just like in the original definition of linearizability. In contrast to the original definition, invoke and response events do not mark the beginning and end of an operation, but only its interface events, i.e., when an operation is called by a client and when the client receives a response. The latter does not necessarily mean that the corresponding operation is fully finished with its execution, since writes can still be pending in the store buffer during a response.

One of the reasons why the original linearizability definition [HW90] had to be adapted to weak memory models is that the above was not considered. Under weak memory models, there is no single point in time at which an operation begins execution and at which it ends execution, but at least two possible points for each case. For the beginning, it can be the invocation of the method or some later point tied to the store buffer, just like $\mathit{flush}(\mathit{inv})$ in TSO-to-TSO linearizability. Similarly, the end of an operation can be either the return to a client or the last flush issued by the operation. Gotsman et al. define their TSO semantics to generate traces in which the call and return of a method (interface events) represent invoke and response of the operation. It is an elegant way to close the semantic gap between TSO and SC semantics, because it allows for reuse of the original linearizability relation without any significant changes to it.

The TSO-to-SC linearizability definition by Gotsman et al. [GMY12] is as follows:

Definition 19. *The **TSO-to-SC linearizability relation** is a binary relation $\sqsubseteq$ on histories defined as follows: $h \sqsubseteq h'$ if $\forall\, p \in P \bullet h|_p = h'|_p$ and there is a bijection $\pi : \{1, \ldots, \#h\} \to \{1, \ldots, \#h'\}$ such that $\forall\, i \in h_i = h'_{\pi(i)}$ and*

$$\forall\, i, j \bullet (i < j) \wedge \mathit{ret}?(h_i) \wedge \mathit{inv}?(h_j) \Rightarrow \pi(i) < \pi(j)$$

Again, one history linearizes the other if it is a permutation of the same events. The order of events of each process must be maintained. Furthermore, the real-time order imposed by response and invoke events must be respected. In contrast to the original definition, the TSO-to-SC linearizability definition does not state explicitly that the abstract history must be sequential. This is a slight generalization of the original definition (and also the case for TSO-to-TSO linearizability), because it allows for non-atomic abstract specifications. As argued before, correct behavior of data structure is usually defined in terms of atomic specifications which provide sequential histories only. Nevertheless, the inclusion of non-sequential histories into the definition enables two-step verification. In such a two-step approach, one would first show that a data structure implementation under TSO is TSO-to-SC linearizable w.r.t. an implementation under SC. Essentially, this step proves that the program is robust [BMM11] against TSO. In a second verification step, one would show that the

implementation under SC is linearizable w.r.t. an atomic data structure. The latter verification step would be free of weak memory concerns and thus, could be handled with common verification techniques for linearizability.

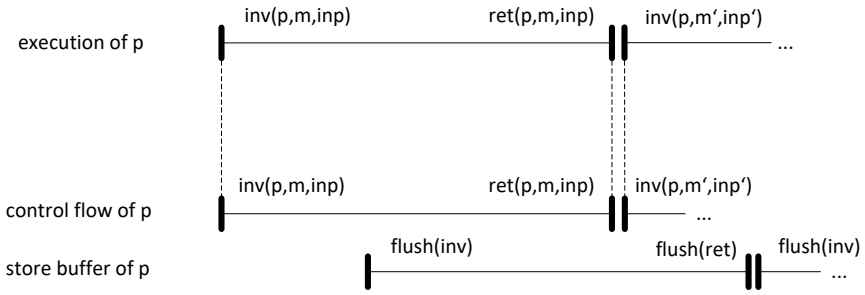

Figure 5.2: Visualizing TSO-to-SC linearizability; store buffer delays are ignored by abstract histories.

In order to contrast TSO-to-SC linearizability from TSO-to-TSO linearizability, we provide Figure 5.2. On the bottom, we have the same TSO history as before, where the events $\mathit{flush}(inv)$ and $\mathit{flush}(ret)$ represent the delay due store buffers. These events are exclusive to the TSO-to-TSO linearizability definition, but we leave them in the figure as a reminder for the delay of writes. Again, the figure shows only process p and its events for operation m followed by m'. The top history shows an SC execution of these operations, where the execution is completed at response time. Please note that the abstract history is a permutation of the concrete history and thus, identical events in the concrete and abstract history do not have to be exactly at the same position. What TSO-to-SC linearizability requires from a concrete data structure implementation is that its LP or its globally visible effect appears between the invoke and response events, just like in the original linearizability definition. In the figure, the operation m must linearize between the dashed lines. However, it also allows writes to be pending at response, e.g., writes that do not affect the visible abstract state or simply writes to variables that are exclusive to the writing process.

Gotsman et al. [GMY12] argue that some algorithms, even though commonly considered correct, are not linearizable w.r.t. this definition, because their linearizing operation, the LP, can be delayed outside the invoke-response interval. In such cases, they suggest relaxing the specification of the abstract data structure in a way that allows operations of it to fail occasionally. Although this can be simple in some cases like the Spinlock implementation (which was used by the authors to showcase such a relaxation), it won't be for most concurrent data structures as they are inherently more complex. Thus, by relaxing the abstract specification, there is high risk of

changing it in an unintended way. An abstract specification, which represents the correct behavior, may be rendered incorrect in this way and ultimately lead to an unsound verification. We do not promote this idea in general, particularly because we would be relaxing the correctness specification in order to be able to prove it. In the next section, we present an alternative linearizability definition that allows delayed linearization without the necessity to relax an abstract specification and without having to provide abstract specifications in terms of TSO histories as in the previous TSO-to-TSO linearizability definition.

In essence, TSO-to-SC linearizability provides the guarantee that all behavior of a concrete data structure (that is observable via interface events) can be reproduced by an abstract data structure. Everything this definition requires is the interface events of an execution, i.e., calls and returns of methods to be taken as invokes and responses in a history. The memory model does not even play an important role for this definition as long as we make some sanity assumption, e.g., histories of a single process are sequential w.r.t. its interface events.

Batty et al. [BDG13] proposed a similar definition of linearizability for the C11 and C++11 memory models. It also enforces linearization between invoke and response (like TSO-to-SC), but with C11 and C++1 it also takes weaker memory models than TSO into account. C11 and C++11 are ISO standards [ISO11b, ISO11a] for the programming language C and C++. Both support common modern multicore processor architectures like the x86 [SSO+10], Power [IBM15] or ARM [ARM13] and can be best viewed as a generalized programming interface to these architectures. Consequently, one could call the linearizability definition by Batty et al. "All-to-SC" or more precisely "C11-to-SC" linearizability.

TSO Linearizability

The previous two adaptations of the linearizability definition for weak memory models [BGMY12, GMY12] have some major practical problems. TSO-to-TSO linearizability has an inherently complex refinement relation, because the abstract specification can be itself a TSO implementation. TSO-to-SC linearizability is simple, but can be too strict for some implementations whose LP can be delayed past the response event. *TSO linearizability* [DSD14, DSGD17] bridges the gap between the other two as it allows delayed linearization, but relies on sequential abstract specifications. To the best of our knowledge, we were the first to present this idea informally [TMW13]. In [DSD14], it was partially formalized for the first time and a proof methodology for it was introduced. In a more recent version [DSGD17], TSO linearizability was fully-fledged formalized and the proof method from [DSD14] was generalized.

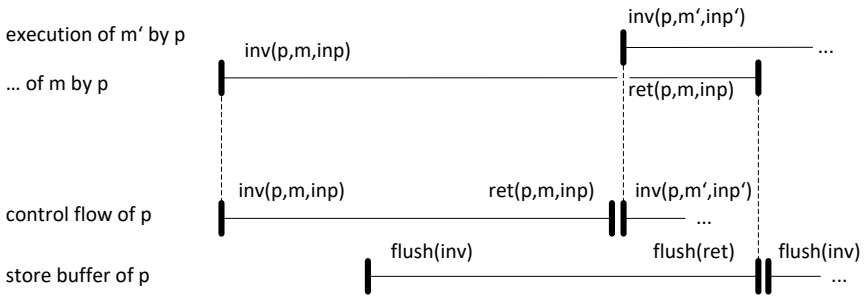

Figure 5.3: Visualizing TSO linearizability; last flush after response becomes the new response.

The key idea behind TSO linearizability is to extend the execution interval between invoke and response by the interval between the response and the last flush issued by a method. This allows methods that are issued by single process to be reordered w.r.t. the order in which they were issued. In Figure 5.3, we visualize the TSO linearizability idea as event intervals. Again, we leave the flush events as they are given in TSO-to-TSO linearizability in order to mark points in time where the store buffer begins and ends flushing content of operation m. The interval in which operation m must linearize, begins with the invoke of m. Depending on what is later, it ends either with the response event or with the latest flush event. In the figure, it is the $\mathit{flush}(\mathit{inv})$ event. This also enables linearization of the later operation m' while the previous operation m may not yet have linearized. At top, we have visualized the execution of both operation m and m' via separate time lines in order to highlight the overlap of them.

Please note that it is particularly this interval overlap that all three linearizability adaptations treat differently. TSO-to-TSO generally allows it, but requires a non-atomic abstract specification, because the behavior within this interval must refine the abstract specification. TSO-to-SC does not allow this interval to contain any important steps, i.e., flushes related to operation m within this interval are skip steps in the abstract specification. In other words, this interval is abstractly not visible. TSO linearizability simply treats this interval as an extension of the execution of operation m and thus, as if operation m and m' were executed concurrently, even though they are executed by the same process p. This time, the top history in Figure 5.3 is not the abstract history, but it is the result of a transformation applied to the concrete history. It is this transformed history which can have internal concurrency and which must be linearizable w.r.t. a sequential history. Here, we mean the original linearizability definition [HW90] modulo the well-formedness assumption, which

clearly does not hold in this case.

In contrast to the other two, TSO linearizability is mostly concerned with atomicity of a concrete data structure and that it behaves semantically correct. TSO linearizability does not enable for a sound replacement of a concrete specification with its abstract specification without changing the observable behavior. The latter is the case, because an abstract data structure is atomic and thus, it does not generate out-of-order executions as can be observed in concrete data structures due to operation overlap. However, TSO linearizability does guarantee that there is an equivalent sequential execution for a concurrent one (possibly not respecting program order). The justification of a concurrent execution with a sequential one, so far, has been one of the major concerns for most correctness conditions for concurrent data structures and that problem is solved by TSO linearizability without the necessity of complex abstract data types.

In order to formalize the notion of linearizability for TSO, flushes have to be taken into account. TSO linearizability is defined in terms of a history transformation. The transformation converts a concurrent history (containing flushes) into another concurrent history that consists only of invoke and response events and where the response events are either the original response events or the latest flush of the corresponding operation. After transformation, the original linearizability definition [HW90] is applied to the transformed concurrent histories. As mentioned previously, such a transformed history can violate the well-formedness assumption by Herlihy and Wing. However, because TSO linearizability cannot and does not intend to provide the same guarantees as the original definition, this is a lesser concern.

The first version of TSO linearizability [DSD14] and the latest version [DSGD17] differ in the way how flushes are represented and taken into account. In [DSD14], flushes are a third type of event (besides invoke and response) and belong to the process, who issued the method. A history is defined as a sequence of these events:

$$E_{tso'} \mathrel{\widehat{=}} inv(P \times M \times INP) \mid ret(P \times M \times OUT) \mid \mathit{flush}(P)$$
$$H_{tso'} \mathrel{\widehat{=}} seq(E_{tso'})$$

Informally, the authors state that the histories in $H_{tso'}$ are transformed by replacing flush events by response events and removing previous response events that are no longer required (only the last response of a method remains). After transformation, the resulting histories contain only invoke and response events, but also allow for overlap within a process. As a consequence of the extension of execution intervals, the real-time order that has to be preserved in an abstract history is relaxed. Without the extension, consecutive methods would have to remain in order in the abstract

history. With the extension, they can appear in any order if their intervals overlap, i.e., if they are concurrent.

As some parts of the former definition of TSO linearizability were given informally, the authors provide an updated and full-fledged formal definition in [DSGD17]. The major changes were applied to representation of histories and a formalization of the transformation was added. In contrast to the previous work [DSD14], flushes are assumed to be issued by a separate cpu process and each flush has its own invoke and response event. A flush is an atomic operation, i.e., invoke immediately followed by corresponding response. By the choice of representing flushes at the same granularity of a method, the authors were able to avoid a third type of event. Histories are defined as:

$$E_{tso} \mathrel{\widehat{=}} inv(P \times M \times INP \times \mathbb{N}) \mid ret(P \times M \times OUT \times \mathbb{N})$$
$$H_{tso} \mathrel{\widehat{=}} seq(E_{tso})$$

In their updated history definition, invoke and response have an additional integer parameter. It is required for the memorization of pending writes in the store buffer. Recording the number of pending writes at invoke and response events allows for identification of the last flush event correspond to them and thus, also to determine the extension of the execution interval. The latter can be determined by computing arithmetic differences, which define the number of flushes that have yet to appear before the last flush.

As a consequence of the deviation from the original invoke and response events, the authors also adapted other parts of the definitions like matching pairs of invoke and response events. However, their definition [DSGD17] remains very similar to the original definition:

Definition 20. *A history $h : H_{tso}$ is **TSO linearizable** with respect to some sequential history hs iff $lin_{TSO}(h, hs)$ holds, where*

$$lin_{TSO}(h, hs) \mathrel{\widehat{=}} \exists\, h_0 \in Hist_{FR} \bullet legal_{TSO}(h \frown h_0) \wedge$$
$$linrel_{TSO}(Trans(complete(h \frown h_0)), hs, h \frown h_0)$$

where $legal_{TSO}(h)$ states some sanity assumptions about a history h, i.e., responses have a corresponding invoke and that there are no more flushes than issued by methods. However, as previously mentioned, legal histories do not have to be necessarily well-formed. The set $Hist_{FR}$ represents sequences of flushes and returns that are required for completion of a history just like in the original linearizability definition. The actual linearizability relation is stated in $linrel_{TSO}$, which is applied to the transformed complete history $Trans(complete(h \frown h_0))$. We refrain from

showing the full formal definition, since we would have to define more fundamentals. Informally, the relation $linrel_{TSO}(h', hs, h)$ encodes the following:

1. existence of permutation f on h' (the transformed history) to a sequential history hs
2. the permutation f also orders the matching pairs (invoke and response) in h (not transformed history) sequentially
3. the real-time order (derived from h but takes extended intervals into account) is respected by the permutation f.

Of course, a specification is TSO linearizable, if all its histories are. That is similar to the other definitions. As the name suggests, TSO linearizability has not been adapted to other memory models yet, although the principle of execution interval extension should also be applicable to other memory models like PSO or RMO.

5.2 Discussion

In this thesis, we do not want to determine the next standard definition for concurrent data structures. However, we do prefer TSO-to-SC and TSO linearizability over the TSO-to-TSO linearizability because of their simplicity. As the above three adaptations of linearizability towards weak memory models show, there are arguments for and against each one of them. These pose a tradeoff between simplicity and the refinement guarantees that they provide.

TSO-to-TSO linearizability provides the strongest guarantees in the sense that a concrete data structure can be replaced in the context of a client by its abstract data structure while maintaining all observable behavior. The latter also includes observable behavior after a method response. However, it can be difficult to find an abstract data structure that is indeed more abstract than the concrete one and also provides the same behavior. In other words, it can be difficult to find a correct specification for an implementation. Furthermore, the relationship between abstract and concrete data structure is more complex than for the other definitions and thus likely more difficult to prove.

TSO-to-SC linearizability also provides strong guarantees and essentially establishes very similar guarantees under weak memory models as provided by the original linearizability definition under SC, i.e., a method seemingly takes effect atomically at some point between its invoke and response. Even though the definition assumes a weak memory model, it does not change this very fundamental concept of the original definition. The latter is also the reason why this definition is easy to understand for somebody familiar with the original definition by Herlihy

and Wing. However, as discussed above, TSO-to-SC linearizability can be too strict for some implementations which are commonly accepted as correct.

TSO linearizability provides the weakest refinement relation to abstract data structures among the considered definitions. It guarantees that methods seem to take effect atomically. However, methods have an extended interval to do so, which is also the reason why the guarantees provided by TSO linearizability are weaker than for the other definitions. In contrast to the other definitions, TSO linearizable data structures cannot be soundly replaced by their abstract data structure. The program order of single processes is only maintained partially in the permuted histories of abstract data structures. Furthermore, preservation of the real-time order does not have the same implications as for the other definitions, because the response events after history transformation do not necessarily represent actual response events, but can also be flushes after an actual response. Thus, TSO linearizability shows that an implementations behaves like a sequential implementations, but not necessarily within the same boundaries.

In the remainder of this thesis, our verification approach will rely on TSO-to-SC and TSO linearizability. Our model checking approach in Chapter 6.1 was developed with TSO linearizability in hindsight, although it was not formally defined at that time, yet. We can therefore claim to have the first model checking approach for TSO linearizability. Our proofs in Chapter 6.2 were carried out for TSO-to-SC linearizability, essentially by using a proof method [DSW11a] that was developed for the original linearizability definition [HW90]. The latter was possible, because we only had to exchange SC program semantics with TSO program semantics and assume fences at invoke and response. Our reduction from Chapter 3 helped us to achieve that as we were able to use SC programs that are equivalent to the same programs under TSO. We did not consider TSO linearizability as a correctness condition for our proofs, because the underlying theory and proof method [DSW11a] for the original linearizability definition would require significant changes for it. The latter were out of scope of this thesis.

5.2.1 Verification Methods for Linearizability

Many verification approaches have evolved since the proposal of linearizability by Herlihy and Wing [HW90]. These can be divided into two types of verification: The first type is state-space exploration-based approaches (model checking) which in most cases cannot show correctness (linearizability) of a data structure. This is because most data structures allow for infinite state space. Thus, unless abstractions are applied, the state space cannot be fully explored. Instead these approaches are good for finding bugs, because they can be automated to large extend and do not need

as much expertise as for a formal proof. The second type of verification approaches aims at ultimate correctness arguments as a result of formal proofs. Correctness proofs do require lots of expertise and are generally hard to automate, because the arguments for data structure correctness are often complex. In this section, we discuss existing verification approaches for linearizability under assumption of sequentially consistent memory model. A discussion on linearizability verification that also involves weak memory models can be found in Section 6.3.

Model Checking

While there are plenty of tools and methods for testing concurrent programs [MQ06, CBM10, Sen15], testing usually provides no guarantee of correctness. That is because testing examines only a (usually) small fraction of possible program executions and thus, testing can only show the presence of bugs, but not their absence. In contrast to testing, model checking [Cla08] explores the whole state-space of a program and thus can determine whether a bug is present or not. However, this applies only to finite state programs or programs whose state-space representation is finite. The key problem from which model checking suffers is the state explosion problem [Val98]. Many techniques evolved to push the boundaries at which the state space of program becomes impossible to handle. Among these techniques, there are CEGAR [CGJ+00], symbolic model checking [BCM+92], partial order reduction [Val89], bounded model checking [BCCZ99] and many others, as well as combinations of them.

Linearizable data structure rarely have a finite state space, unless they are trivial. Most of them are designed for use with an arbitrary number of processes. Model checking requires the number of processes and their operations to be fixed (a test scenario), before being able to explore the state space of a concurrent program. Furthermore, we can define an arbitrary number of different scenarios by using different parameters, combinations of operations or number of processes. Thus, concurrent data structures cannot be fully explored using model checking techniques as there is always the risk of missing a scenario in which the implementation can fail. Instead, model checking of linearizable data structures has to be viewed as an instance of extensive and systematic testing, i.e., model checking of concurrent data structures usually cannot show the absence of a bug, but only its presence. However, experience shows that small test scenarios are often sufficient to find concurrency bugs [BAM07].

Some model checking approaches do not even attempt to check for linearizability, but they check for atomicity of operations [Fla04, WS05]. Atomicity is important for linearizable data structures as it enables operations to take effect atomically. Flanagan [Fla04] instruments code blocks with consistency checks at commit points

which are similar to LPs in linearizability. In fact, his approach is very similar to our own model checking approach in Section 6.1. In Flanagan's approach, the instrumented checks take the sequential semantics into account, which in our approach is an abstract specification of the data structure. Wang and Stoller [WS05] propose an inference of atomic blocks of code that also can be used as an instance of partial order reduction, or if the complete operation can be inferred as atomic, then as a technique to show atomicity of operations. A drawback of atomicity is that one has to know in advance where the commit point/LP of an operation, s.t., it can be instrumented.

A very straightforward idea of model checking linearizability is to compare concurrent histories with sequential histories. This is also the principle behind [VYY09, BDMT10]. Both approaches instrument the implementation at invoke and response, in order to record the events with their parameters or outputs in a history. Vechev et al. [VYY09] first record the concurrent history and check it later against possible sequential histories. Because the concurrent history is part of the program state, there is a bound on the number of invocations per process since each modification of the history also introduces a new state. Burckhardt et al. [BDMT10] do the check in a different order. First, they collect all possible sequential histories and then they check against concurrent histories that are obtainable via exploration. Furthermore, they assume the concurrent data structure to be correct in a sequential setting. This enables them to generate the sequential histories by executing the concurrent implementation without having to provide an abstract data structure for it. The latter enables a fully automated linearizability check.

Other instances of model checking make use of the refinement relation between an abstract and a concrete data structure as in [LCLS09]. Liu et al. [LCLS09] characterize each step of a concrete data structure (more precisely of its labeled transition system (LTS)) as invoke, response or LP. A refinement check on the cross-product of the concrete and an abstract data structure LTS then reveals whether they are consistent. However, their approach is only sound if all of the LPs are known and a check has been performed that shows that no other step of the LTS can be an LP.

Another model checking approach [CRZ+10] proposes the use of method automata as a formalism for data structure specifications. The linearizability check is reformulated into a reachability check on method automata. Method automata are limited to list-based data structures only. The approach does not need any knowledge about LPs, but can benefit from this knowledge as the authors note.

Proof Methods

Since model checking is not sufficient for showing linearizability of a data structure, proof techniques have to do this task. In fact, Herlihy and Wing [HW90] were not only the first to provide a definition of linearizability, but also presented the first proof technique together with the definition. Their approach relies on an abstraction function that maps from concrete states to abstract states. The essential idea for a proof of correctness is to show a subset relation between concrete and abstract states. This is achieved by proving that the abstraction of every concrete state reached by a history is contained in a set of non-empty abstract states that are reachable via a linearization of the same history. Besides being a manual proof technique, their approach is structurally similar to a backward-simulation as Henzinger et al. [HSV13] point out. Backward-simulations are inherently complex to prove.

Simulations are widely used by linearizability proof techniques[GH04, CDG05, DSW07, Hes07]. Gao and Hesselink [GH04] define refinement mappings as a means to reduce a concrete data structure to its abstract data structure. The refinement mapping also involves a simulation between both specifications. Colvin et al. [CDG05] use forward simulations to prove refinement between IO-Automata [LT89]. The latter is a formalism for concrete and abstract data structures. Both, backward and forward simulations have been proven to be sound [DSW07, DSW11a], but only backward simulation is also a complete method to prove linearizability [DSW11a]. Derrick et al. [DSW07] extend the work by Colvin et al. [CDG05] by restating linearizability as a non-atomic refinement property [DW05]. The latter allows an abstract operation to be implemented by a sequence of concrete steps. The general idea is that a linearizable data structure non-atomically refines an abstract one. If a concrete data structure simulates an abstract data structure, then it must also refine it and thus is linearizable. In [DSW11b, SWD12], the authors also provide generalized proof obligations for both, thread-local and global reasoning. The proof obligations together with their soundness and completeness proofs are provided as libraries to the theorem prover KIV [EPS+14] and are ready to use for mechanised proofs. These proof obligations are also the foundation for our own linearizability proofs presented in Chapter 6.2.

Vafeiadis [VHHS06, Vaf07] combines separation logic [Rey02] with rely/guarantee reasoning [Jon83] into a new logic RGSep. He uses the latter to reason about linearizability. His approach is different from simulation-based proof methods, because the abstract state of a data structure is embedded into the concrete state. The actual proof then shows, that an abstraction map holds throughout the entire program. For some if his case studies, he was able to fully automate the proofs.

Many other proof methods emerged for linearizability. Some show atomicity

via simulation and refinement [Hes07], use shape-analysis as a static reasoning technique for linked data structures [ARR+07], or use temporal logic (TL) as another means for thread local reasoning about linearizability [BSTR11, TSR14]. Other approaches [EQS+10, Jon12] try to abstract and reduce concrete operations to the point at which complete operations are atomic. If all operations can be reduced to atomic operations, then the concrete data structure can be considered linearizable. While Elmas et al. [EQS+10] alternate between abstraction and reduction steps, Jonsson [Jon12] uses only reduction steps and maintains a refinement relation.

A lot of effort has also been spent on reducing the complexity of proofs. A notable approach was proposed by Henzinger et al. [HSV13] in that direction. The approach restates linearizability of queues into four comparably simple properties which can be proved independently. Together these properties imply linearizability of queue implementations, but are also restricted to queues only.

A more extensive survey on proof methods for linearizability can be found in [DD15].

5.3 Other Correctness Conditions

Besides the above variants of linearizability, there are several other relaxations of linearizability [HS08, SK09, AKY10, HKP+13]. Part of the motivation for further relaxation of linearizability is that concurrent data structures, even though they are optimized for concurrency, can still become the bottle neck of a system due to contention. Furthermore, the semantics as provided by an abstract data structure in linearizability can be too strict in many cases. For instance, it is not important for most schedulers to strictly adhere to FIFO order of a queue as long as processes do not get preempted for too long. Some variants of linearizability or similar conditions take this into account in their definition and allow concurrent executions to deviate further from a legal execution than linearizability (which requires real-time order preservation).

Quasi-linearizability [AKY10] is one such condition. It relates concrete and abstract histories in a similar way to linearizability, but in addition allows permutations to violate real-time order by a certain bound. Thus, if there exist a legal history for a concurrent history or a permutation of it within the range of a predefined bound, then the concurrent history is considered quasi-linearizable. A similar condition was proposed by Henzinger et al. [HKP+13] and is called *k-linearizability*, where k is also a bound. The difference to quasi-linearizability is that the bound k is not tied to permutation of events, but to an abstract state. In this sense, it is a semantic distance and relaxation. As an example, in a k-linearizable stack it is legal to pop

one of the first k elements where only the top element would be allowed by original linearizability.

Quiescent consistency [HS08] takes the relaxation one step further and requires consistency only at certain states, the quiescent states. A quiescent state is a state, in which no operation is active. Quiescent consistency requires that whenever an execution reaches a quiescent state, then there exists an equivalent sequential history leading to the same state. In between two quiescent states, any behavior is allowed as long is it becomes consistent whenever a quiescent state is reached.

An even more relaxed correctness criterion is *eventual consistency* [SK09], which is not even a safety property but a liveness property. Eventual consistency guarantees that in the absence of new changes, eventually all processes observe the same state (or all reads return the same value). Although it sounds vague, it is a common criterion among distributed software systems with replicated data.

While the above conditions are relaxations to linearizability, they adhere to all types of data structures. Another line of recent research goes towards specific data structures like the transactional memory [HM93]. A transactional memory (TM) allows for arbitrary transactions consisting of reads and writes (like database transactions). A transaction must appear atomically or it must remain invisible to all other transactions. In contrast to linearizable implementations, which have a small and finite set of operations, TMs can have arbitrary transactions that are not known a priori, but still must be guaranteed to appear atomically. Intuitively, a TM is correct if it is serializable [Pap79]. However, serializability does not take aborting transactions into account and thus there remains an open semantic gap. Most of the proposed correctness conditions [GK08, DGLM13, AGHR14] try to fill this gap by relating internal behavior of transactions to that of an abstract specification. In contrast to linearizability, TMs require atomicity of transactions, where each transaction is implemented by multiple operations like *begin*, *read*, *write*, *commit* and *abort*. The complexity of TMs lies within the relationship between these operations and an abstract history, in which they have to appear sequentially ordered by each transaction.

None of the above conditions take weak memory into account, but assume sequentially consistent semantics. Taking weak memory models into account for the above condition raises similar questions as for the adaptations of linearizability, i.e., when does an operation begin and when does it end its execution? Should an operation's execution interval be extended as in TSO linearizability and what would be the consequences for the correctness condition? These questions are out of scope of this thesis. However, a similar approach to the TSO-to-SC adaptation of linearizability by Gotsman et al. should be possible for all of the above correctness

conditions, because the general idea to take the interface events of an operation as the invoke and response events in a history does not dependent on the underlying memory model.

6 Verifying Linearizability under Weak Memory Models

Linearizability [HW90] is a de-facto standard correctness criterion for concurrent data structure implementations as we pointed out in the last chapter. Since its proposal in the early nineties by Herlihy and Wing, many techniques have been developed for the verification of linearizability [VHHS06, DSW07, EQS+10, BDMT10, TMW13]. However, only few consider weak memory models in their approaches. It is uncertain which of the linearizability definitions (see previous chapter) will become the new standard under consideration of weak memory models. Nevertheless, we think linearizability will remain an important correctness condition and thus, we aimed at adapting existing verification techniques to the weak memory model setting.

The reduction that we proposed in Chapter 3 already defines the semantics of TSO and PSO. Once applied to a program, the reduction provides the weak program specific semantics in terms of a new SC program that reproduces the behavior of the original program under TSO or PSO. In this chapter, we focus on two different methods that fit into our overall approach for the verification of concurrent data structures under weak memory models. We use linearizability as the correctness criterion. We follow two different branches corresponding to two different general verification techniques. The first is model checking [Cla08], which is good for finding bugs in implementations. It can be applied automatically and thus can yield results quickly. However, most concurrent data structures can have an infinite state space, because the elements they can hold is usually unbounded as well as the number of processes sharing and using the data structure. Thus, model checking usually cannot show correctness of an implementation. In order to show correctness of an implementation, one would have to explore the complete state space of it. The latter is only possible, if the state space is finite or at least, if it is possible to represent it in

a finite manner, which would consequently make the check finite. Since this is not always possible, our approach uses deductive verification as a second verification technique. Due to their complexity, correctness proofs require a lot of expertise and manual effort. Interactive and partially automated theorem provers can help with proving algorithms correct. Anyway, the manual effort of a correctness proof is usually significantly larger than the effort required for a correctness check. This is also why model checking should always be applied before a correctness proof is attempted. Only, if model checking reveals no implementation errors, a proof attempt makes sense.
Throughout the chapter, we will try to answer the following two research questions:
RQ1: How can we use the proposed reduction for the verification of concurrent data structure implementations?
RQ2: What are the advantages and disadvantages of using the reduction compared to conventional approaches?

In order to answer **RQ1**, we propose different methods that rely on existing verification methods for linearizability under sequential consistency. We revise the original methods by replacing their SC program inputs with the programs generated by our tool Weak2SC. This also includes programs that are generated in combination with an operation memory model or with an explicit encoding of store buffer behavior. The latter will help us answer **RQ2**, since verification based on operational memory models can be considered as a conventional approach. We perform experiments in order to determine the impact of the reduction on the verification in terms of performance, but also investigate the verbosity of the generated models. Verbosity and complexity are especially important for the proof approach, where deep understanding of a program can be more crucial than the automation of the actual proof.

In the following, we present each approach as one possible answer to **RQ1**. Please note that the reduction can be applied to many other verification techniques, which are not necessarily tied to linearizability. Each of the presented approaches is followed by experiments that will help us answer **RQ2** or at least discuss experimental results and our own practical experience.

6.1 Model Checking under Weak Memory Models

We have two model checking approaches that are revised for weak memory models. We have one that we think is promising in terms of performance and practicality and another that is general, but does not perform well. In the following, we present our first approach in detail and provide experiments. The second approach

will be presented without experiments for the sake of an alternative checking procedure that can be easily adapted to other correctness conditions, e.g., quiescent consistency [HS08].

6.1.1 The Idea - Abstract Atomic Specifications

Our first approach requires an abstract atomic specification of the implemented data structure. The general idea is to use the abstract atomic specification for consistency checks at linearization points (LP) of the implementation. The proposed instrumentation and consistency checks are similar to an existing approach for verification of commit atomicity [Fla04], a property that requires executions to be serializable [Pap79]. Since serializability is weaker than linearizability [HW90], we can also use it for detection of non-linearizable executions. The key difference between the two is that serializability requires executions only to be equivalent to any sequential execution while linearizability in addition requires the sequential execution to conform to a sequential execution of an object. In [Fla04] the implementation is instrumented at commit points that are similar to linearization points in linearizability. However, the approach does not consider weak memory models and does not promote the use of an abstract specification. The latter is a crucial difference to our approach and ensures the conformity of concurrent executions to a sequential execution of the underlying data structure. An initial publication of the approach together with the operational memory model for TSO that we use in Weak2SC can be found in [TMW13].

LPs are the steps of an implementation at which an operation takes effect and becomes visible to all processes. Thus, before an operation reaches its LP, all other processes must not see the effect of the operation that is about to linearize. After an LP (or after the operation linearized), all processes must observe a state in which the operation seemingly has finished. Each LP corresponds to an operation. Each process is essentially a sequential program that is a sequence of operations. Concurrent executions are interleavings of the sequential programs corresponding to different processes. In a concurrent run, the sequence of reached LPs can be used to construct the sequential history that linearizability demands for a concurrent run to be linearizable.

Instead of computing the actual history for a concurrent execution, we use the abstract specification and let it perform all operations simultaneously with the implementation, whenever a step corresponding to an LP is taken by the implementation. The abstract specification is encoded as part of the program model to be checked. The operations of the abstract specification must be atomic, such that they can be performed atomically at the linearization points (LP) of the implementation. For

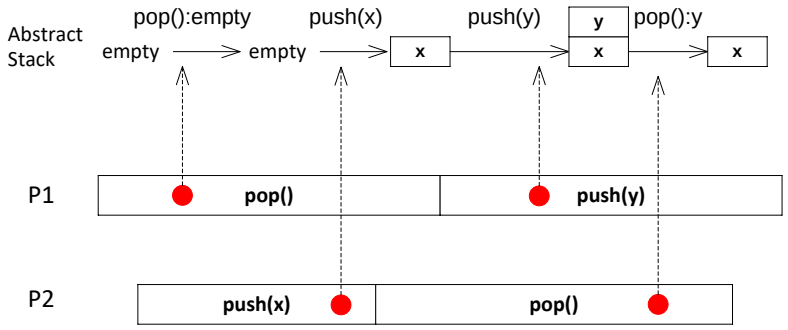

Figure 6.1: Linearization points in concurrent executions and the corresponding modification of the abstract state.

instance, if a stack implementation linearizes on a *pop* operation by observing an empty state, then this should be consistent with the state of the abstract specification. If an operation modifies the data structure, then this has to happen to the implementation as well as to the abstract specification. In addition, LPs must be instrumented with consistency checks, such that inconsistencies are reported as counterexamples by the model checker. As an example see Figure 6.1. It depicts two concurrently running processes performing two consecutive stack operations each. Time progresses from left to right. $P1$ performs a *pop* operation first and then a $push$ operation with argument y. $P2$ pushes x on the stack first, and then attempts to pop an element from the stack. The red dots represent LPs of the operations. Corresponding to the LPs, the abstract stack must perform the same operations atomically with the same outcome. The abstract transitions corresponding to LPs are indicated by a dashed arrow.

Imagine now, the LPs of the *pop* operation by $P2$ and the *push* operation by $P1$ would be reordered. The *pop* operation would then observe a value y, which was not yet pushed to the stack, because the *push* LP of $P1$ would have yet to be reached and thus, has had no effect yet. The abstract specification enables detection of such situations. Consistency checks at LPs must compare the values of the concrete program with the values obtainable from the abstract state. If an inconsistency is detected, either the program has a bug which must be corrected or the LP instrumentation is incorrect. In both cases a counterexample will be reported by the model checker and adjustments will be necessary, either to the program or to the LP instrumentation. The method obviously requires knowledge about the linearization points of an implementation in order to be able to instrument it. Otherwise, the model checker would report spurious counterexamples that are reported due to wrong instrumentation and not because of an actual bug or missing fence in the implementation.

Example application to Work-stealing Queue

In the following, we give an example for the work-stealing queue by Arora et al. [ABP98], which we presented already in the previous chapters. The Promela models generated by WEAK2SC for it can be found in Section 4.4.

As a brief reminder, the work-stealing queue is owned by one process. The owner adds and removes elements from the bottom end of the queue in a stack-like (LIFO) manner. This is also the reason why the corresponding methods are named *pushBottom* and *popBottom*. Other processes can steal an element from the top end (method *popTop*). Stealer processes essentially dequeue the oldest element in the queue. In order to check the work-stealing queue for correctness, we need an abstract data structure that supports these operations.

Figure 6.2 depicts a Promela model for the double ended queue data structure, which is implemented by the Arora et al. queue [ABP98]. We also used it in our experiments that we will discuss in the following section. It uses an array *asDeq* to store its elements. The length of the array is determined by *ASSIZE*, which can be adjusted to personal needs. The variables *asTop* and *asBot* represent both ends of the queue and are initially 0.

The model in Figure 6.2 contains four different atomic operations corresponding to four different possible linearization points in the queue implementation. The first is a linearization corresponding to an empty state. The corresponding abstract atomic operation (lines 5-7) contains only an assertion that checks for emptiness in the abstract state (*asTop == asBot*). Only *popBottom* and *popTop* can linearize by observing an empty state. In such a case, the observable abstract state is not modified. Thus, the *asEmpty* operation does not modify the state and only checks the assertion. A succeeding *popBottom* method removes an element at the bottom end of the queue. The atomic operation *asPopBot* corresponding to the LP is modelled in lines 9-17. It ensures that the abstract queue is not empty. It also has a parameter *popValue* which has to be equal to the bottom element of the abstract state (*asDeq[asBot]* after *asBot* was decremented). The LP corresponding to a succeeding *popTop* method is represented by the *asPopTop* operation. In contrast to the previous operation, it has to ensure that the popped value is equal to the top value in the queue *asDeq[asTop]*. The method *pushBottom* always succeeds. Thus, the operation corresponding to its LP *asPushBot* only ensures that array representing the queue has enough space for one more element. Besides this it adds the same element (passed by parameter *pushValue*) to the queue at the bottom end.

So far, we have only seen an example of how an abstract data structure together with built-in consistency checks can be modelled in Promela. What is still missing is the instrumentation of the actual program behavior that triggers these checks.

```
1  #define ASSIZE 5
2  short asDeq[ASSIZE];
3  byte asTop = 0, asBot = 0;
4
5  inline asEmpty(){
6    assert (asTop == asBot);
7  }
8
9  inline asPopBot(popValue) {
10   atomic
11   {
12     assert(asTop < asBot); //not empty
13     asBot--;
14     assert (asDeq[asBot] == popValue);
15     asDeq[asBot] = 0;
16   }
17 }
18
19 inline asPopTop(popValue) {
20   atomic
21   {
22     assert(asTop < asBot); //not empty
23     assert (asDeq[asTop] == popValue);
24     asDeq[asTop] = 0;
25     asTop++;
26   }
27 }
28
29 inline asPushBot(pushValue) {
30   atomic
31   {
32     assert(asBot < ASSIZE); //make sure, stack array is never full
33     asDeq[asBot] = pushValue;
34     asBot++;
35   }
36 }
```

Figure 6.2: Abstract double ended queue specification in Promela. All operations are atomic.

Figure 6.3 shows the instrumented Promela code for the *pushBottom* method that was generated by Weak2SC based on our proposed reduction. The instrumentation on its own is simple. The difficulty lies in finding the LP of a method, which requires experience and expertise. Once the step corresponding to the LP is identified, it is replaced by an atomic statement. The statement will then perform both steps, the original step corresponding to the LP and its corresponding abstract operation simultaneously. The consistency checks as part of the abstract operation will trigger during an exploration, if one if its internal assertion fails. For the *pushBottom* method, the LP is the step following the label *A08v0* in line 18. The step corresponds to a

```
1  inline pushBottom(elem){
2  short v0, v1, v2, arrayidx, inc;
3  AStart: goto A00;
4  A00: v0 = memory[bot]; goto A01;
5  A01: v1 = memory[v0]; goto A02;
6  A02: v2 = memory[deq]; goto A03;
7  A03: getelementptr(10, v2, v1, arrayidx); goto A04;
8  A04: goto A05arrayidx;
9  A05arrayidx:
10   if
11   :: inc = v1 + 1; goto A06arrayidx;
12   :: memory[arrayidx] = elem; goto A05;
13   fi;
14 A06arrayidx: memory[arrayidx] = elem; goto A06;
15 A05: inc = v1 + 1; goto A06;
16 A06: goto A07;
17 A07: goto A08v0;
18 A08v0: atomic{memory[v0] = inc; asPush(elem);}; goto A08;
19 A08: goto AEnd;
20 AEnd: skip;
21 }
```

Figure 6.3: Instrumented queue implementation for consistency checks against abstract data structure. Program model generated by WEAK2SC based on reduction from store buffer graphs.

flush of the incremented value of the *bot* variable.

What looks simple for the reduction-based program models, can be slightly more complicated for models based on an operational memory model. Since we need to instrument the step corresponding to an LP, we need to modify the operational memory model, if the LP corresponds to a flush. The latter is the case that we have in the *pushBottom* method. Figure 6.4 shows the instrumented Promela code for the control flow of the *pushBottom* method. The LP is in line 12, which is the write of the incremented value of variable *bot*. In order to instrument it, we use a special write inline statement with a third parameter for the pushed value. Because of the separation between program control flow and executions semantics in two different processes, we have to flag the write as an LP. The flagged write will then trigger the abstract operation *asPush* during its flush. The triggering of the abstract operation and the additional field in the write buffer for the LP flag must be added to the operational memory model manually. The extension has to be added with care as the execution semantics provided by the operational memory model must not be changed by the modification. We refrain from showing the extension of the models, as we would have to go through all the encoding details of an operational memory model in Promela. Instead we refer to our Git repository [Tra16], where it can be

```
1  inline pushBottom(elem){
2  short v0, v1, v2, arrayidx, inc;
3  skip;
4  entry:
5   read(bot, v0);
6   read(v0, v1);
7   read(deq, v2);
8   getelementptr(10, v2, v1, arrayidx);
9   write(arrayidx, elem);
10  inc = v1 + 1;
11  mfence();    //required for PSO
12  writeLP(v0, inc, elem);
13  goto ret;
14 ret: skip;
15 }
```

Figure 6.4: Instrumented queue implementation for consistency checks against abstract data structure. Program model generated by WEAK2SC for the use with an operational memory model.

found.

Please note that we cannot trigger the abstract operation from the program process, because a write in a program is non-atomic and corresponds to two separate steps. The first step, performed by the program process, moves the value to the buffer. The second step flushes it to the memory at a later point in time. The latter step is performed by the semantics process, which is defined by the operational memory model. Thus, the abstract operation must be triggered within the semantics process. Otherwise, the abstract state would be modified too early, i.e., before the write is flushed and becomes visible to other processes. In such a case, a concurrent process could try to steal (*popTop*) an element from the queue and would not be able to observe a state that is consistent with the abstract state. The abstract state would contain a new element that is not yet visible to other processes.

Since we have discussed how to instrument a flush transition in our operational memory models, the reader might ask how instrument early read transitions. The answer is simple. We *do not instrument early reads*, because a transition corresponding to an early read cannot be a linearization point. The reason becomes obvious from the SC programs that are generated from the store buffer graph. Early read transitions are transformed to local assignments in these programs, because they are not visible to other processes. A local transition cannot be a linearization point of any deterministic program. The linearization point has to be a transition that is visible to all processes, i.e., a transition accessing the shared state. In our approach, these are always transitions that access the memory, either writing, reading or both

in case of CAS.

The following steps summarize the approach for the verification of programs under weak memory models in combination with an abstract specification:

1. Generate the Promela model for the program using the Weak2SC tool for the target memory model. This is will be the concrete specification of the program. It can be either based on the proposed reduction (an SC program model) or it can be a program model with an exchangeable operational memory.

2. Embed an abstract atomic specification of the data structure into the concrete specification.

3. Identify the linearization points of the program and instrument them with the corresponding abstract operations. Add assertions as part of the instrumentation that will report inconsistencies between the concrete and the abstract specification. If concrete and abstract states are inconsistent, then either the LP instrumentation is incorrect or the concrete specification is not linearizable.

4. Perform a state space exploration. Start with a simple scenario consisting of few processes and operations. Increase the complexity of scenario iteratively. If counterexamples are reported, make sure that they are not spurious due to an incorrect LP by inspecting the counterexample trace. If no counterexamples are reported by the model checker, then the program is probably a good candidate for a linearizability proof.

Abstract Atomic Specification vs. Potential LPs

Linearizability requires programs to behave as if they had atomic operations. Thus, the proposed consistency check at LPs against an abstract atomic specification will certainly reveal many bugs that can occur in a non-linearizable implementation. However, there are also concurrent data structure implementations that cannot be checked using this approach or which are at least difficult to instrument properly. The reason for this is that some implementations have *potential* linearization points [DSW11a]. Potential LPs are LPs that cannot be statically fixed to a certain program location, because whether an operation linearizes or not depends on the execution of other concurrent processes. In other words, the LP of one operation can be also the LP of several other concurrently running operations.

As an example, think of an array based implementation of container data structure that has the operations *add*, *remove* and *contains*. It should be easy to find the program step that corresponds to the LP of *add* and *remove*, because it will be a step that makes an element visible (resp. invisible) to other processes. Assuming the

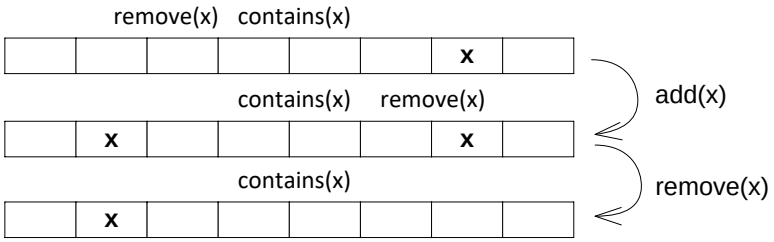

Figure 6.5: Potential linearization points in an array based container data structure.

contains operation would be implemented by a walk over the array, it is impossible to fix the LP to a certain program location. Think of an array that has an element x placed at the end of the array as depicted in Figure 6.5 in the top array. While the *contains* operation walks the array searching for x another concurrent *remove* operation could succeed. Thus, the *contains* operation cannot have linearized, yet. However, an *add* operation could add x at an early index (second array in the figure). The operation *contains* has already passed that index and thus will not observe the newly added element x. In the meantime, the *remove* operation is scheduled to finish and removes element x from the array (bottom array in the figure). The latter causes the *contains* operation to finish with a negative result, even though the array contained the element x at all points in time. The question may rise whether this is a linearizable execution and the answer is yes. Linearizability imposes no order on concurrently running operations, only on those which are sequentially ordered. In the example all three operations are concurrent and thus, can be reordered arbitrarily in a justifying sequential history. For the example, the linearization order is 1. $remove(x)$, 2. $contains(x)$ and 3. $add(x)$. It is the only order that justifies the *contains* operation to not observe element x.

What is interesting in the example is that the array modifications by *add* and *remove* must be reordered in order to find an equivalent sequential history for the concurrent execution, because of the potential LP of *contains*. In our approach, this is not feasible, because the LPs of each operation must be fixed in the code. Otherwise, they cannot be instrumented. Furthermore, we use the abstract atomic specification as an alternative for the construction of a sequential history from a concurrent execution. This is achieved by capturing the sequence of LPs in a concurrent execution and by applying the same changes atomically in the abstract specification. Inconsistencies between abstract and concrete state reveal non-linearizable executions.

However, linearizability allows for concurrently executed operations to linearize in any order as long as the semantics of the implemented data structure allows it. Thus, whenever several operations appear concurrently, there can be several possible

linearization orders which consequently lead to different abstract states. If all LPs can be fixed statically, then there is just one such order. It is the sequence of LP occurrences in an execution. Potential LPs cannot be fixed statically. Thus, there are several possible linearization orders to be considered which correspond to different abstract states. Which of the possible states is the correct one at any point in time, is determined by the future of the execution.

For our approach, this means that we would have to validate against a dynamic set of possible abstract states instead of just one abstract state. An encoding of such a dynamic set of possible states would probably exceed the complexity of the actual data structure implementation that is to be verified. Thus, we restrict this approach to the class of data structure implementations that have no potential LPs. Implementations with potential LPs should be checked with a different approach. One such approach is presented in Section 6.1.3. It is based on history recording and was developed together with the help of one of our bachelor students. A more detailed discussion on potential linearization points can be found in [DSW11a, TTSW14]. The above example is taken from [TTSW14] and is a simplified version of a multiset case study that we proved linearizable.

Intra-Process Reordering vs. Linearizability

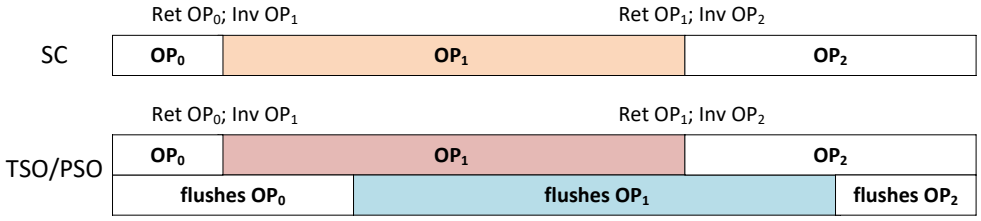

Figure 6.6: Overlapping of method execution and intra-process reordering.

One effect that we also need mention here is the intra-process reordering of linearization points. In essence it is the effect that allows writes to be reordered with later reads. With the above approach, it is possible that LPs of different operations are reordered within a process. In Figure 6.6, we visualize operation execution of a single process for SC, as well as for TSO or PSO. Time passes from left to right. Invokes and returns are marked. For SC, the bar represents the execution of all program statements including their effect to the shared memory. At the bottom, the operation execution is separated into two bars. The top bar represents the execution of all program statements of the operation. The bar below it, represents the delayed execution of writes, i.e., flushes. For the sake of completeness, we would like to

mention that PSO allows single writes to be reordered with each other, which we omit in the figure. Due to the overlap of operations, it is possible that operations linearize in a different order than the program order. This can happen under the circumstance that the earlier operation linearizes with a flush to a certain memory location, while the later operation linearizes with a read of a different memory location.

The proposed consistency checks against the abstract atomic specification do not recognize such a reordering. However, the consistency checks still require both, abstract and concrete, states to be consistent at an LP. Thus, even in this case the linearization order is equivalent to a sequential order. TSO-to-SC linearizability [GMY12, BDG13] as well as the original definition of linearizability [HW90] allow linearization points to be only between invoke and return of a method. TSO linearizability [DSD14, DSGD17] allows the linearization points to be reordered within a process under the above conditions. For the latter definition, our approach is the first and to the best of our knowledge the only model checking approach. Please note that the potential for intra-process reordering of LPs is easy to identify in the code, if the LPs are known. It can occur under the following circumstances:

1. An operation has an LP that corresponds to a flush, which is not guaranteed to happen before the operation's return statement. Such a guarantee can be provided by a fence or a CAS instruction.

2. An operation has an LP corresponding to a read of a memory location that is different from the other operation's written memory location, s.t., both can be reordered.

Thus, even though our approach does not detect the intra-process reordering, it can be easily spotted in the code and avoided, if necessary. It could be necessary, if one wants to establish a stricter definition of linearizability (TSO-to-SC).

6.1.2 Experiments

In this section, we present our experiments. The experiments do not only evaluate our verification approach with abstract atomic specifications presented in this chapter, but also our reduction from Chapter 3 in comparison with the more common approach of using operational memory models.

We have selected several concurrent programs for the experiments. These range from rather simple mutual exclusion algorithms [BL80, Dij68, Pet81, Szy88, Lam74] over actual concurrent data structures such as a stack [Tre86] and a queue [ABP98] to a transactional memory implementation [DDS$^+$10]. All of these algorithms are typical concurrent programs and thus are candidates for bugs due to weak memory

models, which would not be found with verification techniques that assume SC. Some of these algorithms are known to be incorrect under weak memory models and thus, require fences. To the best of our knowledge, we are the first to analyze the transactional memory implementation from [DDS+10] under weak memory models (first published in [TW16]), which requires a fence for PSO in order to be correct.

We implemented all of the programs in C and C++ while following the pseudo code as available from their respective publications and compiled them to LLVM IR[1]. For all of our experiments, we used WEAK2SC to generate the Promela models. For each case study, we generated several program versions. One version relies on the operational memory models that we provide for SC, TSO and PSO. For this type of model, it was sufficient to exchange the underlying memory model in order to analyze it with the respective memory model semantics. The programs generated from store buffer graphs had to be generated for each memory model separately, because the store buffer graph changes with the choice of the memory model. Thus, we have several versions of a transformed program, one for each memory model.

If we were able to find bugs in a program, then we added fences as we found would be necessary to correct the programs. We tried to use as few fences as possible, but cannot claim that the number of fences used is minimal. In order to be able to make such claims, we would have to apply additional analyzes such as [KVY12]. Please note that all of the programs are correct under SC. Thus, we knew that each bug can be fixed by adding fences to these programs unless we got something wrong in our implementation of these programs. Luckily, the latter was not the case as our experiments have shown. However, adding a fence to a program changes its behavior and thus, we had to regenerate all of the above program versions as fenced versions. Because we were able to find weak memory model related bugs in all except one (the Treiber stack) of the case studies, we report on unfenced and fenced versions of the program models.

Algorithm Correctness

Table 6.1 shows the correctness results for the different versions of the programs that we used for our experiments. A checkmark entry means that we were not able to find counterexamples, given the instrumentation of the program with an abstract specification at its LPs as explained in the previous section. A cross indicates that we were able to find a counterexample using this method. A *uwl* entry denotes the cases for which we were not able to perform an experiment, because the reduction was not applicable due to an unfenced writing loop (see Section 3.1). The most left column names the algorithm. The brackets behind the name denote the memory

[1] LLVM Compiler Framework v3.1

Algorithm	tso	tso2sc	pso	pso2sc
Burns [BL80]	×	×	×	×
Burns (TSO)	✓	✓	✓	✓
Dekker [Dij68]	×	uwl	×	uwl
Dekker (TSO)	✓	✓	✓	✓
Peterson [Pet81]	×	×	×	×
Peterson (TSO)	✓	✓	×	×
Peterson (PSO)	✓	✓	✓	✓
Lamport bakery [Lam74]	×	×	×	×
Lamport bakery (TSO)	✓	✓	×	×
Lamport bakery (PSO)	✓	✓	✓	✓
Szymanski [Szy88]	×	×	×	×
Szymanski (TSO)	✓	✓	✓	✓
Arora queue [ABP98]	×	×	×	×
Arora queue (TSO)	✓	✓	×	×
Arora queue (PSO)	✓	✓	✓	✓
Treiber stack [Tre86]	✓	✓	✓	✓
TML [DDS$^+$10]	✓	✓	×	×
TML (PSO)	✓	✓	✓	✓

Table 6.1: Verification results for the transformed programs (tso2sc, pso2sc) and based on an operational memory model (tso, pso). Brackets state the memory model for which a program was fenced.

model for which we added fences in order to make it robust against the respective memory model. No brackets, except for the reference, refer to an algorithm without any additional fences. The columns on the right denote the type of experiment and memory model choice. The column *tso* (resp. *pso*) shows our results for the algorithm version with an underlying operational memory model. The column *tso2sc* (resp. *pso2sc*) shows the results with the algorithm versions obtained by reduction via store buffer graphs.

Our experiments show consistent results among transformed programs and those based on operational memory models. One exception is the Dekker mutex algorithm [Dij68], which could not be transformed via store buffer graph into an SC program due to an unfenced writing loop. It is the only case study with such a loop in our set of case studies and it requires a fence in the previously unfenced loop in order to be correct. The latter is shown via our experiments with operational TSO and PSO memory models. Please note that the operational approach in this case also was only successful, because the test scenario performs only one iteration of acquire and release. The latter ensures that finite store buffers are sufficient for the verification under TSO and PSO. Otherwise, the verification would also not succeed due to quickly filled buffers. One could conclude that unfenced writing loops can be an indicator for erroneous behavior in concurrent data structures under weak memory. However, we would like to be careful with such generalizations, because it is easy to add unfenced writing loops to any given algorithm without changing

its core behavior. Such an algorithm could work as intended, but our reduction would not be applicable anymore due to the added loop. Fortunately, most of the concurrent algorithms are highly optimized and thus, do not contain obsolete code. In order to decide whether unfenced writing loops are a true indicator for erroneous behavior in concurrent data structures, more case studies that have such loops have to be analyzed. The latter is out of scope of this thesis. The remaining case studies performed as expected, i.e., iff we were able to find a counterexample in a reduction based check (tso2sc or pso2sc), then we were also able to find it using our operational memory models (tso or pso).

Among the mutex algorithms, Burns, Dekker and Szymanski do not require any additional fences under PSO, if they are already fenced for TSO. In each of the algorithms, each process writes to only one memory location and thus, the mutual order of writes by each process is preserved. In contrast, processes in the algorithms Peterson and Lamport's Bakery do write to different memory locations and therefore require an additional fence that preserves their order. Similarly, the order of writes in the *pushBottom* operation of the queue by Arora et al. have to be preserved. Otherwise, a new element in the queue becomes visible to other processes before the value of the element is written to the memory. The latter allows other processes to steal an element containing an inconsistent value. The transactional memory (TM) implementation TML also requires an additional PSO fence. Transactions are seemingly atomic and can consist of different reads and writes to different memory locations. A transaction always finishes with a commit. A commit allows other processes to observe the written values and makes the changes permanent. TML requires a fence in order to ensure that the values written by a transaction are flushed before the write corresponding to the commit is flushed. Otherwise, the commit would signal to other transactions that they can read and write to the memory while more changes by the committing transaction are still being flushed. The latter breaks the atomicity of a transaction as it is intended by TM algorithms [HM93]. Actually, the desired correctness condition for TM algorithms is opacity [GK08], which takes the different aspects of a transaction into account, e.g., that a transaction is sequenced into a begin operation, followed by reads and writes and finally a commit or abort operation. An abstract memory combined with atomic transactions enabled us to check for atomicity of transactions in our experiments anyway and revealed that the algorithm requires a fence for PSO. In the Appendix B, Fig. B.3, we also show our TML implementation (including the required fence) of two example transactions that we used in our experiments.

The most robust algorithm in our case studies happened to be the Treiber stack, which required no additional fences, neither for TSO nor for PSO. It does not require additional fences, because it relies on CAS instructions. CAS instructions have a

built-in fence semantics and thus, ensure that the store buffer is emptied before they are executed atomically. Please note that in our previous publication in [TW16], one of our experiment tables contains an accidental entry that suggests, we added a fence for PSO. This was not case and the entry is incorrect. If a CAS instruction would not have built-in fence semantics, then the Treiber stack certainly would require a fence under PSO. Otherwise, the algorithm would suffer from problems similar to those in Arora et al. queue.

Performance Comparison

One aspect of the research question **RQ2** is whether verification performance is improved by applying our proposed reduction compared to using an operational memory model. In order to answer this question, we conducted experiments with identical exploration scenarios and compared them with each other. Table 6.2 shows our first batch of results, particularly for the verification of programs under TSO. A second one with the same programs and scenarios under PSO can be found in Table 6.3. Out of the set of different program versions that we obtained from our previous results, we took those, for which we were not able to find counterexamples under PSO. By choosing these versions, we ensure that the exploration always reveals the complete state space under TSO and PSO and the different explorations are comparable in exploration effort. An exploration yielding a counterexample can be fast if it is lucky, but it may also have to explore almost the entire state space in order to find that counterexample. A comparison of such exploration runs would not be conclusive, which is why we consider only full state space explorations.

The tables provide the size of the program in terms of the number of instructions per method in column "#i". If several methods exist in the implementation, then their respective numbers of instructions are separated by slashes. The remaining columns separate into results obtained from exploration with an underlying operational memory model (tso and pso) and the programs transformed into SC programs (tso2sc and pso2sc). For each run, we provide the number of explored states (#s), the state vector (v) and the time in seconds (t) as reported by the model checker Spin[2]. All of our experiments were performed on an Intel Core I5 4690 processor with 3.5 GHz and 3 GB of memory dedicated to Spin. The times in the tables are pure exploration times as reported by Spin, i.e., they do not include transformation, compilation and manual instrumentation. Each experiment was repeated five times. The table entries are the respective average times. Best results are given in bold font.

The number of instructions per method provides an impression of size of the original programs and their respective encoding in the operational approach. Since

[2] Spin Version 6.2.3

		tso			tso2sc			
Algorithm	#i	#s	v	t	#n	#s	v	t
Burns (TSO) [BL80]	3/19/11	464	172	≈ **0.000**	4/30/12	**425**	**72**	≈ **0.000**
Dekker (TSO)	33	2193	204	0.006	56	**1237**	**120**	≈ **0.000**
Peterson (PSO)	24	2241	188	0.004	30	**1465**	**104**	≈ **0.000**
Lamport Bakery (PSO)	49	12.3 k	248	0.022	59	**7645**	**168**	**0.01**
Szymanski (TSO)	32/35	21.9 k	204	0.056	59/68	**12.2 k**	**124**	**0.012**
Arora Q. (PSO)								
uuuoouo‖sss	9/15/30	186.7 k	428	0.456	13/18/49	**147.2 k**	**316**	**0.202**
uououo‖ss‖ss	9/15/30	24.6 M	492	87.94	13/18/49	**7.90 M**	**344**	**12.58**
Treiber Stack								
uouo‖uouo	13/16	168.9 k	360	0.398	16/29	**115.8 k**	**252**	**0.152**
uuuooo‖ ooouuu	13/16	2.365 M	448	6.66	16/29	**1.702 M**	**340**	**2.816**
TML (PSO)								
wr ‖ wr	30	2136	260	≈ 0.000	55	**1298**	**152**	≈ **0.000**
wrr ‖ wrr	37	**3651**	276	0.008	86	4290	**168**	**0.002**
IRIW: w ‖ w ‖ rr ‖ rr	22/23	17.6 M	356	63.34	33/38	**3.717 M**	**192**	**5.838**

Table 6.2: Verification results for full state space exploration error: #i are lines of LLVM IR instructions ("/" separated for each method); #n number of nodes in the sb-graph ("/" separated for each sb-graph); #s the number of states explored t is the time in seconds.

the transformed SC programs grow in size compared to the original program, we also provide the number of nodes of the respective store buffer graphs. These numbers are actually a good indicator for the amount of behavior that is present only due to weak memory models. Please note that LLVM IR code is already organized by the compiler into control flow blocks. Because an SC store buffer graph is essentially a control flow graph, it has $\#i + 1$ nodes for a program with $\#i$ instructions. If the actual number of nodes exceeds $\#i + 1$, then it can only be due to additional nodes in the store buffer graph, which again can only be due to combinations of program locations with non-empty symbolic store buffers.

Throughout all of our experiments the reduction approach outperformed the usage of operational memory models. In all but one of our experiments, the model checker had to explore less states in the reduced SC programs. Without exception, the checks of the reduced SC programs were faster than those based on operational memory models. The speed up goes up to a magnitude in our experiments and grows with increasing difficulty of the experiments. The latter is mainly driven by the number of concurrent processes and the number of method calls each process executes consecutively.

The mutual exclusion algorithms Burns, Dekker, Peterson, Lamport Bakery and Szymanski posed barely a challenge to the model checker using both of our methods. The scenario that we verified for all of these case studies consists of two processes, each attempts to acquire the mutex and then releases it once. For some of the mutex

		pso			pso2sc			
Algorithm	#i	#s	v	t	#n	#s	v	t
Burns (TSO) [BL80]	3/19/11	464	412	$\approx$ **0.000**	4/30/12	**425**	**72**	$\approx$ **0.000**
Dekker (TSO)	33	2332	340	0.008	60	**1480**	**120**	$\approx$ **0.000**
Peterson (PSO)	24	2241	324	0.004	30	**1465**	**104**	$\approx$ **0.000**
Lamport Bakery (PSO)	49	12.3 k	384	0.052	59	**7645**	**168**	**0.012**
Szymanski (TSO)	32/35	21.9 k	340	0.088	70/82	**15.3 k**	**124**	**0.02**
Arora Q. (PSO)								
uuuoouo‖sss	9/15/30	185.3 k	812	0.894	13/18/52	**147.2 k**	**316**	**0.21**
uououo‖ss‖ss	9/15/30	24.0 M	1068	184	13/18/52	**7.9 M**	**344**	**12.62**
Treiber Stack								
uouo‖uouo	13/16	172.3 k	728	0.842	16/31	**126.8 k**	**252**	**0.174**
uuuooo‖ ooouuu	13/16	2.46 M	816	13.62	16/31	**1.9 M**	**340**	**3.326**
TML (PSO)								
wr ‖ wr	30	2183	428	0.004	59	**1450**	**152**	**0.002**
wrr ‖ wrr	37	7938	444	0.028	111	**5559**	**168**	**0.008**
IRIW: w ‖ w ‖ rr ‖ rr	22/23	18.7 M	692	132	42/58	**4.18 M**	**192**	**6.884**

Table 6.3: Verification results for full state space exploration error: #i are lines of LLVM IR instructions ("/" separated for each method); #n number of nodes in the sb-graph ("/" separated for each sb-graph); #s the number of states explored t is the time in seconds.

algorithms, Spin reported 0 time for the verification, i.e., the time required for the full state space verification is below measurement threshold. For some cases 0 time was reported for all of the five explorations. We use the entry ≈ 0.000 in the table for these cases in order to highlight that the actual time of the exploration must be close to zero. Please note that Spin reports times only with at most 10ms or two digit accuracy and that the entries with values below 0.01 are the resulting average of our five runs.

The difference between both approaches in verification times becomes more obvious with the size of the programs or the verification scenario. We did not extend the scenario that we used to check the mutex algorithms, but rather focused on actual concurrent data structure implementations as intended from the beginning of this thesis. The work-stealing queue by Arora et al. and the Treiber stack are two such implementations. In contrast to mutex algorithms, these algorithms allow for infinite abstract and concrete states. However, we can still verify finite scenarios by providing a set of processes, each executing a predefined sequence of operations. The tables denotes the scenarios in a short notation. For the Arora et al. queue, we use "u" for a *pushBottom* call, "o" for a *popBottom* call and "s" for a *popTop* call. The parameter to *pushBottom* is a unique integer value for each call. We denote parallel composition as ‖. For instance, a scenario uu‖s denotes two processes, where the first process executes two consecutive *pushBottom* calls and the second process a *popTop* call. We use a similar notation for the Treiber stack, where "u" corresponds

to a *push* call and "o" to a *pop* call. The TML algorithm differs from the previous two in the sense that we need consider transactions. In our scenarios, each process executes one transaction. A transaction can perform arbitrary modifications to the shared memory, which have to appear atomic. The explored scenarios represent the litmus tests from Chapter 2 where each transaction performs writes ("w") and reads ("r") of one of the processes from the underlying litmus test. Thus, if the TML algorithms is not properly synchronized, we should be able to observe results that are identical to the ones these litmus tests aim to detect. In fact, we were able to identify a fence that is required under PSO.

From both tables, we can observe smaller verification times with reduction-based approach for all of our experiments. If we ignore the explorations below time measurement threshold, then the speedup provided by the reduction over an operational memory model is roughly a factor of 2 to 10. The largest speedup was observed for the scenarios with three or four processes involved. Please note that even for the TML scenario, where the reduction approach had to explore more states than the operational approach, we were able to observe a speedup.

We think there are several reasons for the observed speedup in our experiments. One obvious reason for it is that the model checker had to explore less states in most cases. This is likely due to fewer variables used in the reduced programs, while the operational memory model requires some auxiliary variables as part of the encoding of its behavior. Furthermore, fewer variables result in a smaller state vector. The latter allows for faster state coverage checks during exploration, since less information has to be taken from memory in case of cache miss. Furthermore, more states can be held in the processor cache, which should result in less cache misses on average. Please note that a cache allows for a significantly faster access to data than the memory, if the requested data is present in the cache. Another benefit from the comparably small state vectors of the reduction-based approach is that the exploration is capable of exploring more states before it runs out of memory. Another possible argument for the observed speedup is the partial order reduction [Val89] provided by Spin, which was active throughout our experiments. It avoids exploration of states that are present due to commuting events. For instance, two concurrent processes could modify one of their local variables. The order of these modifications does not matter since they will end up in the same state. Thus, it is sufficient to explore only one of the possible orders. We think, it is also possible that the partial order order reduction as implemented by Spin performs better with reduced programs, because they do not separate program and semantics into two processes as our operational approach does.

Please note that in [TW16] we used simple assertions in our experiments in order to compare verification performance between operational memory models and

the reduced programs. The experiments in this section used an abstract specification as proposed in the previous section. In contrast to simple assertions, the abstract specification has to be carried throughout the whole exploration as part of the state. Thus, the results presented here differ from our previously published results. You will also notice that we do not present results for the fib_bench case study [sv-16]. The reason for it is that is not a data structure implementation and thus, there is no abstract specification for it. Instead, it is just a rather simple concurrent program with assertions that have to hold.

6.1.3 An alternative Idea - History Checking

An alternative approach for the verification of linearizability under any memory model is to record all possible histories of a program during exploration and then to check whether the histories are linearizable while ignoring the rest of the program. The idea is to check concurrent histories against sequential histories. For this, we need a set of histories produced by an implementation. The set can be gathered by logging events in test runs or systematically through state space exploration. In addition, we need to define a checking procedure, which determines whether a concurrent history is linearizable. For linearizability, we have to find a witness history that is (1) sequential, (2) equivalent to the concurrent history and (3) respects its real-time order. An execution (or more accurately its history) can be considered linearizable, if we can find a witness for it. If there exists a concurrent history for which no witness history can be found, then it is a counterexample for linearizability of the implementation.

The idea, we follow here, is very general and in its spirit combines the ideas from [VYY09, BDMT10]. Vechev et al. [VYY09] define history based linearizability checks using abstract atomic specifications in Spin, where the abstract specifications are similar to those in the previous section. Burckhardt et al. [BDMT10] use the implementation under analysis in order to generate all possible sequential histories. Later, these are used as a set of potential witnesses for all concurrent histories. In contrast to the above, our approach is to use Spin for the exploration of all possible concurrent histories while using actual sequential data structure implementations for the generation of witness histories in the linearizability check. Both of the above approaches assume an SC memory model. However, a memory model influences only the set of possible concurrent histories. If a weak memory model has an impact on the correctness of an implementation, then it must be observable via a non-linearizable history. This is also how the approach aims to find bugs due to weak memory models.

The history based check has been implemented by one of our students (Katharina

Dridger) during her bachelor thesis [Dri14]. She implemented checks for Linearizability [HW90] and also for Quiescent Consistency [HS08]. The approach can also be extended to other correctness conditions that rely on sequential histories as witnesses. In the following, we elaborate on this approach.

Logging of Concurrent Histories

A model checker like Spin explores the state space of a Promela model, in our case a data structure implementation. Thus, the explored state space also includes all possible interleavings of the program. With all possible interleavings, we also get all possible concurrent histories for an explored test scenario.

Unfortunately, these histories are difficult to extract from the explored state space, which is why the program must be instrumented. The instrumentation of the program is straightforward in the sense that we need to log the invoke and response events of all operations, including parameter and return values. Thus, at each operation invocation we log an invoke event and at each operation return we log a return event. Each event is a tuple $(eType \times eID \times P \times Op \times Param)$, where $eType = \{inv, res\}$, $eID \subseteq \mathbb{N}$ is a unique event identifier, $P \subseteq \mathbb{N}$ the process identifier, $Op \subseteq \mathbb{N}$ an operation identifier and $Param$ one or more input or output values. Please note that we can add other events as long as we make sure that the instrumentation is correct. Thus, the approach is adequate for checking all three adaptations of linearizability (see Chapter 5) to weak memory models as well as the original linearizability definition [HW90].

Promela does not allow for dynamically sized data structures like lists. Thus, we need to define an array h in our model that will store the events and particularly their order of appearance in an execution. This array represents the concurrent history and it is the only variable that our later checking procedure will consider from for any execution of the implementation. The length of the array $\#h$ must be $\#h \geq 2n$, where n the number of method calls in the explored test scenario. Since h is finite, we can only log finitely many method calls with this approach.

In principle, this is all we have to do in order to gather all possible concurrent histories of an implementation in a particular test scenario. However, we also need a condition to be checked for each of the logged histories and which determines whether the history is linearizable or not. In particular, we also want it to be validated during the exploration and not afterwards, since finding a non-linearizable history renders the rest of an exploration obsolete. Figure 6.7 provides an overview of the separate steps in this approach. The linearizability check is implemented in C as an extension to Spin and must be added as an assertion in the Promela model. The check must be triggered after all processes have finished their work. The latter can

be achieved by flags for each process that are set to true once the process is finished. The reason why the correctness check should be triggered after all processes are finished is because it is expensive. It deals with permutations of events as the check has to find an equivalent sequential history for the concurrent one. If it fails to find one, the assertion evaluates to false and stops the state space exploration in Spin with a history as a counterexample to the correctness condition. Consequently, the check considers only complete histories and no prefixes of it.

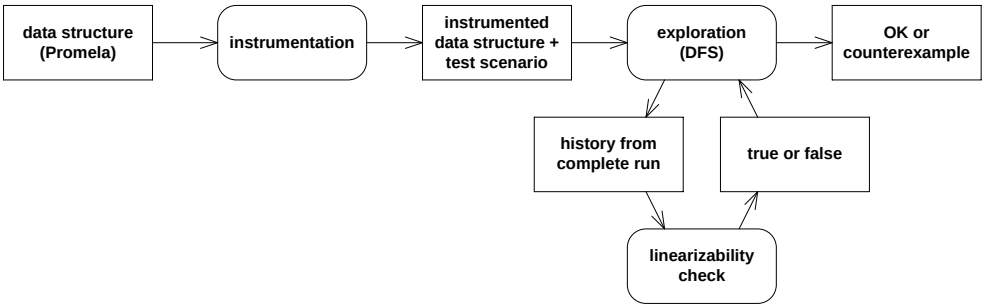

Figure 6.7: Overall history checking procedure. Exploration triggers history checks via assertion whenever all processes have run to completion.

Checking Histories for Correctness

Checking whether a single concurrent history is linearizable or correct w.r.t. some other correctness condition is not very difficult. In principle, it is the search for a permutation of events in the concurrent history, s.t., characteristic constraints of the checked correctness condition hold in the permutation. If such a permutation can be found, then it is the witness for the correctness of the concurrent history. More importantly, the permutations we search for are sequential executions of the implemented data structure. That means, we can use actual sequential implementations in order to generate all possible sequential histories by using the same inputs as in a concurrent history obtained during exploration. Furthermore, it is sufficient to permute pairs of events corresponding to an operation, because an invoke is always followed by its return in a sequential history.

With this in mind, our student Dridger implemented a standard library of simple sequential implementations of a queue, a stack, a set and a multiset as an extension to Spin. On top of them, a sequential history generator was implemented. Figure 6.8 shows the major steps of the implemented linearizability check. As an input, it takes the concurrent history ch as provided by the instrumented Promela model and generates all sequential permutations with the given inputs from ch. The result

is a set HS_{pot} of potential linearizability witnesses. However, this set has yet to be filtered according to the other two requirements for linearizability, which require it to be equivalent to ch and respect the real-time order of it. Filtering for equivalence is simple, because we only have to check whether the output of return events is identical to what was stated in ch. Since the input values of invoke events were already considered in the generation HS_{pot}, only the output values of each response event have to be compared with those in ch. Whether the real-time order is preserved by a candidate $hs \in HS_{pot}$ or not is also simple. For each event hs we only need to check whether it violates any ordering that needs to be preserved from ch. Please note that the real-time order also covers the sequential order of operations by each process. Both checks can be performed in $O(n^2)$ per potential witness history. Throughout the above filtering procedure of HS_{pot} it is possible that we end up with an empty set of witness histories. If HS_{pot} is empty, then ch is a non-linearizable history and the checking procedure returns false. At this point, the exploration in Spin is stopped due to the assertion containing the linearizability check and the history ch is reported. Otherwise, if the set HS_{pot} remains non-empty at the end of the filtering steps, the exploration can continue, because each of the histories in HS_{pot} is a linearizability witness for the currently explored history ch.

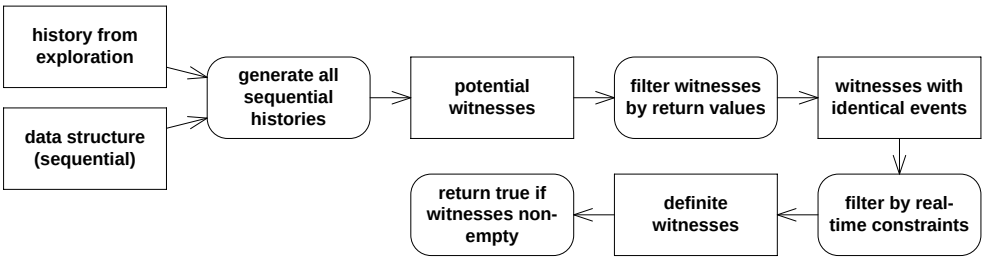

Figure 6.8: Stepwise checking procedure of a concurrent history. Sequential data structure and the history are input parameters to the check.

Please note that the overall procedure of this approach works best with a depth-first-search (DFS) state-space exploration, because our Promela models trigger the check when all processes finished their work. DFS ensures early validation of histories in the exploration procedure and thus, enables the exploration to abort early in case of detecting a non-linearizable history. In a breadth-first-search (BFS) exploration, the linearizability check would be triggered only after a large fraction of the state-space was already explored. The latter can be a waste of time and resources if the data structure is not linearizable and can be avoided by exploration in DFS manner.

Weak Memory Models and Discussion

As the reader may have noticed, weak memory models do not play an important role in this approach. The instrumentation is mostly independent of the underlying memory model, because only operation invocations and responses are taken into account. In general, the approach can be applied for each of the linearizability definition adaptations from Chapter 5.1 and also other correctness conditions. Two of the latest versions of linearizability [BGMY12, DSGD17] also take flushes in terms of special events into account. An instrumentation to log flushes as part of the history is considerably more difficult, because one has to identify statically when a particular flush occurs. The latter highly depends on how the weak memory semantics are modeled, be it as part of the analyzed program or in terms of a separate operational memory model. In either case, expertise is required. However, most correctness conditions consider invoke and response events only. For such correctness conditions this approach is practical and simple to apply.

The only manual task in our approach is to instrument the data structure properly and to define test scenarios for the exploration. The Promela model of the data structure can be generated with the help of Weak2SC. The approach does not require any knowledge of linearization points and thus also suits non-expert users. However, it suffers from some problems with complexity and performance, which are difficult to avoid.

First of all, the history enlarges the program state in this approach. Thus, it takes more memory to encode a state as a bit vector. Consequently, a smaller number of states can be explored than without a history as part of the program state. This can have a significant effect on the explorable state-space, especially if the actual program does not have many variables. In such cases, most of the state vector will be required for the history. Furthermore, we have to make sure that test scenarios are finite. Otherwise, the exploration would run out of memory quickly just by filling the history and we could not complete any exploration. Spin does not allow for dynamic data structures anyway and that is also the reason why we use an array to encode the history in our Promela models. In contrast to our previous approach in Section 6.1, this approach can only check finite executions instead of finite state programs, which can have infinite executions.

A second limiting factor to this approach is the number of times the linearizability check is triggered due to the interleaving of different processes. The state-space exploration in a model checker like Spin explores all possible interleavings of concurrent processes. The latter are of course more fine grained than a history that consists of invoke and response events. An exploration will likely reveal many program executions that may lead to different states, but which are represented by

the same history. A linearizability check is triggered by an assertion in the Promela model whenever the exploration reveals a new state at the program location of the assertion. Many of the histories will be identical. In order to avoid checking identical histories multiple times for a corresponding linearizability witness, we can memorize previously checked histories. Obviously, a memorization of previously checked histories would also come at the price of even more memory consumption and would further decrease the number of states that can be explored.

Generally, we can say that the history checking approach does not scale well with the size of the test scenarios. In particular, this is the case, because the search for a linearizability witness is a search for a permutation of history events. The latter has factorial complexity. The experiments by our bachelor student Dridger [Dri14] had less than ten operations (all processes combined) and at most four concurrent processes before they ran out of memory. The checked data structures implementations were the Michael and Scott queue [MS96], the Treiber stack [Tre86], QC-Queue [DDS+14] and a multiset implementation [EQS+10]. Her experiments were conducted under SC semantics and thus, did not even take the increase of possible interleavings due to weak memory models into account. However, her experiments also show that small test scenarios can be verified with this approach.

Besides being a simple but not well scaling approach for checking linearizability, it is also an approach that is easy to extend in two ways: First, more data structures can be added to the mechanism for generating sequential histories, simply by providing their implementation and a definition of its possible invoke and response events (per operation). This would allow for linearizability checks of other than the provided data structures. A second way to extend this approach is by providing a different history checking procedure, e.g., for different correctness conditions or in order to experiment with different optimizations. Our student Dridger implemented a checking procedure for quiescent consistency [HS08]. Thereby, she was one of the first to provide a model checking approach for quiescent consistency. The history check is a separate module of our extension to Spin and can be easily exchanged with other checking procedure implementations.

6.2 Proving Linearizability under Weak Memory Models

As a proof approach, we suggest the linearizability theory developed by Derrick et al. [DSW07]. It was stated for the original definition of linearizability [HW90] and thus considers only sequential consistency [Lam79]. However, the more recent definitions of linearizability by Gotsman et al. [GMY12] and the more general ver-

sion of it by Batty et al. [BDG13] can be viewed as generalizations of the original linearizability definition that do not change its essential meaning. As explained in the previous chapter, the boundaries for the linearization points (have to lie between invoke and response of an operation) remain the same in these definitions, even under consideration of weak memory semantics. Thus, we can apply existing SC proof techniques for linearizability as long as the behavior that we consider incorporates the weak memory semantics of our target architecture. The latter is achieved by our reduction from Chapter 3 or it can be part of a memory model specification (in the style of an operational model) that reveals all possible program transitions during a proof.

We have chosen the proof theory and proof approach by Derrick et al. [DSW07, DSW11a], because it is fully specified in the theorem prover KIV [EPS+14] and was also proven to be sound and complete [DSW11a, SWD12]. Furthermore, we were able to gain some expertise with the theory and the theorem prover in proving earlier case studies correct [TWS12, TTSW14]. The latter can be crucial when proving programs correct as the proofs are inherently complicated. The linearizability theory comes with two versions of proof obligations, local [DSW11b] and global [DSW11a, SWD12] ones, which is also the reason why Weak2SC provides two types of data type encodings for KIV. These proof obligations can be instantiated within the theorem prover for any given case study and then, with the help of the theorem prover, be proven correct.

In this section, we exemplify the application of the proof approach to our running example, the Arora et al. work-stealing queue [ABP98]. Particularly, we want to identify the impact of our proposed reduction on the proof effort compared to an operational style memory model specification. The latter shall help us answer **RQ2** as stated at the beginning of this chapter.

6.2.1 Overview

The proof methodology proposed by Derrick et al. [DSW11a, DSW11b] shows linearizability via a non-atomic refinement proof between a concrete and an abstract data type. Both, the concrete and the abstract data structure, are formalized as instances of an abstract data type in Z [DB14]. Essentially, the methodology shows that all histories of the concrete data structure are in a subset relation with those of the abstract data structure. Such proofs are usually achieved by showing forward or backward simulations between two data types [LV95]. Both types of simulations are sound and in case of the backward simulation also a complete proof technique [SWD12] for linearizability. In this chapter, we present an instance of a backward simulation from [DSW11b], which in contrast to the forward simulation in [DSW11a] also can

handle potential linearization points. Even though our case studies do not have potential LPs, we chose these proof obligations as a precautionary measure just in case we might encounter an LP that we were not aware of. Backward simulations are often more difficult to prove than forward simulations, because they can require reasoning about the past of an object. The latter can be counterintuitive. However, often they cannot be avoided as in case of potential linearization points. We assume that the concrete data structures are results of the program transformation that we introduced in Chapter 3. Thus, their specifications include all the behavior that can be caused by weak memory models to them.

The methodology by Derrick et al. [DSW11b] provides local proof obligations that can be instantiated for case studies. If these proof obligations can be proven to hold, then this implies that there exists a certain type of simulation between the concrete and the abstract data type. The latter does not have be stated explicitly in order to prove the proof obligations, but. it implies that the concrete data type is a non-atomic refinement [DW05] of the abstract data type. That is, one of the many concrete operations that implement an abstract operation simulates the abstract operation while all other concrete operations are stutter steps. The step simulating the abstract operation is what we refer to as the linearization point (LP). At last, the non-atomic refinement implies that the concrete data type is linearizable w.r.t. the abstract data type. The proof obligations as well as their soundness proofs [DSW11a, DSW11b, SWD12] are formalized in the theorem prover KIV [EPS+14] and are ready to use. Thus, all we have to do in order to prove an implementation linearizable is to prove the local proof obligations for it.

The Local Proof Obligations for Linearizability

A data type $DT = (S, SInit, (OP_{p,i})_{p \in P, i \in I})$ consists of a state space S, a set of initial states $SInit$ and a set of operations $(OP_{p,i})_{p \in P, i \in I}$ that are executed by some process p. We use I (and later also J) to index operations. We prefix these sets and definitions with an "A" for an abstract (ADT) and "C" for a concrete (CDT) data type. Please note that we decompose the concrete state CS into globally shared state GS and a local state function $lsf : P \to LS$ that maps processes to their local states LS. This enables reasoning and definition of the proof obligations in a thread modular way.

An operation AOP_i (resp. COP_j) is a predicate over a state $as, as' \in AS$ (resp. $gs, gs' \in GS, ls, ls' \in LS$), where the unprimed variables refer to the state before the operation and the primed variable refers to the state after.

Figure 6.9 visualizes the refinement relation that the proof obligations from [DSW11a] prove. At the top, we see an abstract operation $AOP(in, out)$ with input

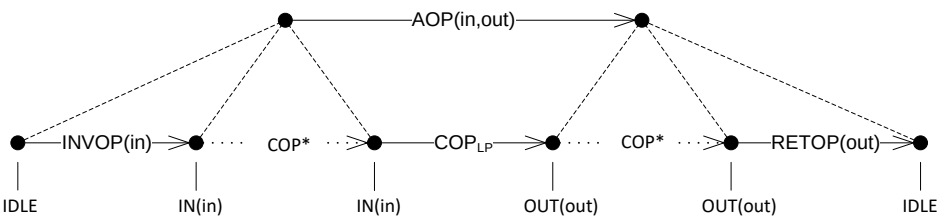

Figure 6.9: Visualizing non-atomic refinement and the linearization status.

parameter in and a result out. We assume $in, out \in V$ to be from some value domain V. The operation brings us from one abstract state to the other. At the bottom, we see a sequence of concrete operations which together implement the abstract operation. These start always with an invoke operation $INVOP(in)$ and end with a final operation $RETOP(out)$. Both operations have identical input and output parameter to the abstract operation. In between, there are possibly many concrete operations COP, out of which one is the linearizing operation COP_{LP} or simply the LP. Abstract and concrete states must be related by a refinement relation $R \subset AS \times (GS \times LS)$ which is visualized by the dashed lines. In the proof obligations, the refinement relation is further split into an abstraction function $Abs : GS \to AS$ and an invariant predicate INV over $(GS \times LS)$. In principle, all the concrete states before COP_{LP} must be related to the abstract state before $AOP(in, out)$ and all the concrete states after COP_{LP} must be related to the abstract state after it. Abstractly, all of these steps, except for COP_{LP}, are *skip* steps, which do not modify the state. The linearization status (denoted at each concrete state) describes whether an operation has linearized already and will return with output out or whether it has yet to linearize with its input in.

The linearization points must be known in advance, because we need to define which of the concrete operations is the LP. That is achieved by the linearization status, which is represented by a status function. The function $status : LS \to STATUS$ defines whether a process is $IDLE$, whether it has not yet reached its LP ($IN(in)$) or whether it has reached its LP already and only needs to return ($OUT(out)$). An additional status value $INOUT(in, out)$ represents a state, in which an operation potentially may have linearized already, but its outcome can still be influenced by other processes and thus is not definite, yet. The latter can only appear in operations with potential LPs.

The local proof obligations are structured into three important parts. The first and most important part defines the non-atomic refinement between concrete and abstract operations. Since the first part is local, we also need to make sure that other

processes do not interfere with the local assumptions of each process. Thus, the second part of the proof obligations deals with interference freedom and disjointness among processes. The third and last part covers initialization, in which invariants and the refinement relation must be established for initial states.

Non-atomic Refinement The proof obligations for the non-atomic refinement split concrete operations into essentially four different cases. Each case has its own lemma that must be proven. The cases correspond to the possible linearization status of an operation and particularly to how the status can change from one status to another. The first lemma must be proven for invoke operations:

$$\begin{aligned}&(\forall\, as \in AS, gs, gs' \in GS, ls, ls' \in LS \bullet INVOP(in, gs, ls, gs', ls')\\ &\qquad \wedge Abs(gs') = as \wedge INV(gs, ls) \wedge status(gs, ls) = IDLE)\\ &\Rightarrow Abs(gs) = as \wedge INV(gs', ls') \wedge status(gs', ls') = IN(in)\end{aligned}$$

where $INVOP(in, gs, ls, gs', ls')$ is the invoke operation and it must start at a state gs, ls with input in. The invariant $INV(gs, ls)$ is assumed to hold initially. The invoke operation modifies the state to gs', ls'. As we deal with an instance of backward simulation, we assume the abstraction $Abs(gs') = as$ to hold for the modified state gs' and we have to prove that it was also valid initially $Abs(gs) = as$. Furthermore, the linearization status must have been $IDLE$ at start. After the invoke, the new status must be $IN(in)$, i.e., the operation must not have linearized yet. Also, the invariant must still hold $INV(gs', ls')$ in the new state.

The second case must be proven for all internal operations COP_j that are neither invoke, nor response operations and which lie before the linearizing operation:

$$\begin{aligned}&(\forall\, as' \in AS, gs, gs' \in GS, ls, ls' \in LS \bullet COP_j(gs, ls, gs', ls')\\ &\qquad \wedge Abs(gs') = as' \wedge INV(gs, ls) \wedge status(gs, ls) = IN(in))\\ &\Rightarrow Abs(gs) = as' \wedge INV(gs', ls') \wedge status(gs', ls') = IN(in)\\ &\quad \vee (\exists\, as \in AS, out \in V \bullet AOP_{abs(j)}(in, as, as', out)\\ &\qquad \wedge Abs(gs) = as \wedge INV(gs', ls') \wedge status(gs', ls') = OUT(out))\end{aligned}$$

where two outcomes are possible. Either COP_j was a skip step or a linearizing step. In both cases, the invariant must hold initially, $INV(gs, ls)$, and after COP_j was performed, $INV(gs', ls')$. Also $status(gs, ls) = IN(in)$ must hold initially in both cases. If the operation COP_j is not an LP (a skip step), then the $status(gs, ls)$ must remain $IN(in)$ and the abstract state $Abs(gs) = Abs(gs') = as'$ must not be modified. If COP_j is the LP of the implemented operation, then there must have been

an abstract state as, from which the abstract operation $AOP_{abs(j)}(in, as, as', out)$ leads to the new state as'. Here, we use $abs(j)$ to map concrete operations to their corresponding abstract operation that they implement. Furthermore, the return value out is finalized in case of an LP and the status must change to $OUT(out)$.

The third case deals with all internal operations COP_j after the LP and before the response operation:

$$\begin{aligned}&(\forall\, as \in AS, gs, gs' \in GS, ls, ls' \in LS \bullet COP_j(gs, ls, gs', ls')\\ &\qquad \wedge Abs(gs') = as \wedge INV(gs, ls) \wedge status(gs, ls) = OUT(out))\\ &\Rightarrow Abs(gs) = as \wedge INV(gs', ls') \wedge status(gs', ls') = OUT(out)\end{aligned}$$

where again the invariant INV has to hold before and after the operation COP_j. The abstraction function $Abs(gs) = Abs(gs')$ must also map both global states gs, gs' to the same abstract state as. Furthermore, the linearization status must not be modified by the operation, because otherwise, a concrete operations would be able to linearize again as we could end up in the previous case.

The last case is the lemma that must be proven for response operations:

$$\begin{aligned}&(\forall\, as \in AS, gs, gs' \in GS, ls, ls' \in LS \bullet RETOP(gs, ls, gs', ls', out)\\ &\qquad \wedge Abs(gs') = as \wedge INV(gs, ls) \wedge status(gs, ls) = OUT(out))\\ &\Rightarrow Abs(gs) = as \wedge INV(gs', ls') \wedge status(gs', ls') = IDLE\end{aligned}$$

where $RETOP(gs, ls, gs', ls', out)$ is a response operation with the output out. As in all previous cases, the invariant has to hold before and after the operation. Similarly, the abstraction function must map both states to the same abstract state. The specific part of this case is that the linearization status changes from $OUT(out)$ to $IDLE$, which allows the process to invoke new operations.

The above proof obligations consider only internal operations COP_j being the LP of an implemented operation. However, in some cases an invoke or response operation can also be the LP. In such a case, the proof obligations for the invoke or response must look similar to the second case, in which they also have to perform an abstract operation $AOP_{abs(j)}$ and change their linearization status accordingly. Also important to note is that by stating whether an operation is $IN(in)$ or $OUT(out)$, we must know definitely whether a concrete operation is the LP or not (the LP is fixed). That is not always possible as there are implementations with potential linearization points. A potential LP often occurs, if the outcome of an operation is determined by another concurrent process. Potential LPs are handled in the proof obligations by an additional linearization status value $INOUT(in, out)$. The latter represents a state in which an operation has potentially linearized, but is still able to

revise its decision. The additional status value also adds several possible cases to the above proof obligations by allowing an operation to transition to $INOUT(in, out)$ and out of it to the other status values. We omit these cases here, because our later presented case studies did not require them and we do not want to further distract from our overview of the local proof obligations. A more detailed explanation of potential linearization points can be found in [DSW11b, TWS12, TTSW14].

Interference Freedom and Disjointness The previously presented proof obligations for non-atomic refinement consider only refinement of individual operations. Interference by other processes is not taken into account. Thus, the proof obligations can only be considered valid, if we ensure that other processes do not interfere with the refinement relation that holds for each process individually. In other words, we have to prove interference freedom:

$$\begin{aligned}(\forall\, gs, gs' \in GS, lsp, lsp', lsq \in LS \bullet\ & COP_j(gs, lsp, gs', lsp') \\ & \wedge INV(gs, lsp) \wedge INV(gs, lsq) \wedge D(lsp, lsq)) \\ \Rightarrow\ & INV(gs', lsq) \wedge D(lsp', lsq) \wedge status(gs, lsq) = status(gs', lsq)\end{aligned}$$

where lsp, lsp' belong to some process p performing the operation COP_j and lsq is the local state of another process q. The process q represents all other processes symbolically with whom process p can interfere. Initially, the invariants of both processes $INV(gs, lsp) \wedge INV(gs, lsq)$ hold. Furthermore, a disjointness predicate $D(lsp, lsq)$ is assumed to hold as well. The latter is sometimes necessary for formalizing invariant properties among different processes, which cannot be formalized in a pure thread modular setting, e.g., disjointness of process identifiers ($p \neq q$). Essentially the above lemma combines all steps of p with all possible states of q. All of this combinations must not invalidate the invariant $INV(gs', lsq)$ of process q, even if the global state was changed by p. The disjointness predicate $D(lsp', lsq)$ also must still hold and the linearization status of q must remain unchanged by the operation COP_j. Identical lemmas also must be proven for invoke and response operations, since they also have the potential to interfere with other processes.

Initialization The previous proof obligations and the interference freedom proof obligations together guarantee that the concrete data structure refines the abstract data structure in a concurrent setting. However, each lemma that we presented here assumes the invariants and/or the abstraction function to hold in the first place. Thus, as a last proof obligation we must prove correct initialization, which must

establish exactly this assumption for initial states:

$$
\begin{aligned}
&\forall\, gs \in GSInit \bullet \\
&(\exists\, as \in ASInit \bullet Abs(gs) = as) \\
&\wedge\, (\forall\, ls \in LS, p \in P \bullet LSInit(ls, p) \Rightarrow INV(gs, ls)) \\
&\wedge\, (\forall\, ls, lsq \in LS, p, q \in P \bullet LSInit(ls, p) \wedge LSInit(lsq, q) \wedge p \neq q \Rightarrow D(ls, lsq))
\end{aligned}
$$

where $GSInit$ is the set of global initial states and $ASInit$ the set of abstract initial states. The abstraction function Abs must map all initial global states to an initial abstract state. Furthermore, each initial local state $ls \in LS$ owned by a process $p \in P$ must satisfy its invariant $INV(gs, ls)$. Consider $LSInit(ls, p)$ to be true, iff ls belongs to process p and is an initial state. Lastly, all initial local states ls, lsq owned by disjoint processes p, q must satisfy the disjointness predicate $D(ls, lsq)$.

This completes our presentation of the local proof obligations for linearizability, which in turn imply the existence of a backward simulation between the concrete and abstract data structure. More details on the construction of the simulation can be found in the work by Derrick et al. [DSW11a, DSW11b, SWD12].

The practical steps towards a linearizability proof within the above framework are as follows:

1. Provide specification of the abstract data type ADT in terms of a state definition AS and a set of abstract operations AOP_i.

2. Provide specification of the concrete data type CDT in terms of shared and local states GS and LS and a set of operations COP_j. Weak2SC can generate these specifications from store buffer graphs.

3. Provide an abstraction relation ABS (or function) that maps concrete states to abstract states.

4. Identify LPs and define the $status$ function by mapping concrete operations to skip steps and abstract steps accordingly.

5. Provide a local invariant INV.

6. Instantiate the proof obligations from the provided linearizability theory and carry out the proofs.

These steps will guide us through the linearizability proofs for our case study. As stated above, Weak2SC can generate the concrete state specifications and the set of operation from store buffer graphs. However, at the time when we carried out the proofs, the transformation was not yet implemented in Weak2SC and we had to do it

manually. Therefore, parts of the provided specifications in this section may deviate from what was presented in Section 4.4.2. The experiment aims at comparing proof effort and complexity of a store buffer graph based encoding against an operational style encoding. When we initially proposed our reduction in [WT15], we carried out a similar proof comparison for the Burns Mutex algorithm [BL80] and the results were promising. The proof based on the reduction, could be automated to a higher degree and the invariant was simpler. It was also back then, when we proved the Arora et al. work stealing queue to be linearizable under TSO. The underlying concrete specification was an encoding of its store buffer graphs. In order to compare the influence of the underlying encoding on the proof effort, we proved the work-stealing queue linearizable once again. This time, we went the extra mile and proved it with an underlying operational style encoding of TSO. Thus, we can report on two case studies, each proved twice to be linearizable. Please note that the effort required for a linearizability proof can be days if not weeks of work, which is also why we report only on two case studies here.

In the following, we will limit our presentation to the *pushBottom* operation of the Arora et al. queue, in order to keep presentation concise. The *pushBottom* operation is the shortest, but allows us to present all facets of the proof method. The complete proofs can be found on the DVD attached to this thesis.

6.2.2 Abstract Data Type

An abstract data type defines the legal behavior of a linearizable data structure. A data structure is linearizable if all its concurrent histories are equivalent to a sequential history. In order to provide all possible sequential histories of a data structure, we can use an abstract specification that has only atomic operations. All its executions are sequential and thus correspond to a sequential history.

An abstract atomic specification consists of two parts, an initialisation predicate $ASInit$ and a set of operations AOP_{op}, each encoding one operation op atomically. Since our chosen case study is a queue, we can define its behavior in terms of operations on a list x. Consequently, we represent the abstract state as a list. Lists and general operations on lists (e.g., obtain first element or remove last element) are predefined in the theorem prover KIV.

$$
\begin{aligned}
ASInit(x) &\mathrel{\widehat{=}} x = \varnothing \\
AOP_{pushBottom}(n, x, x', r) &\mathrel{\widehat{=}} x' = x + n \\
AOP_{popBottom}(n, x, x', r) &\mathrel{\widehat{=}} x = \varnothing \wedge r = null \\
&\quad \vee\, x' = x.butlast \wedge r = \ulcorner x.last \urcorner
\end{aligned}
$$

$$AOP_{popTop}(n, x, x', r) \mathrel{\widehat{=}} x = \varnothing \wedge r = null \\ \vee\ x' = x.rest \wedge r = \ulcorner x.first \urcorner$$

Initially, a queue should be empty and thus $x = \varnothing$ must hold. There are three operations to be defined, *pushBottom*, *popBottom* and *popTop*. For each of the operations, we define a predicate AOP_{op} describing it with an input variable $n \in \mathbb{N}$, the list before x and after x' the operation took effect and an output variable $r \in \mathbb{N} \cup null$. The $AOP_{pushBottom}$ operation simply appends the parameter n to the end of the list x. Thus, the new list is $x' = x + n$, where the plus operator is a concatenator between a list and an element. We do not have to define what the return parameter r is, because the *pushBottom* operation of the work-stealing queue does not produce any output. Similarly, we do not use the input n for the encoding of the other two operations. In addition, we have to consider a possibly empty list $x = \varnothing$, for which both operations return $r = null$. The *popBottom* operation operates at the bottom end of the queue or in other words at the end of the list. Thus, $AOP_{popBottom}$ returns the last element of the list $x.last$, where $last$ is one of the predefined list operations. Please note that we use $\ulcorner n \urcorner$ to lift a natural number n to the sort we defined for $\mathbb{N} \cup null$. The new list x' after removal of the last element from x becomes $x.butlast$. The *popTop* method operates on the top end of the queue or at the beginning of the list. Thus, the operation AOP_{popTop} removes the first element $x.first$ and returns it. Consequently, the list x' after removal of the first element is $x.rest$, where $rest$ applied to x returns the list obtained by removing the first element from x.

As a next step, we have to define the concrete data type, such that we can define its relation to the abstract data type.

6.2.3 Concrete Data Type

The concrete data type is an encoding of a data structure implementation. Before we define its operations, we first need a definition of the set of states. Later, the concrete operations will be defined over this set of states. As already mentioned previously, the proof obligations come in two different versions, local [DSW11b] and global [SWD12]. These differ essentially in the way the state is formalized. The global proof obligations require a state definition that combines all variables, the shared state such as the memory and the local state of all processes, into one single state tuple CS. On the other hand, the local proof obligations assume a separation between the state shared by all processes GS and the local state of a process LS. Both must be defined as separate tuples. Please note that the global state can be constructed simply by combining the shared state GS with a local state function $lsf : P \to LS$ that maps all processes $p \in P$ to a local state $ls \in LS$. The latter is

exactly what happens in the soundness proofs of the linearizability theory for the local proof obligations [DSW11b, SWD12].

By choosing the local proof obligations, we can avoid quantification over processes when defining operations or invariant properties as we always talk about the combination of one local state with the shared state. This limits our ability to define properties, e.g., putting states of different processes in relation, but it also simplifies our specification task as we consider only one process at a time.

Since we want to analyze the impact of our reduction approach to the proof effort, we provide two different concrete specifications. The first is a specification based on an operational style encoding of the semantics. This is essentially an explicit encoding of the store buffer and all possible behavior related to it. Local states in this encoding have a store buffer. The semantics are given as a set of axioms defining the effects of program statements, such as read, write or the effect of a flush. The concrete specification encodes the control flow of each operation of our data structure. Flushes are triggered implicitly and non-deterministically at any state that has a non-empty store buffer.

State Definition

We use one definition of the shared state in both of our concrete specifications. The shared state is the memory function $Mem : \mathbb{N} \to \{\mathbb{N} \cup null\}$. We use the notation $mem[n] = a$ in order to say that the memory at location n has the value a. We use $mem' = mem[n, a]$ to denote a memory mem' that is identical to mem except for location n for which value a replaces the old value $mem[n]$.

We define the local state tuple as follows:

$$
\begin{aligned}
LS \subseteq (&PC \times P && \text{(.pc .p)}\\
&\times \mathbb{N} \times \mathbb{N} \times \mathbb{N} && \text{(.bot .top .tag)}\\
&\times \mathbb{N} \times \mathbb{N} \times \mathbb{N} && \text{(.bota .topa .taga)}\\
&\times \mathbb{N} \times \mathbb{N} \times \mathbb{N} && \text{(.deqa .ntop .ntag)}\\
&\times \mathbb{N} \times \{\mathbb{N} \cup null\} && \text{(.elem .ret)}\\
&\times (\mathbb{N} \times \{\mathbb{N} \cup null\})^*) && \text{(.buf)}
\end{aligned}
$$

where we added access functions to each tuple entry at the right of the definition (in brackets). The first two entries encode the program location PC and the process identifier P. These are followed by the local variables as used by the work-stealing queue implementation. The variables $bota$, $topa$, $taga$ and $deqa$ encode pointers, and thus represent memory locations that contain the actual values. The local variables bot, top and tag hereby represent actual values, which the algorithm obtains from

reading a value from the above locations. The variable $deqa$ points to the first location of the array that holds the elements of the queue. The array is assumed to be an infinite sequence of consecutively allocated memory locations. The implementation by Arora et al. also does not handle the case of a full array and thus, implicitly assumes an infinite length of the array. The tuple entries $elem$ and ret are used to encode input and output parameters. The last entry buf represents the store buffer and is only required by the operational style encoding of the concrete specification.

Store Buffer Semantics

The TSO semantics based on explicit store buffer representation were defined as follows:

$$\begin{aligned}
read(loc, buf, mem) =& (loc \in buf \\
& \supset latest(loc, buf) \\
& ;\ mem[loc]) \\
write(loc, r, ls.buf) =& ls.buf + (loc \times r) \\
\mathit{flush}(buf, mem, buf', mem') \mathrel{\widehat{=}}& buf \neq \varnothing \wedge buf' = buf.rest \\
& \wedge mem' = mem[buf.first.l, buf.first.v] \\
cas(loc, r1, r2, buf, mem, mem') \mathrel{\widehat{=}}& buf = \varnothing \wedge (mem[loc] = r1 \\
& \supset mem' = mem[loc, r2] \\
& ;\ mem' = mem)
\end{aligned}$$

where we defined the semantics of reads and writes as functions for convenience and used predicates for flushes and CAS operations. The function $read$ determines the value that it returns based on its parameter buf and mem. If $loc \in buf$, then the value is taken from the store buffer. In this case it is an early read. The obtained value has to be the latest entry $latest(loc, buf)$ in the store buffer. Otherwise, the $read$ function maps to the memory value $mem[loc]$ of the requested location loc. We define the case distinction using the short notation in KIV, where a condition is followed by "$\supset$" and the statement in case of a true condition, followed by "; " and the else case statement. Note that statements do not have to be boolean expressions as in case of the above $read$ function, which maps to $\mathbb{N} \cup null$. The $write$ function maps to a new buffer value. It simply creates a pair of the given location parameter $loc \in \mathbb{N}$ and the value $r \in \mathbb{N} \cup null$ and appends it at the end of the buffer buf. The flush predicate defines two cases. The first case represents a skip step that does not modify the state, $buf' = buf$ and $mem' = mem$. The second case is only enabled, if the buffer is non-empty, $buf \neq \varnothing$. In this case, the new buffer value buf' becomes

the remaining list of the previous buffer after removing the first element, which is $buf.rest$. The memory is also updated by the flushed entry $buf.first$. Thus, the new memory mem' becomes $mem[buf.first.l, buf.first.v]$, where $.l$ (resp. $.v$) is the access function to the location (resp. value) of the entry. The cas predicate encodes the semantics of a compare-and-swap instruction. Similar to a fence, it blocks if the buffer is not empty. A cas succeeds, if the memory mem at location loc equals the expected value $r1$. In this case, the memory is updated with the new value $r2$ at the location loc. Otherwise, the memory remains unmodified, $mem' = mem$.

Concrete Operations

In the following, we provide an excerpt of the concrete operations that we used to encode the behavior of the *pushBottom* method of the work-stealing queue. The complete specification can be found in our proof projects on the DVD attached to this dissertation. In fact, we provide two versions of the operation encoding. The first one used the TSO semantics as defined above. The second one is defined according to our reduction in Chapter 3 and could also be generated by Weak2SC. However, as we carried out the proofs before the transformation from store buffer graphs towards KIV specifications was automated in Weak2SC, the encodings were created manually and therefore deviate slightly from our implementation in Weak2SC, e.g., in terms of naming conventions. We present both encodings by defining the operations of the *pushBottom* method.

We distinguish between three types of operations: invoke ($INVOP$), return ($RETOP$) and internal operation (COP). An invoke updates the program location from being idle (N) to the first program location of a method and assigns an input value ($elem$ in our case study) to a local variable. Similarly, a return operation changes the program location to being idle again and defines the returned value (named $return$). All the remaining behavior corresponding to the *pushBottom* operation is defined by internal operations, COP_i.

Explicit Store Buffer Encoding The operation encoding of *pushBottom* based on explicit store buffer representation and the above TSO semantics is defined as follows:

$$
\begin{aligned}
INVOP_{ub0} \mathrel{\widehat{=}} & ls.pc = N \wedge ls.pid = owner \wedge mem' = mem; \\
& \wedge\, ls' = (ls.pc := UB0\ .elem := elem) \\
COP_{ub1} \mathrel{\widehat{=}} & ls.pc = UB0 \wedge mem' = mem; \\
& \wedge\, ls' = (ls.pc := UB1\ .bota := read(bota, ls.buf, mem).v) \\
COP_{ub2} \mathrel{\widehat{=}} & ls.pc = UB1 \wedge mem' = mem;
\end{aligned}
$$

$$
\begin{aligned}
&\wedge\, ls' = (ls.pc := UB2\ .bot := read(ls.bota, ls.buf, mem).v)\\
COP_{ub3} \mathrel{\widehat{=}} &ls.pc = UB2 \wedge mem' = mem;\\
&\wedge\, ls' = (ls.pc := UB3\ .deqa := read(deqa, ls.buf, mem).v)\\
COP_{ub4} \mathrel{\widehat{=}} &ls.pc = UB3 \wedge mem' = mem\\
&\wedge\, ls' = (ls.pc := UB4\ .buf := write(ls.deqa + ls.bot, \ulcorner ls.elem \urcorner, ls.buf))\\
COP_{ub5} \mathrel{\widehat{=}} &ls.pc = UB4 \wedge mem' = mem\\
&\wedge\, ls' = (ls.pc := UB5\ .buf := write(ls.bota, \ulcorner ls.bot + 1 \urcorner, ls.buf))\\
RETOP_{ub6} \mathrel{\widehat{=}} &ls.pc = UB5 \wedge mem' = mem\\
&\wedge\, return = \ulcorner 1 \urcorner \wedge ls' = (ls.pc := N)\\
COP_{flush} \mathrel{\widehat{=}} &\exists\, buf' \bullet ls' = ls.buf := buf'\\
&\wedge\, flush(ls.buf, mem, buf', mem')
\end{aligned}
$$

Unprimed and primed variables represent the variables before and after the effect of an operation. Each operation is a predicate over ls, ls' and mem, mem'. Most changes do only affect a single variable of the local state while leaving the rest unmodified. We use the KIV short notation in these cases, e.g. $ls' = (ls.pc := UB0\ .elem := elem)$ in order to define that ls' equals to ls, except for the entries pc and $elem$, which are updated with the respective values.

Except for the flush operation COP_{flush}, all of the above operations only modify the local state ls. All of them are enabled only at certain program locations and update them to the next program location, thereby encoding the control flow of *pushBottom*. However, COP_{flush} is not restricted to a certain program location and is enabled whenever the store buffer $ls.buf$ is not empty. The first three COP operations read from memory. Transition COP_{ub4} and COP_{ub5} write to it, but since a write does not write directly to the memory, only the store buffer is modified. Variable $ls.deqa$ points to the start of the array in the memory. Since, the array allocates a number of consecutive memory locations, $ls.deqa + ls.bot$ points to the $ls.bot$ cell in the array. In fact, we reduced two local operation to a single one here in order to avoid an additional temporary variable that holds the value $ls.deqa + ls.bot$. This is sound, since local operations cannot be observed by other processes. In other words, they commute with the operations of other processes and thus qualify for partial order reduction [Val89].

Store Buffer Graph Encoding The operation encoding of *pushBottom* based on our proposed reduction approach is as follows:

$$
\begin{aligned}
INVOP^r_{ub0} \mathrel{\widehat{=}} & ls.pc = N \wedge ls.pid = owner \wedge mem' = mem \\
& \wedge ls' = (ls.pc := UB0\ .elem := elem) \\
COP^r_{ub1} \mathrel{\widehat{=}} & ls.pc = UB0 \wedge ls' = (ls.pc := UB1\ .bota := mem[bota].v) \\
& \wedge mem' = mem \\
COP^r_{ub2} \mathrel{\widehat{=}} & ls.pc = UB1 \wedge ls' = (ls.pc := UB2\ .bot := mem[ls.bota].v) \\
& \wedge mem' = mem \\
COP^r_{ub3} \mathrel{\widehat{=}} & ls.pc = UB2 \wedge ls' = (ls.pc := UB3\ .deqa := mem[deqa].v) \\
& \wedge mem' = mem \\
COP^r_{ub4} \mathrel{\widehat{=}} & ls.pc = UB3 \wedge ls' = (ls.pc := UB4) \\
& \wedge mem' = mem[ls.deqa + ls.bot, \ulcorner ls.elem \urcorner] \\
COP^r_{ub5} \mathrel{\widehat{=}} & ls.pc = UB4 \wedge ls' = (ls.pc := UB5) \\
& \wedge mem' = mem[ls.bota, \ulcorner ls.bot + 1 \urcorner] \\
RETOP^r_{ub6} \mathrel{\widehat{=}} & ls.pc = UB5 \wedge ls' = (ls.pc := N) \\
& \wedge return = \ulcorner 1 \urcorner \wedge mem' = mem
\end{aligned}
$$

The operations look very similar to the ones we used for the explicit store buffer encoding. Please note that we again removed the same local operations that would otherwise introduce additional temporary local variables. Operations $INVOP^r_{ub0}$ and $RETOP^r_{ub6}$ are identical to $INVOP_{ub0}$ and $RETOP_{ub6}$. However, the remaining operations encode the edges from our store buffer graph (see Figure 4.4 on p. 76) and their effect according to our transformation. So, instead of using additional predicates and functions that define the semantics, mostly based on the local store buffer, we use simple SC semantics. Thus, the first three (read) operations simply take their value from the memory. It may not be obvious, but this significantly simplifies later proof reasoning. We know from our store buffer graph that the store buffer is empty during the first three statements. Thus, these reads will not take their value from the store buffer. However, in our explicit store buffer encoding the *read* predicate takes a non-empty store buffer into account. Thus, a later proof will unfold a case for a non-empty buffer. This case then must be ruled out by establishing an invariant property, which states that the store buffer is empty throughout the first three statements of *pushBottom*. It is particularly the latter steps that make proofs complex or tedious and which are obsolete for the above encoding, because they are computed in advance.

Operations COP^r_{ub4} and COP^r_{ub5} may seem like they encode the writes COP_{ub4}

and COP_{ub5} from the previous encoding, but this is not the case. COP^r_{ub4} and COP^r_{ub5} encode the flush edges from the store buffer graph, which correspond to the respective writes. The actual write edges became skip operations and a skip is a commuting operation, which is why they were omitted here. Otherwise, we would have to introduce an additional pc value (say UBi) and an operation that does nothing but to modify the pc value from $UB3$ to UBi. In addition, we would have to modify operation COP^r_{ub4} to start at location UBi. We would like to remind the reader that the pc values in our store buffer graph encoding are essentially enumerating the nodes in the store buffer graph, and each node corresponds to a program location and a symbolic store buffer state. In contrast, the program locations in the previous encoding correspond to actual program locations in the code. The naming of the program locations in both encodings is therefore a coincidence and not given by construction.

For convenience, we also introduced a few constants that we could use to abbreviate our notation. In our LLVM code for the Arora et al. queue, we have pointers to memory locations that are defined statically. These are $deqa$, $bota$, $topa$, $taga$, all suffixed for with an "a" for address. However, these pointers must be initialized and are not changed by the program implementation. Thus, we defined constants deq, bot, top, tag to represent these memory locations.

$$
\begin{aligned}
mem[deqa].v &= deq \\
mem[bota].v &= bota \\
mem[topa].v &= top \\
mem[taga].v &= tag
\end{aligned}
$$

You may also notice that top and tag are represented by separate variables, although the implementation by Arora et al. represents both values as a single age variable. The operations reading from or writing to variable age are modified to perform the same operation on top and tag. The adjustment allows us to avoid verbosity in our encoding. Otherwise, we would have to deal with a memory function that maps to pairs of natural numbers, which woulds require even more lifting of variables than we already have, e.g., suffix $.v$ in order to obtain the non-$null$ value in a memory location.

6.2.4 Abstraction Function

After providing an abstract and concrete specification of the implemented data structure, we have to relate the two. In principle, defining a relation is sufficient, but a function that maps each concrete state to an abstract state is more convenient for

a proof. In case of a relation, it is possible that a concrete state is related to several abstract states or that there is no abstract state related to a concrete state. The later linearizability proof will almost certainly fail if we run into such a case. In case of a function, each concrete state is mapped to a single abstract state, which reliefs us from this type of complications.

In order to define an abstraction function, we need to first ask whether local states of processes are relevant for the abstract state or whether we can fully determine the abstract state from the shared state. This is an essential question, because if local states are relevant and the abstract state cannot be fully observed from the shared state, then the abstraction function (taking only shared state into account) may miss important aspects and we may not be able to succeed with a linearizability proof based on the local proof obligations [DSW11a].

The local proof obligations for linearizability take the execution of a single process into account, which executes against a symbolic process that represents all other processes. The latter is only valid, if they all appear to be identical to the executing processes, i.e., they can be generalized to one symbolic adversary process. If this is not the case, as it is for the Arora et al. queue (owner and stealer roles of processes), we have two options: we can use global proof obligations [SWD12] or we can add auxiliary variables. The global proof obligations do not separate between shared and local state encoding and therefore allow to refer to local states of all processes directly. However, the specification of invariants etc. also tend to be more verbose. The other option that we also chose for our running example is to add auxiliary variables. Auxiliary variables allow us to share information from the local state of other processes by adding this information to the shared state. Thus, we can then refer to the auxiliary variables whenever we would otherwise refer to the local state of a process. Auxiliary variables enable us to enjoy the benefits of local reasoning while reasoning about global arguments whenever it cannot be avoided.

The abstraction function for the Arora et al. queue maps from a memory mem to a list $list$ that represents the queue. Therefore, we first define how to retrieve a list from a memory in general. The memory locations reserved for the queue q in the memory mem start at some constant location deq. By considering all locations that are larger than deq as array locations, we assume the array has infinite length. We have to make this assumption. Otherwise, the implementation would be trivially not linearizable, because it does not handle finite size arrays. Within that range of array locations, we now have to distinguish which elements are considered as being in the abstract queue. These are given by the variables top and bot, where top marks the start of the queue and bot the end of it. Each location within that range must be an element of the abstract queue. The function $q : memory \times \mathbb{N} \times \mathbb{N} \times \mathbb{N} \to list$ is

defined recursively as follows:

$$n \leq m \models q(mem, k, m, n) = \varnothing$$
$$m < n + 1 \models q(mem, k, m, n + 1) = q(mem, k, m, n) + mem[k + n].v$$

where $m, n, k \in \mathbb{N}$. Variables m and n are the start and end of the queue. Variable k marks the beginning of the array locations. The first row provides an empty queue if $n \leq m$. The second row defines the recursion, which appends the element $mem[k + n].v$ to the end of the list given by $q(mem, k, m, n)$. As a reminder, the memory function mem maps to $null \cup \mathbb{N}$ and by using the suffix $.v$, we explicitly request the value to be not $null$.

Naively, we could define $q(mem, deq, mem[top].v, mem[bot].v)$ to be the abstraction function, because the elements between top and bot are meant to be the abstract queue. However, this alone is not enough. In particular, the mapping to abstract states depends on the state of the owner processes. When a queue owner performs a *popBottom* operation, variable bot is decremented and furthermore, if the owner notices that it was the last element in the queue, then bot is reset to 0 again (see Figure 4.2 on page 71). A change of the bot value would be immediately visible in the abstract state based on the above function. However, bot is modified before the *pushBottom* linearizes, which means that, during the computation by the owner process, the change must not be reflected in the abstract state until the linearization point of *pushBottom* is reached.

In other words, the abstraction function must deal with two cases: one case in which the owner has not modified bot and one case in which the owner decremented or has even reset bot. In order to distinguish these two cases, we decided to add an auxiliary variable[3] $range$ that encodes the progression of the owner process. Initially, $range = 0$ holds. When the owner decrements bot during execution of *pushBottom*, the $range$ is set to 1. When the owner resets variable bot to 0, then $range$ becomes 2. When *pushBottom* linearizes and the remaining queue is not empty or when the reset of the implementation finishes by setting $top = 0$ and incrementing tag in the last statement of *pushBottom*, $range = 0$ is set to hold again. We modified the operations encoding *pushBottom* to set these values. Please note that it is safe to use auxiliary variables in order to make behavior observable in the shared state. However, an auxiliary variable must have no influence on an operation's enabling condition, i.e., an auxiliary variable must not be contained in the set of unprimed variables of an operation encoding.

[3] Auxiliary variables have their own dedicated memory locations in our encoding. Thereby, we can avoid cascading tuple definitions that would result in verbose notation of formulas. Thus, whenever we use $range$ (resp. $obot$) in our formulas, the technically correct representation is $mem[range].v$ (resp. $mem[obot].v$), which we avoid for the purpose of presentation.

Now that we can determine whether the owner process has modified bot or not, we have to define what the abstract state in each $range$ is. For $range = 0$, it is clearly our naive abstraction function. For $range = 1$, we could use $bot + 1$ as the end of the queue. However, for the last $range = 2$, where bot is set to 0, the end of the queue is lost (only known to owner process) even though *pushBottom* does not have necessarily linearized, yet. In order to overcome this, we add a second auxiliary variable $obot$ that captures the bot value before it is modified by *pushBottom*. Essentially, $obot$ acts as a snapshot of the bot value throughout execution of *pushBottom* and thus, while the end of the queue would not be observable from the shared state mem without the auxiliary variable. Given that second auxiliary variable, the abstraction function is defined as follows:

$$\begin{aligned} Abs(mem) =& (range = 0) \\ & \supset\ q(mem, deq, mem[top].v, mem[bot].v) \\ & ;\ q(mem, deq, mem[top].v, obot) \end{aligned}$$

where we again use KIV short notation for if-then-else. While $range = 0$, we let top and bot determine the start and end of the queue in the array. Otherwise (while bot is modified), we let top and $obot$ determine the start and end of the queue. Please note that stealer processes only modify the top variable and thus, can still modify the abstract state obtained from the abstraction function. The latter is crucial for the proof, since the abstraction function must hold for each and every step of the concrete specification. Any change in the concrete state must be immediately reflected by the abstract state while the abstract specification defines whether it is consistent with a sequential execution.

Linearization Status

In addition to the abstraction function, the local proof obligations for linearizability [DSW11b] also require a fixation of the linearization point (LP). The abstraction function does not define how often the abstract state is allowed to change during execution of a method. In principle, the abstract state to which the abstraction function maps could change with every step of the concrete specification. However, this is not what we want for the linearizability proofs. For linearizability, there must be exactly one LP per method execution. The latter ensures that methods appear to be atomic. The abstraction function alone does not guarantee that.

In the proof obligations by Derrick et al. [DSW11b], this relationship is captured by a $status$ function that defines when and under which conditions a method linearizes. Implicitly, this also defines which of the concrete steps must be skip steps and which have to be mapped to an actual abstract step AOP. The $status$ function

is a big case distinction over program locations and additional conditions. In the following, we provide an excerpt of the $status$ function that concerns the *pushBottom* operation. For the encoding with explicit store buffers, this part looks as follows:

$$status(mem, ls) = IN(in) \qquad \textbf{if}(ls.pc \in \{UB0, UB1, UB2, UB3, UB4\} \vee ls.pc = UB5 \wedge ls.buf \neq \varnothing)$$
$$status(mem, ls) = OUT(out) \qquad \textbf{else}$$

where $IN(in)$ with input parameter in means that the LP has not been passed yet and $OUT(out)$ means that the LP has been passed and out is the return value. For the *pushBottom* operation, the LP is not influenced by parameter in and as it does not return a value, it has also no influence on out. The operation has one single LP and that is when incrementation of bot is written to the memory. Therefore, we can say that *pushBottom* has not linearized until it reaches program location $UB5$. For location $UB5$, we further have to distinguish whether the written value has also been flushed to the memory. As long as the store buffer is not empty ($ls.buf \neq \varnothing$), the operation cannot have linearized and it must have linearized, otherwise.

For the store buffer graph-based encoding, the $status$ function is simpler:

$$status(mem, ls) = IN(in) \qquad \textbf{if}(ls.pc \in \{UB0, UB1, UB2, UB3, UB4\})$$
$$status(mem, ls) = OUT(out) \qquad \textbf{else}$$

In case of *pushBottom*, we can simply distinguish the status of linearization by program location only. The above case of a non-empty store buffer does not occur here. Since write edges in the store buffer graph become skip operations and because skips commute with other operations, we were able to remove it for convenience. If the skips were still present, we would have additional program locations in the above set corresponding to the nodes with non-empty store buffers in the store buffer graph.

We provide the remaining linearization points of the Arora et al. queue informally and refer to the LLVM code in Figure B.1 and Figure B.2 in the appendix. The LPs of *popBottom* and *popTop* are as follows:

- *popBottom* linearizes with its first read (line 44) of variable bot, if $mem[top].v = mem[bot].v$ holds at this point. In this case the queue is empty and the method will return $null$.
- *popBottom* also linearizes by observing the age value (line 56) that combines top and tag variables in one variable. However, it only linearizes at this point, if $mem[top].v < mem[bot].v$ holds and thus, the queue remains non-empty.

- The last linearization point of *popBottom* is the CAS instruction (line 81). At this point, the owner process competes with a stealer process for the last element in the queue. If the CAS succeeds, then *popBottom* obtains the element, otherwise a stealer process has already obtained it.

- *popTop* linearizes with its read of the bot variable (line 22), if $mem[top].v \geq mem[bot].v$ holds. In this case, the queue is empty.

- *popTop* has to succeed with its CAS instruction (line 32) in order to obtain an element from the queue. In contrast to an owner, the stealer process does not necessarily compete for the last element at this stage, but possibly with other stealer processes. Therefore, it can also fail because of another stealer process although the queue is not empty.

Given the above definitions of the abstraction function and the status function, we have defined how and when the abstract data structure must reflect the changes of the concrete data structure. What remains to be defined before a proof can be attempted is the invariant, which we explain in the following section.

6.2.5 Invariant

Besides finding an abstraction function, the most difficult and time consuming part of a correctness proof is to find and establish an invariant for a program. The difficulty comes from different aspects of a proof. First of all, an invariant must be proven to hold throughout the entire program execution, i.e., for each possible step of the transition system. Second, it has to be strong enough in order to deduce that the abstraction function maps to a state that is consistent with the abstract specification. The difficulty arises from the interplay of the two and from the fact that, at least at the beginning, one is usually not aware of all special cases that can appear in an execution of the program to be proven correct.

Structurally, an invariant is a big conjunction of cases over possible program locations and the properties that hold at these locations. Some properties do not depend on program locations. Others hold over a range of program locations. Our choice of the local proof obligations also matters in the sense that we need to quantify over processes in the global proof obligations. For the local proof obligations this is not necessary, because local invariants refer to the shared state and one process only. However, the invariant still has to hold for all processes, but the quantification within the invariant becomes obsolete.

Not being able to specify relationships between different processes in the local proof obligations can also be a drawback, which can be tackled by a disjointness predicate and auxiliary variables. The latter is also the case for the Arora et al.

work-stealing queue, where we had to encode local variables of the owner process into auxiliary variables as part of the shared state. The reason for it is that just from observing the shared state and the state of a stealer process, we do not know enough in order to deduce the correct abstract state. Because the owner process can have decremented variable bot during his *popBottom* operation, we may observe a state in which bot is decremented and *popBottom* may have not linearized yet. From a stealer perspective, this situation looks identical to all other states, where bot is not decremented. However, the abstract state is different for these two cases, which is also why our abstraction function distinguishes them.

Again, we provide only an excerpt of our invariant that we used to prove the case study linearizable. Please note that the complete invariant specification in KIV is over 200 lines of specification long. A first excerpt INV_{gen} shows properties that do not depend on program locations and which is identical in both of our encodings. The program location dependent properties are separated into $INVub_{sb}$ for the explicit store buffer encoding and $INVub_r$ for the store buffer graph encoding. The latter two are invariant excerpts for the *pushBottom* operation only.

$$
\begin{aligned}
&INV_{gen}(mem, ls) \mathrel{\widehat{=}} \\
&\quad (range = 0 \Rightarrow mem[top].v \leq mem[bot].v) \\
&\quad \wedge\, (range = 1 \Rightarrow mem[top].v \leq obot \wedge mem[bot].v + 1 = obot) \\
&\quad \wedge\, (range = 2 \Rightarrow mem[bot].v = 0 \\
&\qquad \wedge\, (mem[top].v = obot \vee mem[top].v + 1 = obot)) \\
&\quad \wedge\, (range \in \{0, 1, 2\}) \\
&\quad \wedge\, (\forall\, n \bullet (n < deq + mem[bot].v \wedge n \geq deq) \Rightarrow mem[n] \neq null) \\
&\quad \wedge\, (range \neq 0 \Rightarrow \forall\, n \bullet (n < deq + obot \wedge n \geq deq) \Rightarrow mem[n] \neq null) \\
&\quad \wedge\, (owner(ls.pc) \Rightarrow ls.pid = owner) \\
&\quad \wedge\, (nowner(ls.pc) \Rightarrow ls.pid \neq owner) \\
&\quad \wedge\, (range \neq 0 \Rightarrow obot \neq 0)
\end{aligned}
$$

The first three conjuncts are the most important properties of the algorithm. In $range = 0$, we know that the owner process does not try to remove an element from the queue. In this case, we know that top is always smaller than bot (queue non-empty) or equal to bot (queue is empty). During $range = 1$, the owner tries to remove an element from the queue and has already decremented the bot variable. Before bot is decremented, its value is assigned to $obot$. Thus, $mem[bot].v + 1 = obot$ holds. In $range = 2$, the value of bot is reset to 0 by the owner process. In this case, there is at

most one element in the queue. The owner and a stealer process could still compete for the last element within this range. The winner of the last element becomes the process whose CAS operation succeeds. Of course, our invariant must also establish that an owner process assigns the proper values to the auxiliary variables $range$ and $obot$ during execution of *popBottom*. This is achieved in a similar way to the program location dependent invariants $INVub_{sb}$ and $INVub_r$ for the *pushBottom* operation. Here, we only state that $range$ must be in the set $\{0, 1, 2\}$. A minor invariant property is also that when $range \neq 0$, then $obot$ also cannot be 0 as it is the value of bot before it is decremented.

Furthermore, we need to establish that values taken from memory, e.g., the return value of a *popBottom* operation, are not $null$. In order to do so, we quantify over all memory locations between start of the array deq and the current end of the queue represented by $deq + mem[bot].v$. During the phase of a decremented bot value (when $range \neq 0$ holds), we use our auxiliary variable $obot$ instead of bot in order to establish that the last element in the queue remains not $null$ after bot has been decremented. The latter argument is particularly necessary at a linearization point of *popBottom*, which occurs when bot is decremented, but the actual value of the last element has not been read from the memory, yet. We also need to establish the roles of processes, i.e., an owner can only be at program locations of *pushBottom* and *popBottom* while a stealer process can only be at locations of a *popTop* operation. However, both processes can be at the program location N, representing the idle state. This is also why the predicates $owner$ and $nowner$ are not complementary.

The local invariant of a process executing *pushBottom* for the encoding with explicit store buffers looks as follows:

$$
\begin{aligned}
&INVub_{sb}(mem, ls) \mathrel{\widehat{=}} \\
&\quad (ls.pc \in \{UB0, UB1, UB2, UB3\} \Rightarrow ls.buf = \varnothing) \\
&\quad \wedge (ls.pc = UB4 \Rightarrow ls.buf = \varnothing \\
&\qquad \vee ls.buf = \varnothing + mk(ls.deqa + ls.bot, \ulcorner ls.elem \urcorner) \\
&\quad \wedge (ls.pc = UB5 \Rightarrow ls.buf = \varnothing \\
&\qquad \vee ls.buf = \varnothing + mk(ls.bota, \ulcorner ls.bot + 1 \urcorner)) \\
&\qquad \vee ls.buf = (\varnothing + mk(ls.deqa + ls.bot, \ulcorner ls.elem \urcorner)) + mk(ls.bota, \ulcorner ls.bot + 1 \urcorner)) \\
&\quad \wedge (ls.pc \in \{UB1, UB2, UB3, UB4, UB5\} \Rightarrow ls.bota = mem[bota].v) \\
&\quad \wedge (ls.pc \in \{UB2, UB3, UB4, UB5\} \Rightarrow ls.bot = mem[ls.bota].v) \\
&\quad \wedge (ls.pc \in \{UB3, UB4, UB5\} \Rightarrow ls.deqa = mem[deqa].v) \\
&\quad \wedge (ls.pc \in \{UB4, UB5\} \wedge ls.buf = \varnothing \Rightarrow mem[deqa + ls.bot] = \ulcorner ls.elem \urcorner) \\
&\quad \wedge (ls.pc = UB5 \wedge ls.buf = \varnothing \Rightarrow mem[bot] = \ulcorner ls.bot + 1 \urcorner)
\end{aligned}
$$

where the first three conjuncts describe the possible states of the store buffer. If we do not define the possible states of the store buffer here, we would have to deal with arbitrarily filled store buffers in our proofs. The proof would most likely fail, because we have to at least assume that the store buffer does not contain writes to the relevant variables like bot or top. Otherwise, an arbitrary write or particularly its flush to one of those variables would violate our invariant properties immediately. The store buffer is empty until it is filled in location $UB4$ and $UB5$, where it can also be empty as a result of flushes that may have occurred. The notation for an explicit store buffer value in our KIV encoding always concatenates a memory location-value pair (e.g., $mk(ls.bota, \ulcorner ls.bot + 1 \urcorner)$) to an empty list constant $\varnothing$.

Besides the possible states of the store buffer, we also have to relate some of the local variables to the shared ones in the memory. These may or may not depend on the state of the store buffer. For the local variables $bota$, bot and $deqa$ representing pointer variables in our original C code, we need to establish that they do not change after they are read until the end of *pushBottom* at location $UB5$. The algorithm does not modify those locations and therefore, they are trivial. The more important properties are the last two conjuncts. The first of them ensures that the input parameter $ls.elem$ is indeed written to the memory at location $deqa+ls.bot$, however only if the store buffer $ls.buf$ is empty. Similarly, the last conjunct establishes the incrementation of $mem[bot]$, and again only if the the store buffer is emptied already.

For the store buffer graph encoding of the *pushBottom* method, we used the following invariant:

$$
\begin{aligned}
&INVub_r(mem, ls) \mathrel{\widehat{=}} \\
&\quad (ls.pc \in \{UB1, UB2, UB3, UB4, UB5\} \Rightarrow ls.bota = mem[bota].v) \\
&\quad \wedge\, (ls.pc \in \{UB2, UB3, UB4, UB5\} \Rightarrow ls.bot = mem[ls.bota].v) \\
&\quad \wedge\, (ls.pc \in \{UB3, UB4, UB5\} \Rightarrow ls.deqa = mem[deqa].v) \\
&\quad \wedge\, (ls.pc \in \{UB4, UB5\} \Rightarrow mem[deqa + ls.bot] = \ulcorner ls.elem \urcorner) \\
&\quad \wedge\, (ls.pc = UB5 \Rightarrow mem[bot] = \ulcorner ls.bot + 1 \urcorner)
\end{aligned}
$$

which contains all of the properties from explicit store buffer encoding, except for the store buffer states. Since some skip transitions were removed (as mentioned earlier), we also have identical program location ranges. Otherwise, we would simply extend the program location ranges over which the above properties have to hold. Please note that the store buffer state from the previous invariant is captured within the set of operations encoding the *pushBottom* behavior and therefore does not have to be

	explicit sb		sb-graph	
	#steps	#intera.	#steps	#intera.
Arora et al. Queue [ABP98]	12,197	1,624	6,923	1,100
Burns Mutex [BL80]	3,784	201	1,536	63

Table 6.4: Number of proof steps in the theorem prover KIV for the linearizability proofs of the above case studies.

considered here. The complete invariant can be found in the KIV project provided with this thesis.

6.2.6 Proof Procedure and Comparison

Since we proved the local proof obligations [DSW11a], our proofs are separated into several proofs. These proofs combined imply linearizability of the implemented data structure w.r.t. an abstract data structure. Most of the proofs were trivial, but we had three proofs where most of our effort was spent. Two of these three proofs show refinement of internal concrete operations and the abstract specification (see Section 6.2.1). The latter is split into two separate proofs, one for the preservation of the invariant and one for the abstraction function. The third proof concerns non-interference. It is no surprise that most of our time was spent in particularly these three proofs as they deal with all the complexity from the internal behavior of the concrete data structure.

All of the proof obligations were proved via a case distinction over the possible concrete operations. This enabled us to tackle each operation separately with all its preconditions and postconditions, be it the invariant or the abstract state transitioning from one state to another. Table 6.4 shows the overall number of steps and manual interactions during the proofs for both of our case studies, the Arora et al. work-stealing queue [ABP98] and the Burns Mutex algorithm [BL80]. For the latter, the proof results were also published in [WT15]. The left two columns contain the steps and interactions that we obtained by using explicit store buffer encoding of TSO semantics. The right two columns show the number of steps and interactions that we were able to achieve with our store buffer graph encoding. We should mention that we did not focus on automation of these proofs, but rather let the theorem prover KIV apply its heuristics. Only in some cases like the lifting of variables, we added some rules to automate parts of the proofs. These rules were applied in all four proofs.

The number of steps and interactions give a good hint on which of the approaches requires more effort. However, these number have to be taken with a grain of salt

as some steps can be more difficult than others up to the point that we can fully automate some steps. Nevertheless, we can say that several properties in the explicit store buffer encoding were more complex than those in the encoding of the store buffer graph. That is simply because a store buffer is not present in the store buffer graph encoding an all properties refer to valuations of either some memory location or some local variable, i.e., rather simple properties. These properties hold over some range of program locations.

With explicit store buffers, there are additional properties concerning the store buffer that have to be stated and also derived in the proofs, just as can be seen by the invariants $INVub_{sb}$ and $INVub_{r}$ in the previous section. Since store buffers are lists of elements, the reasoning requires dealing with an additional set of axioms and lemmas for lists. Some properties depend on the state of the store buffer. For these properties, reasoning about store buffers is required for obvious reasons. In a store buffer graph encoding, these states are represented as program locations only and thus, simplify both, the invariant and the actual reasoning in the theorem prover.

However, store buffer graphs have also a drawback. Generally, one has to deal with significantly more program locations, because program locations in a store buffer graph encoding represent the combination of an original program location with a certain symbolic state of the store buffer. Nevertheless, the relief from reasoning about store buffers simplifies not only the properties we have to deal with, but also the number of proof steps as our linearizability proofs show. This also summarizes our answer to **RQ2** posed in the beginning of this chapter. It shows that our reduction from Chapter 3 can be used to simplify and reduce effort of correctness proofs when dealing with weak memory semantics. Although it is difficult to generalize from two case studies, we have to take them as promising results, at least. Please note that each of the linearizability case studies required days to weeks of effort to setup all specifications and iterate through several proof attempts until all proofs went through. The latter is also the reason why we can only present two case studies here.

6.3 Related Work and Discussion

In previous chapters, we have discussed different lines of research. In Chapter 2.7, we discussed different frameworks for comparison of memory models, but also for exposing programs to weak memory model semantics. In Chapter 3.3.3, we discussed closely related work to our proposed reduction towards SC programs, which incorporate behavior of the programs under weak memory semantics. In Chapter 5.2.1, we discussed different methods for linearizability verification under SC semantics. While there has been a lot of research in the last two decades on linearizability and weak memory models, only few combined both lines of research.

The most important work on this combination was already discussed in Chapter 5. It is the adaptation of the original definition of linearizability [HW90] to weak memory models, TSO-to-TSO linearizability [BGMY12], TSO-to-SC [GMY12], All-to-SC [BDG13] linearizability and TSO linearizability [DSD14, DSGD17]. In fact, all of these papers can also be viewed as a proof method for their respective definition of linearizability under weak memory models.

However, the tool support and verification techniques for the rather new adaptations of linearizability to weak memory models have been very limited so far. Thanks to our reduction from Chapter 3, we were able to adapt some existing linearizability verification techniques (developed under SC assumption) to the setting of weak memory models. With our approach from Section 6.1 and which was published in [TMW13], we were one of the first to provide a model checking approach for TSO linearizability. With the presented alternative model checking idea based on history checking, we also provide an approach that fits to the other variations of linearizability and which can be also extended to other correctness criteria. However, for infinite state programs like most concurrent data structures, model checking usually can only show presence of bugs in concurrent data structure, not their correctness. Thus, it can be seen as an instance of systematic testing. For proving linearizability under SC, there are plenty of methods [DD15], some of which we discussed in Chapter 5.2.1 already. With our reduction towards SC programs, we enable many of them also to be carried out in a weak memory setting.

Few other approaches besides the ones that we have discussed deeply in the previous chapter focus on linearizability under weak memory models. In essence, linearizability is just a safety property (although being an important one) and most verification approaches that are sensitive to weak memory semantics focus on safety properties in general, not particularly linearizability. For instance, they focus on certain aspects that can be helpful in verification, e.g., robustness [BMM11, Der15], data-race-freedom [CS10] or fence inference [KVY12]. Of course, if a program can only examine SC behavior, even though the underlying architecture provides weak semantics, then verification does not need to take weak memory models into account. A similar argument applies to data-race-freedom, which makes reasoning about weak memory semantics obsolete. As mentioned earlier, requiring robustness or data-race-freedom can be too strict as one may want to allow for non-SC behavior for performance reasons. Furthermore, many concurrent data structures contain data races intentionally, which disqualifies data-race-freedom as a general solution.

Either way, if a program provides non-SC behavior or is simply incorrect (e.g., not linearizable) under weak memory models, then it most likely requires additional fence instructions. The latter are meant to enforce program order during execution and thus avoid some of the effects due to weak memory models. The main question

is where to put fences in a program. Fences are expensive instructions and therefore one wants to minimize their usage. Several approaches have been developed for fence inference. Most approaches for verification under weak memory models can also be used for inference of fence placement in a program. Essentially any counterexample to a property that appears only under weak semantics, but not under SC, will provide insights on where to put fences in order to avoid it.

Three approaches target fence inference particularly [BAM07, LNP+12, KVY12]. Burckhardt et al. [BAM07] compares SC execution traces with those exhibited from an artificial memory model *Relaxed*. The latter is an approximate memory model of the most relaxed memory models like RMO, but leaves out some of the behavior that is possible under Power. The approach is based on bounded model checking using a SAT solver. Liu et al. [LNP+12] try to infer fence placement by dynamic fences synthesis. They provide a tool framework that automatically repairs programs if they lead to previously specified incorrect behavior. To this end, they identify possible reorderings with their own scheduler. If incorrect behavior is detected during a test run, then possible fence placements are determined with help of a SAT solver. Their framework also supports linearizability [HW90], but the authors do not clarify on their interpretation of linearizability under weak memory models. Their approach seems to be similar to our history-based check (see Sec. 6.1.3), but is applied dynamically during test runs. Kuperstein et al. [KVY12] aim for automatic fence inference and focus particularly on the efficiency of fence placements. Starting at a reachable state that violates a correctness property, their algorithm detects avoidable transitions, which by being removed from the transition system, make the state unreachable. Their approach iteratively computes a maximally permissive set of transitions. However, there are often multiple ways to remove transitions from the transition system by placing fences. Thus, their results are not guaranteed to be minimal.

Burnim et al. [BSS11b, BSS11a] apply extensive testing and monitoring methods in order detect non-SC behavior under TSO and PSO. Their monitoring algorithm delays writes as much as possible (in the resp. memory model) for any given execution. If the delayed executions result in a different outcome from an SC execution, then the program is not robust and thus, needs fences. Their approach is sound and complete. Their testing approach [BSS11b] is guided by potential cycles in the happens-before relations. The latter are determined dynamically and indicate non-SC behavior. Abe et al. [AUMM16] have developed a very detailed operational Promela model of TSO, PSO and RMO. Since their memory model also models instruction fetching, it is inherently more complex than our operational model and thus, does not scale well for TSO and PSO. However, the detailed model also allows them to verify programs under RMO. As mentioned earlier, other verification approaches generally focus on

checking and testing robustness or general safety properties like state reachability. The latter two approaches also fall into this category. We discussed the important work in the research are in previous chapters already.

7 Conclusion

In this Chapter, we summarize the main contributions of the thesis as well as mention some of the possible future work. We finish it with a discussion of some design decisions and concluding remarks.

7.1 Summary

Verification of concurrent data structures is difficult, even without consideration of weak memory semantics. It has been well studied over the past years and linearizability [HW90] has emerged as the de-facto standard correctness criterion. However, program verification for concurrent programs rarely considers the effects of weak memory models. The latter can cause programs to behave unexpectedly or faulty. Instead, verification approaches widely assume sequential consistency, which leaves a semantics gap between the behavior that is verified and the behavior that can be observed on actual hardware.

In order to overcome this gap, we proposed a reduction towards SC programs, the major contribution of this thesis. The reduction enables a transformation from programs under TSO or PSO into an SC program that simulates the behavior of the original program and the effects of the memory model to it. We have proven that the behavior of the original program under TSO (resp. PSO) and the transformed SC program are bisimulation equivalent. A consequence of this equivalence is that programs can be verified in their transformed SC variant with existing verification tools developed for SC, while knowing that the verification results also hold for the original program under weak memory model.

Based on our reduction, we proposed a general verification approach for con-

current programs under weak memory models. The approach involves a symbolic exploration of a program under TSO (resp. PSO). The latter results in a store buffer graph for each method, each of which represents all possible behavior of one processor executing the respective method under TSO (resp. PSO) semantics. We provide transformations from store buffer graphs to Promela models and KIV inputs. Promela models are input models to the model checker Spin and aim at finding bugs in an implementation via state space exploration. When model checking reveals no bugs, the next step is to carry out a formal correctness proof. For this step, we use the interactive theorem prover KIV. The steps up to the point of providing Promela models and KIV inputs have been implemented and automated in our tool Weak2SC.

We evaluated our reduction by comparing verification of the transformed programs against more common modeling techniques, e.g., operational memory models in case of model checking or explicit modeling of store buffers as part of a process for the theorem prover. In both cases our results look promising. In our evaluation, we used linearizability [HW90] as the correctness condition. To this end, we looked at different linearizability definitions under weak memory models, in particular to its adaptations [BGMY12, GMY12, BDG13, DSD14, DSGD17].

We presented a model checking approach that checks for TSO linearizability [DSD14, DSGD17] based on atomicity checks with the help of abstract data structures as part of the Promela model. The approach was evaluated over programs which were encoded based on store buffer graphs and on an operational memory models. For most of our case studies, we achieved better performance with the models based on store buffer graphs. Our reduction is limited to programs which have no unfenced writing loops. Their store buffer graph would be infinitely large, if constructed. If a program contains unfenced writing loops, other semantics encodings should be preferred, e.g., an operational memory model. The performance improvement was essentially achieved by avoiding complex encoding of store buffers and their behavior while also minimizing auxiliary variables as part of states. In addition, we presented an alternative idea for model checking of concurrent programs under weak memory semantics based on history checks. The approach is decoupled from memory model semantics and thus, only needs a checking procedure for the correctness of a history. In this sense it is flexible and can be used for verification of different definitions of linearizability, but also for other correctness conditions.

In our case studies for proving linearizability, we compared explicit store buffer encoding against store buffer graph based encoding of program behavior. We used the proof obligations from [DSW11b], which combined with program encoding result in a proof of TSO-to-SC linearizability [GMY12]. Again our store buffer graphs have shown advantages over the explicit store buffer encoding. In particular, we were able to achieve smaller proofs and we had to deal with less complexity in the

behavioral encoding.

For now, it remains open which of the linearizability adaptations towards weak memory models will establish as the new standard correctness criterion for concurrent data structures. Independent of what it will be, we will need practical representations of programs incorporating weak semantics that reduce complexity of a correctness proofs and improve performance of correctness checks. The reduction proposed in this thesis achieves this, albeit for a limited class of concurrent programs.

7.2 Future Work

In this Section, we want to briefly mention some of the possible future work following from this dissertation. More detailed discussions can be found in the previous chapters.

Store Buffer Graph Size The size of store buffer graphs grows quickly with the size of a program, but most importantly with the amount of non-determinism that can occur due to store buffers. So far, store buffer graphs represent all possible behavior of an operation under TSO (resp. PSO). Parts of that behavior cannot be observed by other processes due to being local behavior only. These parts could be abstracted and they also qualify for partial order reduction, which would reduce the size of the graph. There is also an interesting approach by Elmas et al. [EQS+10] which iteratively applies reduction and abstraction rules to concurrent programs in order to prove them linearizable. Their approach is only sound under SC as they say. It would be interesting to investigate how their approach can be applied to store buffer graphs, either to reduce their size or in the best case as another proof method for linearizability under weak memory models.

Pointer Alias Analysis Our implemented tool Weak2SC would benefit from a pointer-alias analysis. During construction of the store buffer graph, it is possible that two symbolic representations of the same memory location are found. If there are two representations for one memory location, then there will be two different nodes representing the same state in the store buffer graph. This can falsify the constructed store buffer graph, because our proven bisimulation equivalence holds only under the assumption of each memory location being represented by a unique symbol. Pointer-alias analysis could help identify such cases. If identified, we could treat the seemingly different nodes in the store buffer graph as potentially equivalent, and thus restore validity of the store buffer graph.

Other Verification Languages So far, we have focused on Promela for Spin as a model checker and KIV for theorem proving. It would be interesting to see how other verification tools handle encodings of store buffer graphs. In particular, we think a

symbolic model checker could further improve verification results due to its ability to handle sets of states in contrast to Spin as an explicit state model checker. Generating store buffer graph encodings in other verification languages can be of great help in general. Not only would it help with weak memory sensitive verification, but also automate derivation of program encodings from LLVM IR to the respective language. As of now, the latter are usually the result of manual encoding and thus error prone.

Further Automation We are ultimately interested in linearizability proofs, which, even with the help of the transformation provided by Weak2SC, remain tedious and full of manual tasks. One such task is finding and stating an invariant that enables a linearizability proof to complete. Even with a correct idea of an invariant, mistakes are made frequently during its formalization and are noticed often only after hours have been spent on a proof attempt. With the combination of a model checker and a theorem prover, one could formalize the invariant in the theorem prover and check it with the model checker before manual effort on a proof attempt is spent. However, this task can be very complex as the expressiveness of the input languages to a model checker and a theorem prover can differ due to different domains. Thus, it can be challenging to keep both models consistent, especially throughout incremental modification.

Method Boundaries In our store buffer graphs, we assume fences at the beginning and the end of a method. Such fences can be necessary in order to make sure a data structure is TSO-to-SC linearizable. However, they are not realistic in general and instruction reordering can occur across method boundaries due to weak memory semantics. In future extensions, we therefore need a possibility to specify initial store buffer content. Also this content needs to be considered when other methods are called from within a method. Considering this in the store buffer graph construction means to weave in one store buffer graph into the other and to allow for some additional reorderings w.r.t. memory model semantics. Currently, this can only be achieved by copying the code of two consecutive method into one method and by constructing the store buffer graph of the combined method. Although not particularly challenging, the support of reordering across method boundaries would further improve practicality of our approach and its implementation, Weak2SC.

Concurrency IDE When developing concurrent programs one has to be aware of the effects by weak memory models. At the best, developers should be provided with feedback while they are still programming, similar to highlighting of compile errors in the code in most IDEs. Most IDEs do not differentiate between sequential and concurrent program code. The visualization remains the same in both cases. We suggest using visualizations like store buffer graphs or similar graph like structures that enable visualization of possible reorderings under weak memory models. As argued before, these can help in understanding where to put synchronization primitives in

the code. Store buffer graphs are great candidates for such a visualization, since they are process local, i.e., tools do not have to be aware of other processes and they can be computed quickly. However, for a general visualization in an IDE, we would first have to find a finite representation of unfenced writing loops in a store buffer graph.

7.3 Design Decisions

Our approach aims at verification of concurrent data structures. For this we had to answer several critical questions. 1. How do we represent their behavior? 2. What is their correct behavior and 3. how can we verify it? In the end, we also wanted to have a practical tool that helps with automation of tedious steps in the verification process.

From the very beginning it was clear that our behavioral models had to consider weak memory models, in order to close the gap between the SC semantics that is assumed by standard verification methods and the weak semantics provided by the, nowadays, ubiquitous multicore processors. Since there is no consideration of high-level programming language constructs in the formalization of weak memory models, our choice was obvious in taking a compiled program (LLVM IR) as a starting point for verification. We also had an attempt of defining high-level language semantics incorporating TSO semantics [DTDW13], but we must admit that it is very challenging to define such semantics on a sound basis, because compilers can interfere in too many different ways with the program and all of them must be considered.

Our first choice for modeling program behavior was to use operational memory models (Promela models for Spin), because they impose barely any restrictions, except for being bounded by the length of store buffers. However, in a sense, an operational memory model is nothing but a detailed abstraction of actual multicore processor hardware. Thus, a developer using such a model still has to have expertise in weak memory models. More importantly, standard verification tools rely on SC semantics. In order to be able to use them, we would have to provide an operational memory model to each of them, written in the respective input language. Driven by these issues, we developed our reduction by which we were truly be able to get rid of store buffers in models of program behavior, albeit for limited class of programs. These models do not require expertise in weak memory models, because they assume SC semantics and because of this, they can also be verified with standard verification tools.

The second and third questions deal with correctness of concurrent data structures under weak memory models. Under SC, linearizability as defined by Herlihy and Wing [HW90] evolved as the de-facto standard correctness condition for con-

current data structures. However, under weak memory models there is no such standard, yet and as we have discussed in depth (see Ch. 5), there are several interpretations of linearizability under weak memory models. We do prefer TSO-to-SC (resp. All-to-SC) linearizability [GMY12, BDG13], because it is the closest interpretation to the original definition [HW90] and it provides strong guarantees. We also use this variant in our linearizability proofs. Nonetheless, it can be too strict in some cases, which is also the reason why we proposed a different notion of linearizability in [TMW13] that was later formalized as TSO linearizability [DSD14, DSGD17]. Our model checking approach [TMW13] checks for TSO linearizability and thanks to our reduction, can now also be verified by tools assuming SC semantics.

Since we wanted to provide a practical tool implementation of our approach, we wanted allow software developers to integrate it into existing development processes. Eclipse IDE is a common software developer platform to which our tool, Weak2SC, is an extension. We developed Weak2SC following model-driven development standards from parsing the LLVM IR code to the output encoding of transformed SC program. Thus, we also enable for seamless integration of future extensions of Weak2SC, e.g., definitions of other store buffer graph encodings in terms of other input languages to other verification tools.

7.4 Concluding Thoughts

With this thesis, we provide a verification approach for concurrent programs and in particular concurrent data structures under TSO and PSO. Currently, more relaxed memory models like Power and ARM are becoming more and more ubiquitous and we expect this trend to continue. The semantics of these processors are more relaxed, because some processor cores share their store buffers (or even caches) and some do not. These memory models allow for even more non-determinism than TSO or PSO. With an ever growing number of cores in a multicore processor, these memory models are here to stay and we need to tame their complexity. In the near future, we expect this to be one of the major challenges in concurrent program verification. Brute-force state space exploration of programs executed under such memory models is not sufficient, as it does not scale well, even for small programs. Instead, we think abstractions or programming disciplines will be needed, which guarantee stricter execution semantics of programs executed under weaker memory models, similar to robustness [BDM13], but not necessarily SC.

A Proofs

A.1 Behavioral Equivalence

Theorem 1. *Let P be a program with fenced or write-free loops only and with no unfenced wd-chains and $MM \in \{PSO, TSO\}$ a memory model. Then*

$$lts_{MM}(P) \sim_\ell lts_{SC}(w2sc(P, MM)) \,.$$

Proof of Local Bisimilarity: The general proof idea is to make a case distinction over all types of transitions. For each transition type, we prove that it simulates a transition of the respective other transition system and that the states reached are again related by the bisimulation relation. Our proof proceeds for both memory models (TSO and PSO) in one go and thus, we have additional case distinctions where TSO and PSO differ in their semantics.

Let $lts_{MM}(P) = (S, \rightarrow_1, S_0)$ and $lts_{SC}(w2sc(P, MM)) = (Q, \rightarrow_2, Q_0)$ be the transition systems for program P, with $MM \in \{TSO, PSO\}$. We use the following relation in our proof:

$$\begin{aligned}
\mathcal{R}_{MM} := \{(s, q) \mid \quad & s = (pc_1, sb_1, reg_1), q = (pc_2, sb_2, reg_2) \\
& \wedge\, pc_1 = first(pc_2) && \text{(C1)} \\
& \wedge\, sb_2 = \varnothing && \text{(C2)} \\
& \wedge\, sb_1 = conc_{MM}(reg_2, second(pc_2)) && \text{(C3)} \\
& \wedge\, \forall\, r \in Reg_1 : reg_1(r) = reg_2(r)\} && \text{(C4)}
\end{aligned}$$

and the concretization function:

$$
\begin{aligned}
conc_{TSO}(reg, \langle\,\rangle) &= \langle\,\rangle \\
conc_{TSO}(reg, \langle (x,n)\rangle \frown sb) &= \langle (x,n)\rangle \frown conc_{TSO}(reg, sb) \\
conc_{TSO}(reg, \langle (x,r)\rangle \frown sb) &= \langle (x, reg(r))\rangle \frown conc_{TSO}(reg, sb) \\
concVar(reg, \langle\,\rangle) &= \langle\,\rangle \\
concVar(reg, \langle n\rangle \frown sb) &= \langle n\rangle \frown concVar(reg, sb) \\
concVar(reg, \langle r\rangle \frown sb) &= \langle reg(r)\rangle \frown concVar(reg, sb) \\
conc_{PSO}(reg, sb)(v) &= concVar(reg, sb(v))
\end{aligned}
$$

Please note that we gave each line of the bisimulation relation a separate tag C1-C4. We will use them to clarify what we assume at certain steps of the proof. We place them in bold font, **C1-C4**, in order to mark location in the proof, where we think that we have shown them to hold for the primed pair of states.

Initial states:

Let $s_0 = (\ell_0, \langle\,\rangle, reg_1)$ be an initial state in S_0.
Then by construction of $w2sc(P, MM)$, there exists a corresponding initial state $q_0 = ((\ell_0, \langle\,\rangle), \varnothing, reg_2)$ in Q_0, $first(q_0(pc)) = s_0(pc) = \ell_0$ and $conc_{MM}(second(q_0(pc))) = s_0(sb) = \langle\,\rangle$. Furthermore, $s_0 \models Init$ and $q_0 \models Init'$ and $\forall\, r \in Reg_1$: $reg_1(r) = reg_2(r)$. Thus, $(s_0, q_0) \in \mathcal{R}_{MM}$.

By construction of $w2sc(P, MM)$, we can find for any initial state $q_0 \in Q_0$ its corresponding initial state $s_0 \in S_0$. Hence, the other direction also holds true.

Mutual simulation — ($lts_{SC}(w2sc(P, MM))$ simulates $lts_{MM}(P)$):
Let $(s, q) \in \mathcal{R}_{MM}$ with $s = (pc_1, sb_1, reg_1)$, $q = (pc_2, sb_2, reg_2)$. Since $(s, q) \in \mathcal{R}_{MM}$, we know $pc_1 = first(pc_2)$, $sb_2 = \varnothing$, $sb_1 = conc_{MM}(reg_2, second(pc_2))$, and $\forall\, r \in Reg_1$: $reg_1(r) = reg_2(r)$. Thus, we can safely assume $s = (\ell, sb_1, reg_1)$, $q = ((\ell, ssb), \varnothing, reg_2)$, where ℓ is some arbitrary but fixed program location and ssb is some symbolic store buffer value s.t. $conc_{MM}(reg_2, ssb) = sb_1$.
Now, assume $s \xrightarrow[mem,mem']{lab}_1 s'$ in $lts_{MM}(P)$. We distinguish the cases based on the type of lab:

- $lab = (r := n)$:

  The label can be the result of three different cases:

  1. A simple local assignment, in which case $n = reg_1(expr)$, where $expr$ is the expression or constant, which is evaluated and assigned to the

to register r. Then P has an operation $COP_s = (pc = \ell \wedge pc' = \ell' \wedge r := expr)$ and by sem. def. $s' = (\ell', reg_1', sb)$ with $reg_1' = reg_1[r \mapsto reg_1(expr)]$ and $mem' = mem$.

By construction, then $w2sc(P)$ has an operation $COP_q = (pc = (\ell, ssb) \wedge pc' = (\ell', ssb) \wedge r := expr)$. **(C1)**

Then, by sem. def., $lts_{SC}(w2sc(P, MM))$ contains a transition $q \xrightarrow[mem_q, mem_q']{r:=reg_2'(expr)}_2 q'$ and $q' = ((\ell', ssb), \varnothing, reg_2')$ with $reg_2' = reg_2[r \mapsto reg_2(expr)]$.**(C2)**

By sem. def. we can choose $mem = mem_q$, from which $mem' = mem_q'$ follows, immediately.

By assumption (C4), we know $\forall\, r \in Reg_1 : reg_1(r) = reg_2(r)$. This implies $reg_1(expr) = reg_2(expr)$ and $\forall\, r \in Reg_1 : reg_1'(r) = reg_2'(r)$. **(C4)**

By construction, $w2sc(P, MM)$ has no write-def-chains and thus, $r \notin ssb$ holds and consequently $\forall\, r' \in Reg_2 : r' \neq r \Rightarrow reg_2(r') = reg_2'(r')$ holds. Thus, $conc_{MM}(reg_2, ssb) = conc_{MM}(reg_2', ssb) = sb_1$ **(C3)** and hence, $(s', q') \in \mathcal{R}_{MM}$.

2. The label corresponds to local read operation. Then P has an operation $COP_s = (pc = \ell \wedge pc' = \ell' \wedge read(x, r))$ and $x \in sb_1$, $n = lst_{MM}(x, sb_1)$.

   By sem. def. $sb_1 = sb_1'$, $reg_1'[r \mapsto lst_{MM}(x, sb_1)]$ and $mem = mem'$.

   By construction, then $w2sc(P, MM)$ has either an operation $COP_q = (pc = (\ell, ssb) \wedge pc' = (\ell', ssb) \wedge r := r_{src})$, $r_{src} = lst_{MM}(x, ssb)$ or if the entry in the symbolic store buffer is a constant, then $COP_q = (pc = (\ell, ssb) \wedge pc' = (\ell', ssb) \wedge r := n_q)$, $n_q = lst_{MM}(x, ssb)$. **(C1)**

   Then, by sem. def., $lts_{SC}(w2sc(P, MM))$ has a transition $q \xrightarrow[mem_q, mem_q']{r:=reg_2(r_{src})}_2 q'$ (resp. $q \xrightarrow[mem_q, mem_q']{r:=n_q}_2 q'$) and $q' = ((\ell', ssb), \varnothing, reg_2')$ with $reg_2' = reg_2[r \mapsto reg_2(r_{src})]$ (resp. $reg_2' = reg_2[r \mapsto n_q]$).**(C2)**

   By sem. def. we can choose $mem = mem_q$, from which $mem' = mem_q'$ follows, immediately.

   By assumption ($(s, q) \in \mathcal{R}_{MM}$), the latest value for x is $n = lst_{MM}(x, sb_1) = lst_{MM}(x, conc_{MM}(reg_2, ssb))$ and $n = reg_2(r_{src})$ (resp. $n = n_q$) follows immediately from it. Also $\forall\, r \in Reg_1 : reg_1'(r) = reg_2'(r)$ follows from it. **(C4)**

   By construction, $w2sc(P, MM)$ has no wd-chains and thus, $r \notin ssb$. Thus, also $sb_1 = sb_1' = conc_{MM}(reg_2, ssb) = conc_{MM}(reg_2', ssb)$ **(C3)** and $(s', q') \in \mathcal{R}_{MM}$.

3. The label corresponds to a write source of a write-def-chain. Then P has an operation $COP_s = (pc = \ell \wedge pc' = \ell' \wedge write(x, r_{org}))$. Then by sem. def., $s' = (\ell', reg_1', sb')$ with $n = reg_1(r_{org})$, $mem' = mem$ and under TSO $sb' = sb \frown \langle(x, n)\rangle$, resp. under PSO $sb' = sb[x \mapsto sb \frown \langle n\rangle]$.

   By construction, then $w2sc(P, TSO)$ (resp. $w2sc(P, PSO)$) has an operation $COP_q = (pc = (\ell, ssb) \wedge pc' = (\ell', ssb \frown \langle(x, r)\rangle) \wedge r := r_{org})$ (resp. $COP_q = (pc = (\ell, ssb) \wedge pc' = (\ell', ssb[x \mapsto ssb(x) \frown \langle r\rangle]) \wedge r := r_{org})$). **(C1)**

   Let $n_q = reg_2(r_{org})$. Then by sem. def., $lts_{SC}(w2sc(P, MM))$ has a transition $q \xrightarrow[mem_q, mem_q']{r:=n_q}_2 q'$ and for TSO $q' = ((\ell', ssb \frown \langle(x, r)\rangle), \varnothing, reg_2')$ (resp. for PSO $q' = ((\ell', ssb(x) \frown \langle r\rangle), \varnothing, reg_2')$) and $reg_2' = reg_2[r \mapsto n_q]$. **(C2)**

   By sem. def. we can choose $mem = mem_q$, from which $mem' = mem_q'$ follows, immediately. By assumption (C4), $reg_1(r_{org}) = reg_2(r_{org}) = reg_2'(r)$ holds. Please note that $r \notin Reg_1$ as it is an auxiliary variable. Thus, $\forall\, r \in Reg_1 : reg_1'(r) = reg_2'(r)$ and furthermore, $n = n_q$ follows. **(C4)**

   From the above and assumption (C3), for TSO we can derive $sb_1' = sb_1 \frown \langle(x, n)\rangle = conc_{MM}(reg_2', ssb \frown \langle(x, r)\rangle)$ and resp. for PSO $sb_1' = sb_1(x) \frown \langle n\rangle = conc_{MM}(reg_2', ssb[x \mapsto ssb(x) \frown \langle r\rangle])$. **(C3)**

   Hence, $(s', q') \in \mathcal{R}_{MM}$.

- $lab = b$

  Then P has an operation $COP_s = (pc = \ell \wedge pc' = \ell' \wedge bexpr)$ and by sem. def. $s' = (\ell', reg_1', sb_1')$ with $reg_1' = reg_1$ and $sb_1' = sb_1$ and $mem' = mem$ and $b = reg_1(bexpr)$.

  By construction, then $w2sc(P, MM)$ has an operation $COP_q = (pc = (\ell, ssb) \wedge pc' = (\ell', ssb) \wedge bexpr)$. **(C1)**

  Then by sem. def. $lts_{SC}(w2sc(P, MM))$ contains a transition $q \xrightarrow[mem_q, mem_q']{b_q}_2 q'$ and $q' = ((\ell', ssb), \varnothing, reg_2')$, $reg_2' = reg_2$ and $b_q = reg_2(bexpr)$. **(C2)**

  By sem. def. we can choose $mem = mem_q$, from which $mem' = mem_q'$ follows, immediately.

  By assumption (C4), $reg_1(bexpr) = reg_2(bexpr)$ holds and thus also $b = b_q$ holds. Registers and store buffers are not modified. Hence, register equality

$(\forall\, r \in Reg_1 : reg'_1(r) = reg'_2(r))$ **(C4)** is preserved and $conc_{MM}(reg_2, ssb) = sb'_1$. **(C3)** Hence, $(s', q') \in \mathcal{R}_{MM}$.

- $lab = skip$:

  Then P has an operation $COP_s = (pc = \ell \wedge pc' = \ell' \wedge op)$ with $op \in \{fence, write(x, r), write(x, n)\}$. We distinguish the cases based on op:

  - $op = fence$: Then by sem. def., $s' = (\ell', reg'_1, \langle\,\rangle)$ with $reg'_1 = reg_1$ and $mem' = mem$.

    By construction, then $w2sc(P)$ has an operation $COp_q = (pc = (\ell, \langle\,\rangle) \wedge pc' = (\ell', \langle\,\rangle) \wedge skip)$. **(C1)**

    Then by sem. def., $lts_{SC}(w2sc(P, MM))$ has a transition $q \xrightarrow[mem_q, mem'_q]{skip}_2 q'$ and $q' = ((\ell', \langle\,\rangle), \varnothing, reg'_2)$ with $reg'_2 = reg_2$. **(C2)**.

    By assumption (C3 and C4), and because $reg'_1 = reg_1$ and $reg'_2 = reg_2$ we also have $\forall\, r \in Reg_1 : reg'_1(r) = reg'_2(r)$ **(C4** and **C3)**.

    By sem. def. we can choose $mem = mem_q$, from which $mem' = mem'_q$ follows, immediately.

    Hence, $(s', q') \in \mathcal{R}_{MM}$.

  - $op = write(x, r)$: Then by sem. def., $s' = (\ell', reg'_1, sb'_1)$ with $reg'_1 = reg_1$ and under TSO $sb'_1 = sb_1 \frown \langle (x, reg_1(r)) \rangle$ (resp. under PSO $sb'_1 = sb_1[x \mapsto sb_1(x) \frown \langle reg_1(r) \rangle]$) and $mem' = mem$.

    Furthermore, by our definition of labels (Def.12), the $write(x, r)$ cannot be the source of a write-def-chain. We label those like local assignments.

    By construction, then $w2sc(P, TSO)$ has an operation $COP_q = (pc = (\ell, ssb) \wedge pc' = (\ell', ssb \frown \langle (x, r) \rangle) \wedge skip)$ and $w2sc(P, PSO)$ an operation $COP_q = (pc = (\ell, ssb) \wedge pc' = (\ell', ssb[x \mapsto ssb(x) \frown \langle r \rangle]) \wedge skip)$. **C1**

    Then by sem. def., $lts_{SC}(w2sc(P, MM))$ contains a transition $q \xrightarrow[mem_q, mem'_q]{skip}_2 q'$ and under TSO $q' = ((\ell', ssb \frown \langle (x, r) \rangle), \varnothing, reg'_2)$ (resp. under PSO $q' = ((\ell', ssb[x \mapsto ssb(x) \frown \langle r \rangle]), \varnothing, reg'_2)$) with $reg'_2 = reg_2$. **C2, C4**

    By sem. def. we can choose $mem = mem_q$, from which $mem' = mem'_q$ follows, immediately. By assumption (C3 and C4), we know $reg_1(r) = reg_2(r)$. Thus, we get $sb'_1 = conc_{TSO}(reg'_2, ssb \frown \langle (x, r) \rangle)$ and $sb'_1 = conc_{PSO}(reg'_2, ssb[x \mapsto ssb(x) \frown \langle r \rangle])$. **C3**

    Hence, $(s', q') \in \mathcal{R}_{MM}$.

– $op = write(x, n)$: Then by sem. def., $s' = (\ell', reg_1', sb_1')$ with $reg_1' = reg_1$ and under TSO $sb_1' = sb_1 \frown \langle (x, n) \rangle$ (resp. under PSO $sb_1' = sb_1[x \mapsto sb_1(x) \frown \langle n \rangle]$) and $mem' = mem$.

Furthermore, by our definition of labels (Def.12), the $write(x, n)$ cannot be the source of a write-def-chain. We label those like local assignments.

By construction, then $w2sc(P, TSO)$ has an operation $COP_q = (pc = (\ell, ssb) \wedge pc' = (\ell', ssb \frown \langle (x, n) \rangle) \wedge skip)$ and $w2sc(P, PSO)$ an operation $COP_q = (pc = (\ell, ssb) \wedge pc' = (\ell', ssb[x \mapsto ssb(x) \frown \langle n \rangle]) \wedge skip)$. **(C1)**

Then by sem. def., $lts_{SC}(w2sc(P, MM))$ contains a transition $q \xrightarrow[mem_q, mem_q']{skip}_2 q'$ and under TSO $q' = ((\ell', ssb \frown \langle (x, n) \rangle), \varnothing, reg_2')$ (resp. under PSO $q' = ((\ell', ssb[x \mapsto ssb(x) \frown \langle n \rangle]), \varnothing, reg_2')$) with $reg_2' = reg_2$. **(C2)**

By assumption (C4) and because of $reg_1' = reg_1$ and $reg_2' = reg_2$, we conclude $\forall\, r \in Reg_1 : reg_1'(r) = reg_2'(r)$. **(C4)**

By sem. def. we can choose $mem = mem_q$, from which $mem' = mem_q'$ follows, immediately. By assumption (C3 and C4) and from the above arguments, we get
$sb_1' = conc_{TSO}(reg_2', ssb \frown \langle (x, n) \rangle)$ and
$sb_1' = conc_{PSO}(reg_2', ssb[x \mapsto ssb(x) \frown \langle n \rangle])$. **(C3)**
Hence, $(s', q') \in \mathcal{R}_{MM}$.

- $lab = write(x, n)$:

  Then P has an operation $COP_s = \mathit{flush}$, which is enabled and by sem. def., $sb \neq \langle\,\rangle$, $s' = (\ell, reg_1', sb_1')$ with $reg_1' = reg_1$ and $sb_1 = \langle (x, n) \rangle \frown sb_1'$ and $mem' = mem[x \mapsto n]$.

  By construction, there are two cases, depending on whether value n stems from a register or a constant:

  – if constant, then by construction $w2sc(P, MM)$ contains an operation $COP_q = (pc = (\ell, ssb) \wedge pc' = (\ell, ssb') \wedge write_{sc}(x, n))$ with $ssb = \langle (x, n) \rangle \frown ssb'$ in case of TSO, and in case of PSO $ssb' = ssb[x \mapsto ssb(x)']$ with $ssb(x) = \langle n \rangle \frown ssb'(x)$. **(C1)**

    Then by sem. def., $lts_{SC}(w2sc(P, MM))$ contains a transition $q \xrightarrow[mem_q, mem_q']{write(x,n)}_2 q'$ and $q' = ((\ell', ssb'), \varnothing, reg_2')$ and $reg_2 = reg_2'$. **(C2)**

    By assumption (C4) and because of $reg_1' = reg_1$ and $reg_2' = reg_2$, we conclude $\forall\, r \in Reg_1 : reg_1'(r) = reg_2'(r)$. **(C4)**

By assumption (C4 and C3) and from the above arguments, we can derive for TSO:

$sb_1 = conc_{MM}(reg_2', ssb)$

$\langle (x, n) \rangle \frown sb_1 = conc_{MM}(reg_2', \langle (x, n) \rangle \frown ssb')$

$sb_1' = conc_{MM}(reg_2', ssb')$.

and for PSO:

$sb_1 = conc_{MM}(reg_2', ssb)$

$sb_1[x \mapsto \langle n \rangle \frown sb_1(x)] = conc_{MM}(reg_2', ssb[x \mapsto \langle n \rangle \frown ssb'(x)])$

$sb_1' = conc_{MM}(reg_2', ssb')$. **(C3)**

By sem. def. we can choose $mem = mem_q$, from which $mem' = mem_q' = mem[x \mapsto n]$ follows.

Hence, $(s', q') \in \mathcal{R}_{MM}$.

– if register, then by construction $w2sc(P, MM)$ contains an operation $COP_q = (pc = (\ell, ssb) \wedge pc' = (\ell, ssb') \wedge write_{sc}(x, r))$ with $ssb = \langle (x, r) \rangle \frown ssb'$ in case of TSO,
and in case of PSO $ssb' = ssb[x \mapsto ssb(x)']$ with $ssb(x) = \langle r \rangle \frown ssb'(x)$. **(C1)**

Then by sem. def., $lts_{SC}(w2sc(P, MM))$ contains a transition $q \xrightarrow[mem_q, mem_q']{write(x, reg_2(r))}_2 q'$ and $q' = ((\ell', ssb'), \varnothing, reg_2')$ and $reg_2 = reg_2'$. **(C2)**

By assumption (C4) and because of $reg_1' = reg_1$ and $reg_2' = reg_2$, we conclude $\forall\, r \in Reg_1 : reg_1'(r) = reg_2'(r)$. **(C4)**

From C3 and C4, we get $reg_2(r) = n$, since $sb_1 = conc_{MM}(reg_2, ssb)$ holds initially. By assumption (C4 and C3) and from the above arguments, we can derive for TSO:

$sb_1 = conc_{MM}(reg_2', ssb)$

$\langle (x, n) \rangle \frown sb_1 = conc_{MM}(reg_2', \langle (x, r) \rangle \frown ssb')$

$sb_1' = conc_{MM}(reg_2', ssb')$.

and for PSO:

$sb_1 = conc_{MM}(reg_2', ssb)$

$sb_1[x \mapsto \langle n \rangle \frown sb_1(x)] = conc_{MM}(reg_2', ssb[x \mapsto \langle r \rangle \frown ssb'(x)])$

$sb_1' = conc_{MM}(reg_2', ssb')$. **(C3)**

By sem. def. we can choose $mem = mem_q$, from which $mem' = mem_q' = mem[x \mapsto n]$ follows.

Hence, $(s', q') \in \mathcal{R}_{MM}$.

- $lab = read(x, r)$:

Then P has an operation $COP_s = (pc = \ell \wedge pc' = \ell' \wedge read(x, r))$ with $x \notin sb_1$, and by sem. def. $s' = (\ell', reg_1', sb_1')$ with $sb_1' = sb_1$ and $reg_1' = reg_1[r \mapsto mem(x)]$ and $mem' = mem$.

By construction, then $w2sc(P, MM)$ has an operation $COP_q = (pc = (\ell, ssb) \wedge pc' = (\ell', ssb) \wedge read(x, r))$. **(C1)**

Then by sem. def., $lts_{SC}(w2sc(P, MM))$ contains a transition $q \xrightarrow[mem_q, mem_q']{read(x,r)}_2 q'$ and $q' = ((\ell', ssb), \varnothing, reg_2')$ and $reg_2' = reg_2[r \mapsto mem_q(x)]$. **(C2)**

By sem. def. we can choose $mem = mem_q$, from which $mem' = mem_q'$ follows, immediately. From this combined with assumption C4, register equality follows again, $\forall\, r \in Reg_1 : reg_1'(r) = reg_2'(r)$. **(C4)**

By construction, $w2sc(P, MM)$ has no write-def-chains and thus, $r \notin ssb$ holds. Thus, $sb_1' = conq_{MM}(reg_2', ssb)$ follows. **(C3)**

Hence, $(s', q') \in \mathcal{R}_{MM}$.

Mutual simulation –
($lts_{MM}(P)$ **simulates** $lts_{SC}(w2sc(P, MM))$**)**:
Let $(s, q) \in \mathcal{R}_{MM}$ with $s = (pc_1, sb_1, reg_1)$, $q = (pc_2, sb_2, reg_2)$. Since $(s, q) \in \mathcal{R}_{MM}$, we know $pc_1 = first(pc_2)$, $sb_2 = \varnothing$, $sb_1 = conc_{MM}(reg_2, second(pc_2))$, and $\forall\, r \in Reg_1$: $reg_1(r) = reg_2(r)$. Thus, we can safely assume $s = (\ell, sb_1, reg_1)$, $q = ((\ell, ssb), \varnothing, reg_2)$, where ℓ is some arbitrary but fixed program location and ssb is some symbolic store buffer value s.t. $conc_{MM}(reg_2, ssb) = sb_1$.
Now, assume $q \xrightarrow[mem, mem']{lab}_2 q'$ in $lts_{SC}(w2sc(P, MM))$. We distinguish the possible cases based on the type of lab:

- $lab = r := n$:

  Then $w2sc(P)$ has an operation $COP_q = (pc = (\ell, ssb) \wedge pc' = (\ell', ssb') \wedge r := expr)$ and by SC semantics def. $q' = ((\ell', ssb'), \varnothing, reg_2')$ with $reg_2' = reg_2[r \mapsto n]$, $n = reg_2(expr)$ and $mem' = mem$. **(C2)**

  By construction of $w2sc(P)$, the label can only be generated in three cases (1. local assignment, 2. a local read or 3. for the write source of a write-def-chain). Based on the case, we can further strengthen our assumptions on the operation COP_q, in particular on the value ssb':

  1. Then P has an operation $COP_s = (pc = \ell \wedge pc' = \ell' \wedge r := expr)$. **(C1)**

     Then, by sem. def. $lts_{MM}(P)$ contains a transition $s \xrightarrow[mem_s, mem_s']{r:=reg(expr)}_1 s'$ with $s' = (\ell', sb_1', reg_1')$, $sb_1' = sb_1$ and $reg_1' = reg_1[r \mapsto reg_1(expr)]$

and $mem'_s = mem_s$. By sem. def. we can choose $mem = mem_s$, from which $mem' = mem'_s$ follows, immediately.

By construction of the sb-graph, we know that $ssb' = ssb$ must hold. By assumption (C4), we get $reg_1(expr) = reg_2(expr) = n$ and hence can also derive $\forall\, r \in Reg_1 : reg'_1(r) = reg'_2(r)$. **(C4)**

By construction $w2sc(P)$ is free of wd-chains, and thus $r \notin ssb$. Thus, $sb'_1 = conq_{MM}(reg'_2, ssb')$ follows. **(C3)**

Hence, $(s', q') \in \mathcal{R}_{MM}$.

2. Then P has an operation $COP_s = (pc = \ell \wedge pc' = \ell' \wedge read(x, r))$ and by definition of labels $x \in sb_1$ must hold. **(C1)**

   Then, by sem. def. $lts_{MM}(P)$ contains a transition $s \xrightarrow[mem_s, mem'_s]{r:=n_s}{}_1 s'$ with $s' = (\ell', sb'_1, reg'_1)$, $sb'_1 = sb_1$ and $reg'_1 = reg_1[r \mapsto n]$, $n = lst_{MM}(x, sb_1)$ and $mem'_s = mem_s$. By sem. def. we can choose $mem = mem_s$, from which $mem' = mem'_s$ follows, immediately.

   By construction of the sb-graph, we know that $ssb' = ssb$ must hold. By assumption (C4), $n = n_s = lst_{MM}(x, sb) = lst_{MM}(x, conc_{MM}(reg_2, ssb))$, and then also $reg'_1(r) = reg'_2(r)$ from which we can derive $\forall\, r \in Reg_1 : reg'_1(r) = reg'_2(r)$. **(C4)**

   By construction $w2sc(P)$ is free of wd-chains, and thus $r \notin ssb$. Thus, $sb'_1 = conq_{MM}(reg'_2, ssb')$ follows. **(C3)**

   $(s', q') \in \mathcal{R}_{MM}$.

3. Then P has an operation $COp_s = (pc = \ell \wedge pc' = \ell' \wedge write(x, r_{org}))$. **(C1)**

   Then, by sem. def. $lts_{MM}(P)$ contains a transition $s \xrightarrow[mem_s, mem'_s]{r:=n_s}{}_1 s'$ with $s' = (\ell', sb'_1, reg'_1)$, $reg'_1 = reg_1$, $n_s = reg_1(r_{org})$
   and under TSO $sb'_1 = sb_1 \frown \langle (x, n_s) \rangle$
   (resp. under PSO $sb'_1 = sb_1[x \mapsto sb_1 \frown \langle n_s \rangle]$)
   and $mem'_s = mem_s$. By sem. def. we can choose $mem = mem_s$, from which $mem' = mem'_s$ follows, immediately.

   By construction of the sb-graph and of $w2sc(P)$, we know that
   under TSO $ssb' = ssb \frown \langle (x, r) \rangle$
   and under PSO $ssb' = ssb[x \mapsto ssb(x) \frown \langle r \rangle]$.
   Furthermore, we know that $expr = r_{org}$ holds for the operation COP_q.
   Please note that r in this case is our auxiliary variable ($r \notin Reg_1$).

   By assumption (C4), we get $n_s = reg_1(r_{org}) = n = reg_2(r_{org})$. Since, $r \notin Reg_1$, $reg'_1 = reg_1$, and $reg'_2 = reg_2[r \mapsto reg_2(r_{org})]$, we can still conclude that $\forall\, r \in Reg_1 : reg'_1(r) = reg'_2(r)$ must hold. **(C4)**

By construction $w2sc(P)$ is free of wd-chains, and thus $r \notin ssb$. However, $r \in ssb'$ holds.

By assumption (C4 and C3) and from $n = reg_1(r_{org}) = reg_2(r_{org}) = reg'_2(r)$, we can derive for TSO:

$sb_1 = conc_{MM}(reg'_2, ssb)$

$sb_1 \frown \langle(x, n)\rangle = conc_{MM}(reg'_2, ssb \frown \langle(x, r)\rangle)$

$sb'_1 = conc_{MM}(reg'_2, ssb')$.

and for PSO:

$sb_1 = conc_{MM}(reg'_2, ssb)$

$sb_1[x \mapsto sb_1(x) \frown \langle n\rangle] = conc_{MM}(reg'_2, ssb[x \mapsto ssb(x) \frown \langle r\rangle])$

$sb'_1 = conc_{MM}(reg'_2, ssb')$. **(C3)**

$(s', q') \in \mathcal{R}_{MM}$.

- $lab = b$

  Then $w2sc(P)$ has an operation $COP_q = (pc = (\ell, ssb) \wedge pc' = (\ell', ssb') \wedge bexpr)$ and by sem. def. $q' = ((\ell', ssb'), \varnothing, reg'_2)$ with $reg'_2 = reg_2$, $ssb' = ssb$, $mem' = mem$ and $b = reg_2(bexpr)$. **(C2)**

  By construction of the sb-graph and $w2sc(P)$, P has an operation $COp_s = (pc = \ell \wedge pc' = \ell' \wedge bexpr)$. Then, by sem. def. $lts_{MM}(P)$ contains a transition $s \xrightarrow[mem_s, mem'_s]{b_s}_1 s'$ with $s' = (\ell', sb'_1, reg'_1)$, $reg'_1 = reg_1$, $sb'_1 = sb_1$ and $b_s = reg_1(bexpr)$. By sem. def. we can choose $mem = mem_s$, from which $mem' = mem'_s$ follows, immediately. **(C1)**

  By assumption (C4), $reg_2(bexpr) = reg_1(bexpr)$ holds and thus also $b = b_s$ holds. Registers and store buffers are not modified. Thus, register equality, $\forall\, r \in Reg_1 : reg'_1(r) = reg'_2(r)$ holds. **(C4)**

  By assumption (C3) and from the above $sb'_1 = conq_{MM}(reg'_2, ssb')$ follows. **(C3)**

  Hence, $(s', q') \in \mathcal{R}_{MM}$.

- $lab = write(x, n)$:

  If the written value was obtained from a constant, then $w2sc(P)$ has an operation $COP_q = (pc = (\ell, ssb) \wedge pc' = (\ell', ssb') \wedge write_{sc}(x, n))$. If the written value was obtained from a register, then $w2sc(P)$ has an operation $COP_q = (pc = (\ell, ssb) \wedge pc' = (\ell', ssb') \wedge write_{sc}(x, r))$.

  By sem. def. $q' = ((\ell', ssb'), \varnothing, reg'_2)$ with $mem' = mem[x \mapsto n]$ and $reg'_2 = reg_2$ and if the value was obtained from a register, then also $n = reg_2(r)$. **(C2)**

Then, by construction of the sb-graph and $w2sc(P, MM)$, P has an operation $COp_s = \mathit{flush}$ that is enabled at the state s.

Then, by sem. def., $lts_{MM}(P)$ contains a transition $s \xrightarrow[mem_s, mem'_s]{write(x,n_s)}_1 s'$ with $s' = (\ell', sb'_1, reg'_1)$, with $\ell' = \ell$, $reg'_1 = reg_1$
and under TSO $sb_1 = \langle (x, n_s) \rangle \frown sb'_1$
(resp. under PSO $sb'_1 = sb_1[x \mapsto sb'_1(x)]$ where $sb_1(x) = \langle n_s \rangle \frown sb'_1(x)$)
and $mem'_s = mem_s[x \mapsto n_s]$.

By assumption (C4) and since $reg'_1 = reg_1$ and $reg'_2 = reg_2$, register equality still holds ($\forall\, r \in Reg_1 : reg'_1(r) = reg'_2(r)$). **(C4)**

By construction of sb-graph and $w2sc(P, MM)$, we can concretize the program locations of COP_q:

Under TSO we have $\ell' = \ell$, $ssb = \langle (x, n) \rangle \frown ssb'$ for a constant n and $ssb = \langle (x, r) \rangle \frown ssb'$ for a register r.
Under PSO we have $\ell' = \ell$, $ssb' = ssb[x \mapsto ssb'(x)]$ where $ssb(x) = \langle n \rangle \frown ssb'(x)$ for constant n and $ssb(x) = \langle r \rangle \frown ssb'(x)$ for a register r. **(C1)**

In any of those cases by assumption (C3), $sb_1 = conc_{MM}(reg_2, ssb)$, we get that $n_s = n$.

By sem. def. we can choose $mem = mem_s$, from which $mem' = mem'_s = mem_s[x \mapsto n_s] = mem[x \mapsto n]$ follows.

By assumption (C4 and C3) and from the above arguments, we can derive for TSO for writing constant n:

$sb_1 = conc_{MM}(reg'_2, ssb)$

$\langle (x, n) \rangle \frown sb'_1 = conc_{MM}(reg'_2, \langle (x, n) \rangle \frown ssb')$

$sb'_1 = conc_{MM}(reg'_2, ssb')$.

and for a write from a register r
$sb_1 = conc_{MM}(reg'_2, ssb)$

$\langle (x, n) \rangle \frown sb'_1 = conc_{MM}(reg'_2, \langle (x, r) \rangle \frown ssb')$

$sb'_1 = conc_{MM}(reg'_2, ssb')$.

and for PSO:

$sb_1 = conc_{MM}(reg'_2, ssb)$

$sb_1[x \mapsto \langle n \rangle \frown sb'_1(x)] = conc_{MM}(reg'_2, ssb[x \mapsto \langle n \rangle \frown ssb'(x))$

$sb'_1 = conc_{MM}(reg'_2, ssb')$.

and for a write from a register r
$sb_1 = conc_{MM}(reg'_2, ssb)$

$sb_1[x \mapsto \langle n \rangle \frown sb'_1(x)] = conc_{MM}(reg'_2, ssb[x \mapsto \langle r \rangle \frown ssb'(x))$

$sb'_1 = conc_{MM}(reg'_2, ssb')$.

(C3)

Hence, $(s', q') \in \mathcal{R}_{MM}$.

- $lab = read(x, r)$:

  Then $w2sc(P, MM)$ has an operation $COp_q = (pc = (\ell, ssb) \wedge pc' = (\ell', ssb') \wedge read_{sc}(x, r))$ and by sem. def. $q' = ((\ell', ssb'), \varnothing, reg'_2)$, $reg'_2 = reg_2[r \mapsto mem(x)]$ and $mem' = mem$. **(C2)**

  By construction of the sb-graph and $w2sc(P, MM)$, then P has an operation $COP_s = (pc = \ell \wedge pc' = \ell' \wedge read(x, r))$ and $x \notin sb_1$ (otherwise we would have a different label). **(C1)**

  Then by sem. def. $lts_{MM}(P)$ contains a transition $s \xrightarrow[mem_s, mem'_s]{read(x,r)}_1 s'$ with $s' = (\ell', sb'_1, reg'_1)$, $sb'_1 = sb_1$ and $reg'_1 = reg_1[r \mapsto mem_s(x)]$ and $mem'_s = mem_s$. By sem. def. we can choose $mem = mem_s$, from which $mem' = mem'_s$ follows, immediately.

  By assumption (C4) and by choosing equal memories $mem = mem_s$, $\forall\, r \in Reg_1 : reg'_1(r) = reg'_2(r)$ follows. **(C4)**

  By construction of the sb-graph and $w2sc(P, MM)$, we can concretize COP_q by stating $ssb' = ssb$. By assumption (C3), $sb'_1 = sb_1$, $ssb' = ssb$, we conclude that $sb_1 = conq_{MM}(reg'_2, ssb')$ still holds. **(C3)**

  Hence, $(s', q') \in \mathcal{R}_{MM}$.

- $lab = skip$:

  Then $w2sc(P, MM)$ has an operation $COP_q = (pc = (\ell, ssb) \wedge pc' = (\ell', ssb') \wedge skip)$ and by sem. def. $q' = ((\ell', ssb'), \varnothing, reg'_2)$ with $reg'_2 = reg_2$ and $mem' = mem$. **C2**

  By construction of the sb-graph and $w2sc(P, MM)$, then P has an operation $COP_s = (pc = \ell \wedge pc' = \ell' \wedge op)$ with $op \in \{fence, write(x, r), write(x, n)\}$. We distinguish the cases based on op

  - $op = fence$:
    Then by sem. def., $lts_{MM}(P)$ contains a transition $s \xrightarrow[mem_s, mem'_s]{skip}_1 s'$ with $s' = (\ell', sb'_1, reg'_1)$ and $sb'_1 = sb_1 = \langle\,\rangle \wedge reg'_1 = reg_1$ and $mem'_s = mem_s$. By sem. def. we can choose $mem = mem_s$, from which $mem' = mem'_s$ follows, immediately. **(C1)**

Then, by construction of sb-graph and $w2sc(P, MM)$, we can concretize COP_q by $ssb' = ssb = \langle\,\rangle$.

We trivially get $sb'_1 = conc_{MM}(reg'_2, ssb') = \langle\,\rangle$. **(C3)**

By assumption (C4) and because registers are not modified, $\forall\, r \in Reg_1 : reg'_1(r) = reg'_2(r)$ holds. **(C4)**

Hence, $(s', q') \in \mathcal{R}_{MM}$.

– $write(x, n)$:

Then by sem. def., $lts_{MM}(P)$ contains a transition $s \xrightarrow[mem_s, mem'_s]{skip}_1 s'$ with
$s' = (\ell', sb'_1, reg'_1)$, $reg'_1 = reg_1$,
under TSO $sb'_1 = sb_1 \frown \langle(x, n)\rangle$
(resp. under PSO $sb'_1 = sb_1[x \mapsto sb_1(x) \frown \langle n\rangle]$)
and $mem'_s = mem_s$.

By sem. def. we can choose $mem = mem_s$, from which $mem' = mem'_s$ follows, immediately. **(C1)**

Then, by construction of sb-graph and $w2sc(P, MM)$, we can concretize COP_q by $ssb' = ssb \frown \langle(x, n)\rangle$ under TSO
and $ssb' = ssb[x \mapsto ssb \frown \langle n\rangle]$ under PSO.

By assumption (C4) and since $reg'_1 = reg_1$, $reg'_2 = reg_2$, we conclude $\forall\, r \in Reg_1 : reg'_1(r) = reg'_2(r)$ holds. **(C4)**

By assumption (C4 and C3) and from the above arguments, we can derive for TSO:

$sb_1 = conc_{MM}(reg'_2, ssb)$

$sb_1 \frown \langle(x, n)\rangle = conc_{MM}(reg'_2, ssb \frown \langle(x, n)\rangle)$

$sb'_1 = conc_{MM}(reg'_2, ssb')$.

and for PSO:

$sb_1 = conc_{MM}(reg'_2, ssb)$

$sb_1[x \mapsto sb_1(x) \frown \langle n\rangle] = conc_{MM}(reg'_2, ssb[x \mapsto ssb(x) \frown \langle n\rangle])$

$sb'_1 = conc_{MM}(reg'_2, ssb')$. **(C3)**

Hence, $(s', q') \in \mathcal{R}_{MM}$.

– $write(x, r)$:

Then by sem. def., $lts_{MM}(P)$ contains a transition $s \xrightarrow[mem_s, mem'_s]{skip}_1 s'$ with
$s' = (\ell', sb'_1, reg'_1)$, $reg'_1 = reg_1$,
under TSO $sb'_1 = sb_1 \frown \langle(x, reg_1(r))\rangle$
(resp. under PSO $sb'_1 = sb_1[x \mapsto sb_1(x) \frown \langle reg_1(r)\rangle]$)
and $mem'_s = mem_s$. **(C1)**

By sem. def. we can choose $mem = mem_s$, from which $mem' = mem'_s$ follows, immediately.

Then, by construction of sb-graph and $w2sc(P, MM)$, we can concretize COP_q by
$ssb' = ssb \frown \langle (x, r) \rangle$ under TSO
and $ssb' = ssb[x \mapsto ssb \frown \langle r \rangle]$ under PSO.

By assumption (C4) and since $reg'_1 = reg_1$, $reg'_2 = reg_2$, we conclude $\forall\, r \in Reg_1 : reg'_1(r) = reg'_2(r)$ holds. **(C4)**

Let $n = reg_1(r)$, by assumption (C4 and C3) and from the above arguments, we can derive for TSO:

$sb_1 = conc_{MM}(reg_2, ssb)$

$sb_1 \frown \langle (x, n) \rangle = conc_{MM}(reg_2, ssb \frown \langle (x, r) \rangle)$

$sb'_1 = conc_{MM}(reg'_2, ssb')$.

and for PSO:

$sb_1 = conc_{MM}(reg_2, ssb)$

$sb_1[x \mapsto sb_1(x) \frown \langle n \rangle] = conc_{MM}(reg_2, ssb[x \mapsto ssb(x) \frown \langle r \rangle])$

$sb'_1 = conc_{MM}(reg'_2, ssb')$. **(C3)**

Hence, $(s', q') \in \mathcal{R}_{MM}$.

□

A.2 Compositionality

Theorem 2. *Let P_1, P_1', P_2, P_2' be sequential programs such that $lts_{MM_1}(P_j) \sim_\ell lts_{MM_2}(P_j')$, $j \in \{1,2\}$. Then*

$$lts_{MM_1}(P_1 \mid\mid P_2) \sim_g lts_{MM_2}(P_1' \mid\mid P_2') \ .$$

Proof of Compositionality: By our assumption, the processes of the parallel compositions are locally bisimulation equivalent. Thus, we can reuse this property in our global bisimulation relation. We show that from a state, in which the bisimulation relation holds, we can only reach states that are also related by the bisimulation relation.

Let $lts_{MM_1}(P_1 \mid\mid P_2) = (S, \rightarrow_1, S_0)$ and $lts_{MM_2}(P_1' \mid\mid P_2') = (Q, \rightarrow_2, Q_0)$ be the transition systems for program $(P_1 \mid\mid P_2)$ and $(P_1' \mid\mid P_2')$ respectively. We use the following relation in our proof: Let $\mathcal{R}_j$, $j \in \{1,2\}$, be the relations showing local bisimilarity of $lts_{MM_1}(P_j)$ and $lts_{MM_2}(P_j')$. Out of this, we construct the following global bisimulation relation $\mathcal{R}$:

$$\mathcal{R} := \{((mem, ls_1, ls_2), (mem, lq_1, lq_2)) \mid (ls_j, lq_j) \in \mathcal{R}_j, j \in \{1,2\}\}$$

Initial states:

- Let $s_0 = (mem, ls_{0,1}, ls_{0,2})$ an be initial state in $lts_{MM_1}(P_1 \mid\mid P_2)$ with $ls_{0,i} = (l_{0,i}, sb_i, reg_i)$ and $s_0 \models Init_1 \wedge Init_2 \wedge Init$, i.e., $ls_{0,1} \models Init_1$ and $ls_{0,2} \models Init_2$ and $mem \models Init$.

  By assumption ($\mathcal{R}_i$), there exists $lq_{0,i} = (l'_{0,i}, sb'_i, reg'_i)$ s.t. $(ls_{0,i}, lq_{0,i}) \in \mathcal{R}_i$.

  Hence, there exists $q_0 = (mem, lq_{0,1}, lq_{0,2})$, an initial state in $lts_{MM_2}(P_1' \mid\mid P_2')$ such that $q_0 \models Init_1 \wedge Init_2$. By assumption, $mem \models Init$ and thus also $q_0 \models Init$ holds.

  Thus, $(u_0, v_0) \in \mathcal{R}$.

- arguments analog for the other direction

Mutual simulation ($lts_{MM_1}(P_1 \mid\mid P_2)$ simulates $lts_{MM_2}(P_1' \mid\mid P_2')$):
Let $(s, q) \in \mathcal{R}$ and $s \xrightarrow[mem,mem']{lab} {}_1 s'$, where $s = (mem, ls_1, ls_2)$, with $ls_i = (l_{s,i}, sb_{s,i}, reg_{s,i})$ and $q = (mem, lq_1, lq_2)$ with $lq_i = (l_{q,i}, sb_{s,i}, reg_{q,i})$.
By semantics definition of parallel composition two cases are possible: either (1) $s' = (mem', ls_1', ls_2)$ or (2) $s' = (mem', ls_1, ls_2')$.
In the following, we prove case (1). Note that (2) is analog to (1).
By assumption $(ls_1, lq_1) \in \mathcal{R}_1$, there exists a lq_1', s.t., $lq_1 \xrightarrow[mem_q,mem_q']{lab} {}_2 lq_1'$ and $(ls_1', lq_1') \in \mathcal{R}_1$.

Then, by semantics definition of parallel composition, $lts_{MM_2}(P_1' \mid\mid P_2')$ contains a transition $q \xrightarrow[mem,mem_q']{lab} q'$ with $q' = (mem_q', lq_1', lq_2)$.

All we have to show now is that $mem_q' = mem'$. However, that follows from our previous proof of local bisimilarity, where we had to derive identical labels for both transition systems. In the previous proof, we had to choose $mem = mem_q$ and obtained $mem_q' = mem'$ in all cases.
Hence, we conclude $(s', q') \in \mathcal{R}$.

Mutual simulation ($lts_{MM_2}(P_1' \mid\mid P_2')$ **simulates** $lts_{MM_1}(P_1 \mid\mid P_2)$ **):**
analog to the other direction

□

Please note that the open semantics of local transition system considers all possible values of mem. In our proof of local bisimulation, we had to find one transition in the other LTS (and with it a mem_q value) for all possible transitions of the first LTS starting at all possible values of mem. This had to be done in both directions. Our choice was $mem = mem_q$ in all cases and it always lead to $mem' = mem_q'$. In the closed semantics, the local states have to agree on the same value of mem before a transition and consequently it must hold after it as we have shown in the previous proof.

B Code Examples

```
1  @bot = common global i32* null, align 4
2  @deq = common global i32* null, align 4
3  @age = common global i32* null, align 4
4
5  define void @pushBottom(i32 %elem) nounwind optsize {
6  entry:
7    %0 = load i32** @bot, align 4, !tbaa !0
8    %1 = load i32* %0, align 4, !tbaa !3
9    %2 = load i32** @deq, align 4, !tbaa !0
10   %idx = getelementptr inbounds i32* %2, i32 %1
11   store i32 %elem, i32* %idx, align 4, !tbaa !3
12   %inc = add i32 %1, 1
13   <<< fence required for PSO >>>
14   store i32 %inc, i32* %0, align 4, !tbaa !3
15   ret void
16 }
17 define i32 @popTop() nounwind optsize {
18 entry:
19   %0 = load i32** @age, align 4, !tbaa !0
20   %1 = load i32* %0, align 4, !tbaa !3
21   %2 = load i32** @bot, align 4, !tbaa !0
22   %3 = load i32* %2, align 4, !tbaa !3
23   %shr = ashr i32 %1, 16
24   %cmp = icmp ugt i32 %3, %shr
25   br i1 %cmp, label %if.end, label %return
26
27 if.end:
28   %4 = load i32** @deq, align 4, !tbaa !0
29   %idx = getelementptr inbounds i32* %4, i32 %shr
30   %5 = load i32* %idx, align 4, !tbaa !3
31   %add5 = add i32 %1, 65536
32   %6 = cmpxchg i32* %0, i32 %1, i32 %add5 seq_cst
33   %7 = icmp eq i32 %6, %1
34   %. = select i1 %7, i32 %5, i32 -2
35   br label %return
36
37 return:
38   %retval.0 = phi i32 [ -1, %entry ], [ %., %if.end ]
39   ret i32 %retval.0
40 }
```

Figure B.1: LLVM IR code after compilation of the code in Figure 4.2. Shows variable definition and the methods *pushBottom* and *popTop*.

```
41  define i32 @popBottom() nounwind {
42  entry:
43    %0 = load i32** @bot
44    %1 = load i32* %0
45    %cmp = icmp eq i32 %1, 0
46    br i1 %cmp, label %return, label %if.end
47
48  if.end:
49    %dec = add i32 %1, -1
50    store i32 %dec, i32* %0
51    %2 = load i32** @deq
52    %idx = getelementptr inbounds i32* %2, i32 %dec
53    %3 = load i32* %idx
54    %4 = load i32** @age
55   <<< fence required for TSO >>>
56    %5 = load i32* %4
57    %shr = ashr i32 %5, 16
58    %cmp1 = icmp ugt i32 %dec, %shr
59    br i1 %cmp1, label %return, label %if.end3
60
61  if.end:
62    %dec = add i32 %1, -1
63    store i32 %dec, i32* %0, align 4, !tbaa !3
64    %2 = load i32** @deq, align 4, !tbaa !0
65    %idx = getelementptr inbounds i32* %2, i32 %dec
66    %3 = load i32* %idx, align 4, !tbaa !3
67    %4 = load i32** @age, align 4, !tbaa !0
68    %5 = load i32* %4, align 4, !tbaa !3
69    %shr = ashr i32 %5, 16
70    %cmp1 = icmp ugt i32 %dec, %shr
71    br i1 %cmp1, label %return, label %if.end3
73  if.end3:
74    store i32 0, i32* %0, align 4, !tbaa !3
75    %and = and i32 %5, 65535
76    %add = add nsw i32 %and, 1
77    %cmp5 = icmp eq i32 %dec, %shr
78    br i1 %cmp5, label %if.then6, label %if.end9
79
80  if.then6:
81    %6 = cmpxchg i32* %4, i32 %5, i32 %add seq_cst
82    %7 = icmp eq i32 %6, %5
83    br i1 %7, label %return, label %if.then7
84
85  if.then7:
86    %.pre = load i32** @age, align 4, !tbaa !0
87    br label %if.end9
88
89  if.end9:
90    %8 = phi i32* [ %.pre, %if.then7 ], [ %4, %if.end3 ]
91    store i32 %add, i32* %8, align 4, !tbaa !3
92    br label %return
93
94  return:
95    %retval.0 = phi i32 [ -1, %if.end9 ],
96          [ -1, %entry ], [ %3, %if.end ], [ %3, %if.then6 ]
97    ret i32 %retval.0
98  }
```

Figure B.2: LLVM IR code after compilation of the code in Figure 4.2. Shows the method *popBottom*.

```
1  #define cas(_ptr, _old, _new)
   __sync_bool_compare_and_swap(_ptr, _old, _new)
2  #define ABORT -1
3  #define OK 1
4
5  int volatile *glb = 0, *x = 0, *y = 0;
6  int lx1= 0,ly1=0,lx2= 0,ly2=0;
7
8  int proc13() {
9    int loc, tmp;
10   //TMBegin
11   do {
12     loc = *glb;
13   } while (loc & 1);
14   //-----------------------------
15   //TMWrite x := 1;
16   if (!(loc & 1)) {
17     if (!cas(glb, loc, loc + 1)) {
18       return ABORT;
19     }
20     loc++;
21   }
22   *x = 1;
23
24   //-----------------------------
25   //TMend
26   if (loc & 1) {
27     __sync_synchronize ();
28     (*glb)++;
29   }
30   return OK;
31 }
```

```
32 int proc33() {
33   int loc, tmp;
34   //TMBegin
35   do {
36     loc = *glb;
37   } while (loc & 1);
38   //-----------------------------
39   //TMRead lx := x;
40   tmp = *x;
41   if (*glb != loc) {
42     return ABORT;
43   }
44   lx1 = tmp;
45
46   //TMRead ly := y;
47   tmp = *y;
48   if (*glb != loc) {
49     return ABORT;
50   }
51   ly1 = tmp;
52
53
54   //-----------------------------
55   //TMend
56   if (loc & 1) {
57     __sync_synchronize ();
58     (*glb)++;
59   }
60   return OK;
61 }
```

Figure B.3: C code of two transactions from the transactional memory implementation, TML by [DDS+10]. Method *proc13* implements *begin, write(x,1) ,commit*; Method *proc33* implements *begin, read(x,lx), read(y,ly), commit.* The implementation also shows the required fence (*sync_synchronize*). Please note that we combined begin, read, write and commit operations into one operation, in order to be able to take the reordering across method boundaries into account.

Bibliography

[AAA+15] Parosh Aziz Abdulla, Stavros Aronis, Mohamed Faouzi Atig, Bengt Jonsson, Carl Leonardsson, and Konstantinos F. Sagonas. Stateless model checking for TSO and PSO. In Christel Baier and Cesare Tinelli, editors, *TACAS 2015*, volume 9035 of *LNCS*, pages 353–367. Springer, 2015.

[AAC+12] Parosh Aziz Abdulla, Mohamed Faouzi Atig, Yu-Fang Chen, Carl Leonardsson, and Ahmed Rezine. Counter-example guided fence insertion under TSO. In *Proceedings of the 18th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, TACAS'12, pages 204–219, Berlin, Heidelberg, 2012. Springer-Verlag.

[AAC+13] Parosh Aziz Abdulla, Mohamed Faouzi Atig, Yu-Fang Chen, Carl Leonardsson, and Ahmed Rezine. Memorax, a precise and sound tool for automatic fence insertion under TSO. In Nir Piterman and Scott A. Smolka, editors, *Tools and Algorithms for the Construction and Analysis of Systems - 19th International Conference, TACAS 2013, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2013, Rome, Italy, March 16-24, 2013. Proceedings*, volume 7795 of *Lecture Notes in Computer Science*, pages 530–536. Springer, 2013.

[ABBM10] Mohamed Faouzi Atig, Ahmed Bouajjani, Sebastian Burckhardt, and Madanlal Musuvathi. On the verification problem for weak memory models. In *Proceedings of the 37th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '10, pages 7–18, New York, NY, USA, 2010. ACM.

[ABP98] Nimar S. Arora, Robert D. Blumofe, and C. Greg Plaxton. Thread scheduling for multiprogrammed multiprocessors. In *Proceedings of the Tenth Annual ACM Symposium on Parallel Algorithms and Architectures*, SPAA '98, pages 119–129, New York, NY, USA, 1998. ACM.

[ABP11] Mohamed Faouzi Atig, Ahmed Bouajjani, and Gennaro Parlato. Getting rid of store-buffers in TSO analysis. In Ganesh Gopalakrishnan and Shaz Qadeer, editors, *CAV*, volume 6806 of *Lecture Notes in Computer Science*, pages 99–115. Springer, 2011.

[AFI+08] Jade Alglave, Anthony Fox, Samin Ishtiaq, Magnus O. Myreen, Susmit Sarkar, Peter Sewell, and Francesco Zappa Nardelli. The semantics of Power and ARM multiprocessor machine code. In *Proceedings of the 4th workshop on Declarative aspects of multicore programming*, DAMP '09, pages 13–24, New York, NY, USA, 2008. ACM.

[AG96] S. V. Adve and K. Gharachorloo. Shared memory consistency models: A tutorial. *IEEE Computer*, 29(12):66–76, 1996.

[AGHR14] Hagit Attiya, Alexey Gotsman, Sandeep Hans, and Noam Rinetzky. Safety of live transactions in transactional memory: TMS is necessary and sufficient. In Fabian Kuhn, editor, *Distributed Computing - 28th International Symposium, DISC 2014, Austin, TX, USA, October 12-15, 2014. Proceedings*, volume 8784 of *Lecture Notes in Computer Science*, pages 376–390. Springer, 2014.

[AH90] Sarita V. Adve and Mark D. Hill. Weak ordering - A new definition. In Jean-Loup Baer, Larry Snyder, and James R. Goodman, editors, *Proceedings of the 17th Annual International Symposium on Computer Architecture. Seattle, WA, June 1990*, pages 2–14. ACM, 1990.

[AH93] 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.

[AKNT13] Jade Alglave, Daniel Kroening, Vincent Nimal, and Michael Tautschnig. Software verification for weak memory via program transformation. In *In Procedings of ESOP 2013*, volume 7792 of *LNCS*, pages 512–532. Springer, 2013.

[AKY10] Yehuda Afek, Guy Korland, and Eitan Yanovsky. Quasi-linearizability: Relaxed consistency for improved concurrency. In Chenyang Lu, Toshimitsu Masuzawa, and Mohamed Mosbah, editors, *Principles of*

Distributed Systems - 14th International Conference, OPODIS 2010, Tozeur, Tunisia, December 14-17, 2010. Proceedings, volume 6490 of *Lecture Notes in Computer Science*, pages 395–410. Springer, 2010.

[Alg12] J. Alglave. A formal hierarchy of weak memory models. *Formal Methods in System Design*, 41(2):178–210, 2012.

[AM06] Arvind Arvind and Jan-Willem Maessen. Memory model = instruction reordering + store atomicity. *SIGARCH Comput. Archit. News*, 34(2):29–40, May 2006.

[AM14] Tatsuya Abe and Toshiyuki Maeda. A general model checking framework for various memory consistency models. In *2014 IEEE International Parallel & Distributed Processing Symposium Workshops, Phoenix, AZ, USA, May 19-23, 2014*, pages 332–341. IEEE Computer Society, 2014.

[AMSS10] Jade Alglave, Luc Maranget, Susmit Sarkar, and Peter Sewell. Fences in weak memory models. In Touili et al. [TCJ10], pages 258–272.

[AMSS12] Jade Alglave, Luc Maranget, Susmit Sarkar, and Peter Sewell. Fences in weak memory models (extended version). *Formal Methods in System Design*, 40(2):170–205, 2012.

[ARM13] ARM. *ARM Architecture Reference Manual - ARMv8, for ARMv8 - A architecture profile*, April 2013.

[ARR+07] Daphna Amit, Noam Rinetzky, Thomas W. Reps, Mooly Sagiv, and Eran Yahav. Comparison under abstraction for verifying linearizability. In Werner Damm and Holger Hermanns, editors, *Computer Aided Verification, 19th International Conference, CAV 2007, Berlin, Germany, July 3-7, 2007, Proceedings*, volume 4590 of *Lecture Notes in Computer Science*, pages 477–490. Springer, 2007.

[AS14] Elvira Albert and Emil Sekerinski, editors. *Integrated Formal Methods - 11th International Conference, IFM 2014, Bertinoro, Italy, September 9-11, 2014, Proceedings*, volume 8739 of *Lecture Notes in Computer Science*. Springer, 2014.

[AUMM16] Tatsuya Abe, Tomoharu Ugawa, Toshiyuki Maeda, and Kousuke Matsumoto. Reducing state explosion for software model checking with relaxed memory consistency models. In Martin Fränzle, Deepak Kapur, and Naijun Zhan, editors, *Dependable Software Engineering: Theories, Tools, and Applications - Second International Symposium, SETTA 2016,*

Beijing, China, November 9-11, 2016, Proceedings, volume 9984 of *Lecture Notes in Computer Science*, pages 118–135, 2016.

[BAM06] Sebastian Burckhardt, Rajeev Alur, and Milo M. K. Martin. Bounded model checking of concurrent data types on relaxed memory models: A case study. In T. Ball and R. B. Jones, editors, *Computer Aided Verification, 18th International Conference, CAV 2006, Seattle, WA, USA, August 17-20, 2006, Proceedings*, volume 4144 of *LNCS*, pages 489–502. Springer, 2006.

[BAM07] Sebastian Burckhardt, Rajeev Alur, and Milo M. K. Martin. CheckFence: checking consistency of concurrent data types on relaxed memory models. In *PLDI*, pages 12–21, 2007.

[BBG+95] Hal Berenson, Philip A. Bernstein, Jim Gray, Jim Melton, Elizabeth J. O'Neil, and Patrick E. O'Neil. A critique of ANSI SQL isolation levels. In Michael J. Carey and Donovan A. Schneider, editors, *Proceedings of the 1995 ACM SIGMOD International Conference on Management of Data, San Jose, California, May 22-25, 1995.*, pages 1–10. ACM Press, 1995.

[BC04] Yves Bertot and Pierre Castéran. *Interactive Theorem Proving and Program Development - Coq'Art: The Calculus of Inductive Constructions*. Texts in Theoretical Computer Science. An EATCS Series. Springer, 2004.

[BC05] Daniel Bovet and Marco Cesati. *Understanding The Linux Kernel*. Oreilly & Associates Inc, 2005.

[BCCZ99] Armin Biere, Alessandro Cimatti, Edmund M. Clarke, and Yunshan Zhu. Symbolic model checking without BDDs. In Rance Cleaveland, editor, *Tools and Algorithms for Construction and Analysis of Systems, 5th International Conference, TACAS '99, Held as Part of the European Joint Conferences on the Theory and Practice of Software, ETAPS'99, Amsterdam, The Netherlands, March 22-28, 1999, Proceedings*, volume 1579 of *Lecture Notes in Computer Science*, pages 193–207. Springer, 1999.

[BCDM15] Ahmed Bouajjani, Georgel Calin, Egor Derevenetc, and Roland Meyer. Lazy TSO reachability. In Alexander Egyed and Ina Schaefer, editors, *FASE 2015*, volume 9033 of *LNCS*, pages 267–282. Springer, 2015.

[BCM+92] Jerry R. Burch, Edmund M. Clarke, Kenneth L. McMillan, David L. Dill, and L. J. Hwang. Symbolic model checking: 10^20 states and beyond. *Inf. Comput.*, 98(2):142–170, 1992.

[BDG13] Mark Batty, Mike Dodds, and Alexey Gotsman. Library abstraction for C/C++ concurrency. In Giacobazzi and Cousot [GC13], pages 235–248.

[BDM13] Ahmed Bouajjani, Egor Derevenetc, and Roland Meyer. Checking and Enforcing Robustness against TSO. In *In Procedings of ESOP 2013*, volume 7792 of *LNCS*, pages 533–553. Springer, 2013.

[BDMT10] Sebastian Burckhardt, Chris Dern, Madanlal Musuvathi, and Roy Tan. Line-up: a complete and automatic linearizability checker. In Benjamin G. Zorn and Alexander Aiken, editors, *Proceedings of the 2010 ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2010, Toronto, Ontario, Canada, June 5-10, 2010*, pages 330–340. ACM, 2010.

[BGMY12] S. Burckhardt, A. Gotsman, M. Musuvathi, and H. Yang. Concurrent library correctness on the TSO memory model. In Helmut Seidl, editor, *ESOP*, volume 7211 of *Lecture Notes in Computer Science*, pages 87–107. Springer, 2012.

[BL80] James Burns and Nancy A. Lynch. Mutual exclusion using indivisible reads and writes. In *In Proceedings of the 18th Annual Allerton Conference on Communication, Control, and Computing*, pages 833–842, 1980.

[BMM11] Ahmed Bouajjani, Roland Meyer, and Eike Möhlmann. Deciding robustness against total store ordering. In Luca Aceto, Monika Henzinger, and Jiri Sgall, editors, *Automata, Languages and Programming - 38th International Colloquium, ICALP 2011, Zurich, Switzerland, July 4-8, 2011, Proceedings, Part II*, volume 6756 of *Lecture Notes in Computer Science*, pages 428–440. Springer, 2011.

[BP09] Gérard Boudol and Gustavo Petri. Relaxed memory models: an operational approach. In Zhong Shao and Benjamin C. Pierce, editors, *Proceedings of the 36th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2009, Savannah, GA, USA, January 21-23, 2009*, pages 392–403. ACM, 2009.

[BSS11a] Jacob Burnim, Koushik Sen, and Christos Stergiou. Sound and Complete Monitoring of Sequential Consistency for Relaxed Memory Models. In Parosh Aziz Abdulla and K. Rustan M. Leino, editors, *TACAS*, volume 6605 of *Lecture Notes in Computer Science*, pages 11–25. Springer, 2011.

[BSS11b] Jacob Burnim, Koushik Sen, and Christos Stergiou. Testing concurrent programs on relaxed memory models. In Matthew B. Dwyer and Frank Tip, editors, *ISSTA*, pages 122–132. ACM, 2011.

[BSTR11] Simon Bäumler, Gerhard Schellhorn, Bogdan Tofan, and Wolfgang Reif. Proving linearizability with temporal logic. *Formal Asp. Comput.*, 23(1):91–112, 2011.

[CBM] CBMC bounded model checker. `http://www.cprover.org/cbmc/`. Accessed: 2016-06-06.

[CBM10] Katherine E. Coons, Sebastian Burckhardt, and Madanlal Musuvathi. GAMBIT: effective unit testing for concurrency libraries. In R. Govindarajan, David A. Padua, and Mary W. Hall, editors, *Proceedings of the 15th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPOPP 2010, Bangalore, India, January 9-14, 2010*, pages 15–24. ACM, 2010.

[CCG+02] Alessandro Cimatti, Edmund M. Clarke, Enrico Giunchiglia, Fausto Giunchiglia, Marco Pistore, Marco Roveri, Roberto Sebastiani, and Armando Tacchella. NuSMV 2: An opensource tool for symbolic model checking. In Ed Brinksma and Kim Guldstrand Larsen, editors, *Computer Aided Verification, 14th International Conference, CAV 2002,Copenhagen, Denmark, July 27-31, 2002, Proceedings*, volume 2404 of *Lecture Notes in Computer Science*, pages 359–364. Springer, 2002.

[CDG05] R. Colvin, S. Doherty, and L. Groves. Verifying concurrent data structures by simulation. *Electr. Notes Theor. Comput. Sci.*, 137(2):93–110, 2005.

[CFR+91] Ron Cytron, Jeanne Ferrante, Barry K. Rosen, Mark N. Wegman, and F. Kenneth Zadeck. Efficiently computing static single assignment form and the control dependence graph. *ACM Trans. Program. Lang. Syst.*, 13:451–490, 1991.

[CGJ+00] Edmund M. Clarke, Orna Grumberg, Somesh Jha, Yuan Lu, and Helmut Veith. Counterexample-guided abstraction refinement. In E. Allen Emerson and A. Prasad Sistla, editors, *Computer Aided Verification, 12th International Conference, CAV 2000, Chicago, IL, USA, July 15-19, 2000, Proceedings*, volume 1855 of *Lecture Notes in Computer Science*, pages 154–169. Springer, 2000.

[CGP01] Edmund M. Clarke, Orna Grumberg, and Doron A. Peled. *Model checking*. MIT Press, 2001.

[CKRW99] P. Cenciarelli, A. Knapp, B. Reus, and M. Wirsing. An event-based structural operational semantics of multi-threaded Java. In J. Alves-Foss, editor, *Formal Syntax and Semantics of Java*, volume 1523 of *LNCS*, pages 157–200. Springer-Verlag, Jan 1999.

[Cla08] Edmund M. Clarke. *The Birth of Model Checking*, pages 1–26. Springer Berlin Heidelberg, Berlin, Heidelberg, 2008.

[CNW13] Dave Clarke, James Noble, and Tobias Wrigstad, editors. *Aliasing in Object-Oriented Programming. Types, Analysis and Verification*, volume 7850 of *Lecture Notes in Computer Science*. Springer, 2013.

[CRZ+10] Pavol Cerný, Arjun Radhakrishna, Damien Zufferey, Swarat Chaudhuri, and Rajeev Alur. Model checking of linearizability of concurrent list implementations. In Touili et al. [TCJ10], pages 465–479.

[CS10] E. Cohen and B. Schirmer. From total store order to sequential consistency: A practical reduction theorem. In M. Kaufmann and L. C. Paulson, editors, *ITP*, volume 6172 of *Lecture Notes in Computer Science*, pages 403–418. Springer, 2010.

[DB14] John Derrick and Eerke A. Boiten. *Refinement in Z and Object-Z - Foundations and Advanced Applications (2. ed.)*. Springer, 2014.

[DD15] Brijesh Dongol and John Derrick. Verifying linearisability: A comparative survey. *ACM Comput. Surv.*, 48(2):19:1–19:43, 2015.

[DDS+10] L. Dalessandro, D. Dice, M. L. Scott, N. Shavit, and M. F. Spear. Transactional mutex locks. In Pasqua D'Ambra, Mario Rosario Guarracino, and Domenico Talia, editors, *Euro-Par (2)*, volume 6272 of *Lecture Notes in Computer Science*, pages 2–13. Springer, 2010.

[DDS+14] John Derrick, Brijesh Dongol, Gerhard Schellhorn, Bogdan Tofan, Oleg Travkin, and Heike Wehrheim. Quiescent consistency: Defining and verifying relaxed linearizability. In Cliff B. Jones, Pekka Pihlajasaari, and Jun Sun, editors, *FM 2014: Formal Methods - 19th International Symposium, Singapore, May 12-16, 2014. Proceedings*, volume 8442 of *Lecture Notes in Computer Science*, pages 200–214. Springer, 2014.

[Der15] Egor Derevenetc. *Robustness against Relaxed Memory Models*. PhD thesis, University of Kaiserslautern, 2015.

[DGLM13] Simon Doherty, Lindsay Groves, Victor Luchangco, and Mark Moir. Towards formally specifying and verifying transactional memory. *Formal Asp. Comput.*, 25(5):769–799, 2013.

[Dij68] E. W. Dijkstra. Cooperating sequential processes. In *Programming Languages*, pages 43–112. Academic Press, 1968.

[dMB08] Leonardo Mendonça de Moura and Nikolaj Bjørner. Z3: an efficient SMT solver. In C. R. Ramakrishnan and Jakob Rehof, editors, *Tools and Algorithms for the Construction and Analysis of Systems, 14th International Conference, TACAS 2008, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2008, Budapest, Hungary, March 29-April 6, 2008. Proceedings*, volume 4963 of *Lecture Notes in Computer Science*, pages 337–340. Springer, 2008.

[DMVY13] Andrei Marian Dan, Yuri Meshman, Martin T. Vechev, and Eran Yahav. Predicate abstraction for relaxed memory models. In Francesco Logozzo and Manuel Fähndrich, editors, *SAS*, volume 7935 of *Lecture Notes in Computer Science*, pages 84–104. Springer, 2013.

[Dri14] Katharina Dridger. Integration von History-basierten Korrektheits-Checks im Model Checker SPIN. Bachelor Thesis, 5 2014. University Paderborn.

[DSD14] John Derrick, Graeme Smith, and Brijesh Dongol. Verifying linearizability on TSO architectures. In Albert and Sekerinski [AS14], pages 341–356.

[DSGD17] John Derrick, Graeme Smith, Lindsay Groves, and Brijesh Dongol. *A Proof Method for Linearizability on TSO Architectures*, pages 61–91. Springer International Publishing, Cham, 2017.

[DSW07] J. Derrick, G. Schellhorn, and H. Wehrheim. Proving linearizability via non-atomic refinement. In J. Davies and J. Gibbons, editors, *iFM*, volume 4591 of *LNCS*, pages 195–214. Springer, 2007.

[DSW11a] J. Derrick, G. Schellhorn, and H. Wehrheim. Mechanically verified proof obligations for linearizability. *ACM Trans. Program. Lang. Syst.*, 33(1):4, 2011.

[DSW11b] J. Derrick, G. Schellhorn, and H. Wehrheim. Verifying linearisability with potential linearisation points. In M. Butler and W. Schulte, editors, *FM*, volume 6664 of *LNCS*, pages 323–337. Springer, 2011.

[DTDW13] Brijesh Dongol, Oleg Travkin, John Derrick, and Heike Wehrheim. A high-level semantics for program execution under total store order memory. In Zhiming Liu, Jim Woodcock, and Huibiao Zhu, editors, *Theoretical Aspects of Computing - ICTAC 2013 - 10th International Colloquium, Shanghai, China, September 4-6, 2013. Proceedings*, volume 8049 of *Lecture Notes in Computer Science*, pages 177–194. Springer, 2013.

[DW05] John Derrick and Heike Wehrheim. Non-atomic refinement in Z and CSP. In Helen Treharne, Steve King, Martin C. Henson, and Steve A. Schneider, editors, *ZB 2005: Formal Specification and Development in Z and B, 4th International Conference of B and Z Users, Guildford, UK, April 13-15, 2005, Proceedings*, volume 3455 of *Lecture Notes in Computer Science*, pages 24–44. Springer, 2005.

[EPS+14] G. Ernst, J. Pfähler, G. Schellhorn, D. Haneberg, and W. Reif. KIV - overview and VerifyThis competition. *Software Tools for Techn. Transfer*, pages 1–18, 2014.

[EQS+10] T. Elmas, S. Qadeer, A. Sezgin, O. Subasi, and S. Tasiran. Simplifying linearizability proofs with reduction and abstraction. In J. Esparza and R. Majumdar, editors, *TACAS*, volume 6015 of *LNCS*, pages 296–311. Springer, 2010.

[Fla04] Cormac Flanagan. Verifying commit-atomicity using model-checking. In *In Proc. 11th Int'l. SPIN Workshop on Model Checking of Software, volume 2989 of LNCS*, pages 252–266. Springer-Verlag, 2004.

[GC13] Roberto Giacobazzi and Radhia Cousot, editors. *The 40th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '13, Rome, Italy - January 23 - 25, 2013*. ACM, 2013.

[GH04] Hui Gao and Wim H. Hesselink. A formal reduction for lock-free parallel algorithms. In Rajeev Alur and Doron A. Peled, editors, *Computer Aided Verification, 16th International Conference, CAV 2004, Boston, MA, USA, July 13-17, 2004, Proceedings*, volume 3114 of *Lecture Notes in Computer Science*, pages 44–56. Springer, 2004.

[GK08] R. Guerraoui and M. Kapalka. On the correctness of transactional memory. In Siddhartha Chatterjee and Michael L. Scott, editors, *PPOPP*, pages 175–184. ACM, 2008.

[GMY12] Alexey Gotsman, Madanlal Musuvathi, and Hongseok Yang. Show no weakness: sequentially consistent specifications of TSO libraries. In

Proceedings of the 26th international conference on Distributed Computing, DISC'12, pages 31–45, Berlin, Heidelberg, 2012. Springer-Verlag.

[God96] Patrice Godefroid. *Partial-Order Methods for the Verification of Concurrent Systems - An Approach to the State-Explosion Problem*, volume 1032 of *Lecture Notes in Computer Science*. Springer, 1996.

[Hes07] Wim H. Hesselink. A criterion for atomicity revisited. *Acta Informatica*, 44(2):123–151, 2007.

[HKP+13] Thomas A. Henzinger, Christoph M. Kirsch, Hannes Payer, Ali Sezgin, and Ana Sokolova. Quantitative relaxation of concurrent data structures. In Giacobazzi and Cousot [GC13], pages 317–328.

[HKV98] Lisa Higham, Jalal Kawash, and Nathaly Verwaal. Weak memory consistency models. part i: Definitions and comparisons. Technical report, Department of Computer Science, The University of Calgary, 1998.

[HKV02] David Harel, Orna Kupferman, and Moshe Y. Vardi. On the complexity of verifying concurrent transition systems. *Inf. Comput.*, 173(2):143–161, 2002.

[HM93] M. Herlihy and J. E. B. Moss. Transactional memory: Architectural support for lock-free data structures. In Alan Jay Smith, editor, *ISCA*, pages 289–300. ACM, 1993.

[Hol03] Gerard Holzmann. *The Spin model checker: Primer and Reference Manual*. Addison-Wesley Professional, first edition, 2003.

[HS08] M. Herlihy and N. Shavit. *The art of multiprocessor programming*. Morgan Kaufmann, 2008.

[HSV13] Thomas A. Henzinger, Ali Sezgin, and Viktor Vafeiadis. Aspect-oriented linearizability proofs. In Pedro R. D'Argenio and Hernán C. Melgratti, editors, *CONCUR 2013 - Concurrency Theory - 24th International Conference, CONCUR 2013, Buenos Aires, Argentina, August 27-30, 2013. Proceedings*, volume 8052 of *Lecture Notes in Computer Science*, pages 242–256. Springer, 2013.

[HW90] M. P. Herlihy and J. M. Wing. Linearizability: a correctness condition for concurrent objects. *ACM Trans. Program. Lang. Syst.*, 12(3):463–492, 1990.

[IBM15] IBM. *Power ISA Version 2.07 B*, April 2015.

[Int] Intel. A formal specification of Intel Itanium processor family memory ordering. http://www.intel.com/design/itanium/downloads/251429.htm. Accessed: 05 July 2016.

[Int12] Intel, Santa Clara, CA, USA. *Intel 64 and IA-32 Architectures Software Developer's Manual Volume 3A: System Programming Guide, Part 1*, May 2012.

[ISO11a] ISO/IEC. Programming languages - c++, 14882:2011. Technical report, ISO, 2011.

[ISO11b] ISO/IEC. Programming languages - c, 9899:2011. Technical report, ISO, 2011.

[Jon83] Cliff B. Jones. Specification and design of (parallel) programs. In *IFIP Congress*, pages 321–332, 1983.

[Jon12] Bengt Jonsson. Using refinement calculus techniques to prove linearizability. *Formal Asp. Comput.*, 24(4-6):537–554, 2012.

[KVY12] Michael Kuperstein, Martin T. Vechev, and Eran Yahav. Automatic inference of memory fences. *SIGACT News*, 43(2):108–123, 2012.

[LA04] Chris Lattner and Vikram Adve. LLVM: A compilation framework for lifelong program analysis & transformation. In *Proceedings of the International Symposium on Code Generation and Optimization: Feedback-directed and Runtime Optimization*, CGO '04, pages 75–, Washington, DC, USA, 2004. IEEE Computer Society.

[Lam74] L. Lamport. A new solution of Dijkstra's concurrent programming problem. *Commun. ACM*, 17(8):453–455, 1974.

[Lam79] L. Lamport. How to make a multiprocessor computer that correctly executes multiprocess programs. *IEEE Trans. Computers*, 28(9):690–691, 1979.

[LCLS09] Yang Liu, Wei Chen, YanhongA. Liu, and Jun Sun. Model checking linearizability via refinement. In Ana Cavalcanti and DennisR. Dams, editors, *FM 2009: Formal Methods*, volume 5850 of *Lecture Notes in Computer Science*, pages 321–337. Springer Berlin Heidelberg, 2009.

[LNP+12] Feng Liu, Nayden Nedev, Nedyalko Prisadnikov, Martin Vechev, and Eran Yahav. Dynamic synthesis for relaxed memory models. *SIGPLAN Not.*, 47(6):429–440, June 2012.

[LT89] N. Lynch and M. Tuttle. An introduction to input/output automata. *CWI-Quarterly*, 2(3):219–246, 1989.

[LV95] N. Lynch and F. Vaandrager. Forward and backward simulations I: Untimed systems. *Inf. Comput.*, 121(2):214–233, 1995.

[LW10] Alexander Linden and Pierre Wolper. An automata-based symbolic approach for verifying programs on relaxed memory models. In *Proceedings of the 17th International SPIN Conference on Model Checking Software*, SPIN'10, pages 212–226, Berlin, Heidelberg, 2010. Springer-Verlag.

[Mil80] Robin Milner. *A Calculus of Communicating Systems*. Springer, 1980.

[MMS+12] Sela Mador-Haim, Luc Maranget, Susmit Sarkar, Kayvan Memarian, Jade Alglave, Scott Owens, Rajeev Alur, Milo M. K. Martin, Peter Sewell, and Derek Williams. An axiomatic memory model for POWER multiprocessors. In Madhusudan and Seshia [MS12], pages 495–512.

[MPA05] Jeremy Manson, William Pugh, and Sarita V. Adve. The Java memory model. In Jens Palsberg and Martín Abadi, editors, *Proceedings of the 32nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2005, Long Beach, California, USA, January 12-14, 2005*, pages 378–391. ACM, 2005.

[MQ06] Madan Musuvathi and Shaz Qadeer. CHESS: systematic stress testing of concurrent software. In Germán Puebla, editor, *Logic-Based Program Synthesis and Transformation, 16th International Symposium, LOPSTR 2006, Venice, Italy, July 12-14, 2006, Revised Selected Papers*, volume 4407 of *Lecture Notes in Computer Science*, pages 15–16. Springer, 2006.

[MS96] Maged M. Michael and Michael L. Scott. Simple, fast, and practical non-blocking and blocking concurrent queue algorithms. In James E. Burns and Yoram Moses, editors, *Proceedings of the Fifteenth Annual ACM Symposium on Principles of Distributed Computing, Philadelphia, Pennsylvania, USA, May 23-26, 1996*, pages 267–275. ACM, 1996.

[MS12] P. Madhusudan and S. A. Seshia, editors. *Computer Aided Verification - 24th International Conference, CAV 2012, Berkeley, CA, USA, July 7-13, 2012 Proceedings*, volume 7358 of *LNCS*. Springer, 2012.

[MSS12] Luc Maranget, Susmit Sarkar, and Peter Sewell. A tutorial introduction to the ARM and POWER relaxed memory models - revision 120.

http://www.cl.cam.ac.uk/ pes20/weakmemory/, October 2012. Accessed: 25 July 2016.

[NPW02] Tobias Nipkow, Lawrence C. Paulson, and Markus Wenzel. *Isabelle/HOL - A Proof Assistant for Higher-Order Logic*, volume 2283 of *Lecture Notes in Computer Science*. Springer, 2002.

[Obj08] Object Management Group (OMG). MOF model to text transformation language, v1.0. OMG Document Number formal/2008-01-16 (`http://www.omg.org/spec/MOFM2T/1.0/`), 2008.

[Obj15a] Object Management Group (OMG). Meta object facility (MOF) core, v2.5. OMG Document Number formal/2015-06-06 (`http://www.omg.org/spec/MOF/2.5/`), 2015.

[Obj15b] Object Management Group (OMG). OMG unified modeling language (OMG UML), infrastructure, v2.5. OMG Document Number formal/2015-03-01 (`http://www.omg.org/spec/UML/2.5/`), 2015.

[Owe10] Scott Owens. Reasoning about the implementation of concurrency abstractions on x86-TSO. In Theo D'Hondt, editor, *ECOOP*, volume 6183 of *Lecture Notes in Computer Science*, pages 478–503. Springer, 2010.

[Pap79] Christos H. Papadimitriou. The serializability of concurrent database updates. *J. ACM*, 26(4):631–653, October 1979.

[PD95] Seungjoon Park and David L. Dill. An executable specification, analyzer and verifier for RMO (relaxed memory order). In *SPAA*, pages 34–41, 1995.

[Pet81] G.L. Peterson. Myths about the mutual exclusion problem. *Information Processing Letters*, 12:115–116, 1981.

[Pos46] Emil Leon Post. A variant of a recursively unsolvable problem. *Bulletin of the American Mathematical Society*, 52:264–268, 1946.

[PP84] Mark S. Papamarcos and Janak H. Patel. A low-overhead coherence solution for multiprocessors with private cache memories. In Dharma P. Agrawal, editor, *Proceedings of the 11th Annual Symposium on Computer Architecture, Ann Arbor, USA, June 1984*, pages 348–354. ACM, 1984.

[Rey02] J. C. Reynolds. Separation logic: A logic for shared mutable data structures. In *LICS*, pages 55–74. IEEE Computer Society, 2002.

[SA08] Jaroslav Sevcík and David Aspinall. On validity of program transformations in the Java memory model. In Jan Vitek, editor, *ECOOP 2008 - Object-Oriented Programming, 22nd European Conference, Paphos, Cyprus, July 7-11, 2008, Proceedings*, volume 5142 of *Lecture Notes in Computer Science*, pages 27–51. Springer, 2008.

[Sen15] Koushik Sen. Concolic testing: a decade later (keynote). In Harry Xu and Walter Binder, editors, *Proceedings of the 13th International Workshop on Dynamic Analysis, WODA@SPLASH 2015, Pittsburgh, PA, USA, October 26, 2015*, page 1. ACM, 2015.

[Sev09] Jaroslav Sevcík. *Program transformations in weak memory models*. PhD thesis, University of Edinburgh, UK, 2009.

[SK09] Marc Shapiro and Bettina Kemme. Eventual consistency. In Ling Liu and M. Tamer Özsu, editors, *Encyclopedia of Database Systems*, pages 1071–1072. Springer US, 2009.

[SPA92] SPARC International, Inc., CORPORATE. *The SPARC Architecture Manual: Version 8*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1992.

[SPA94] SPARC International, Inc., CORPORATE. *The SPARC Architecture Manual (Version 9)*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1994.

[SSA+11] Susmit Sarkar, Peter Sewell, Jade Alglave, Luc Maranget, and Derek Williams. Understanding POWER multiprocessors. In Mary W. Hall and David A. Padua, editors, *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2011, San Jose, CA, USA, June 4-8, 2011*, pages 175–186. ACM, 2011.

[SSO+10] 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.

[ST97] Nir Shavit and Dan Touitou. Software transactional memory. *Distributed Computing*, 10(2):99–116, 1997.

[sv-16] SV-Competition Benchmarks, April 2016. https://github.com/dbeyer/sv-benchmarks.

[SWD12] G. Schellhorn, H. Wehrheim, and J. Derrick. How to prove algorithms linearisable. In Madhusudan and Seshia [MS12], pages 243–259.

[Szy88] B. K. Szymanski. A simple solution to Lamport's concurrent programming problem with linear wait. In *Proceedings of the 2Nd International Conference on Supercomputing*, ICS '88, pages 621–626, New York, NY, USA, 1988. ACM.

[TCJ10] Tayssir Touili, Byron Cook, and Paul Jackson, editors. *Computer Aided Verification, 22nd International Conference, CAV 2010, Edinburgh, UK, July 15-19, 2010. Proceedings*, volume 6174 of *Lecture Notes in Computer Science*. Springer, 2010.

[TMW13] Oleg Travkin, Annika Mütze, and Heike Wehrheim. SPIN as a linearizability checker under weak memory models. In Valeria Bertacco and Axel Legay, editors, *Hardware and Software: Verification and Testing - 9th International Haifa Verification Conference, HVC 2013, Haifa, Israel, November 5-7, 2013, Proceedings*, volume 8244 of *Lecture Notes in Computer Science*, pages 311–326. Springer, 2013.

[Tra16] Oleg Travkin. Weak2SC - git repository. https://github.com/oleg82upb, 2016. Accessed: December 2016.

[Tre86] R. K. Treiber. Systems programming: Coping with parallelism. Technical Report RJ 5118, IBM Almaden Res. Ctr., 1986.

[TSR14] Bogdan Tofan, Gerhard Schellhorn, and Wolfgang Reif. A compositional proof method for linearizability applied to a wait-free multiset. In Albert and Sekerinski [AS14], pages 357–372.

[TTSW14] Bogdan Tofan, Oleg Travkin, Gerhard Schellhorn, and Heike Wehrheim. Two approaches for proving linearizability of multiset. *Sci. Comput. Program.*, 96:297–314, 2014.

[TW14] Oleg Travkin and Heike Wehrheim. Handling TSO in mechanized linearizability proofs. In Eran Yahav, editor, *Hardware and Software: Verification and Testing*, volume 8855 of *Lecture Notes in Computer Science*, pages 132–147. Springer International Publishing, 2014.

[TW16] Oleg Travkin and Heike Wehrheim. Verification of concurrent programs on weak memory models. In Augusto Sampaio and Farn Wang, editors, *Theoretical Aspects of Computing - ICTAC 2016 - 13th International Colloquium, Taipei, Taiwan, ROC, October 24-31, 2016, Proceedings*, volume 9965 of *Lecture Notes in Computer Science*, pages 3–24, 2016.

[TWS12] Oleg Travkin, Heike Wehrheim, and Gerhard Schellhorn. Proving linearizability of multiset with local proof obligations. *ECEASST*, 53, 2012.

[Vaf07] V. Vafeiadis. *Modular fine-grained concurrency verification.* PhD thesis, University of Cambridge, 2007.

[Val89] Antti Valmari. Stubborn sets for reduced state space generation. In Grzegorz Rozenberg, editor, *Advances in Petri Nets 1990 [10th International Conference on Applications and Theory of Petri Nets, Bonn, Germany, June 1989, Proceedings]*, volume 483 of *Lecture Notes in Computer Science*, pages 491–515. Springer, 1989.

[Val98] Antti Valmari. The state explosion problem. In *Lectures on Petri Nets I: Basic Models, Advances in Petri Nets, the Volumes Are Based on the Advanced Course on Petri Nets*, pages 429–528, London, UK, UK, 1998. Springer-Verlag.

[VHHS06] V. Vafeiadis, M. Herlihy, T. Hoare, and M. Shapiro. Proving correctness of highly-concurrent linearisable objects. In J. Torrellas and S. Chatterjee, editors, *PPOPP*, pages 129–136, 2006.

[VYY09] Martin T. Vechev, Eran Yahav, and Greta Yorsh. Experience with model checking linearizability. In *SPIN*, pages 261–278, 2009.

[Wei89] William E. Weihl. Local atomicity properties: Modular concurrency control for abstract data types. *ACM Trans. Program. Lang. Syst.*, 11(2):249–283, 1989.

[WS05] Liqiang Wang and Scott D. Stoller. Static analysis of atomicity for programs with non-blocking synchronization. In Keshav Pingali, Katherine A. Yelick, and Andrew S. Grimshaw, editors, *Proceedings of the ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPOPP 2005, June 15-17, 2005, Chicago, IL, USA*, pages 61–71. ACM, 2005.

[WT15] Heike Wehrheim and Oleg Travkin. TSO to SC via symbolic execution. In Nir Piterman, editor, *Hardware and Software: Verification and Testing - 11th International Haifa Verification Conference, HVC 2015, Haifa, Israel, November 17-19, 2015, Proceedings*, volume 9434 of *Lecture Notes in Computer Science*, pages 104–119. Springer, 2015.

[YGL05] Yue Yang, Ganesh Gopalakrishnan, and Gary Lindstrom. UMM: an operational memory model specification framework with integrated

model checking capability. *Concurrency and Computation: Practice and Experience*, 17(5-6):465–487, 2005.