Reasoning about Weak Semantics via Strong Semantics

Roland Meyer and Sebastian Wolff

Abstract Verification has to reason about the actual semantics of a program. The actual semantics not only depends on the source code but also on the environment: the target machine, the runtime system, and in the case of libraries the number of clients. So verification has to consider weak memory models, manual memory management, and arbitrarily many clients. Interestingly, most programs are insensitive to the environment. Programs are often well-behaved in that they appear to be executed under sequentially-consistent memory, garbage collection, and with few clients — although they are not. There is a correspondence between the actual semantics and an idealized much simpler variant. This suggests to carry out the verification in two steps. Check that the program is well-behaved. If so, perform the verification on the idealized semantics. Otherwise, report that the code is sensitive to the environment.

Arnd is one of the few researchers who is able to switch with ease between the practice of writing code and the theory of defining programming interfaces. Discussions with him had substantial influence on the above verification approach, which we started to develop in Kaiserslautern, two offices next to his. In this paper, we give a unified presentation of our findings.

Happy Birthday, Arnd!

Roland Meyer
TU Braunschweig, e-mail: roland.meyer@tu-bs.de

Sebastian Wolff
TU Braunschweig, e-mail: sebastian.wolff@tu-bs.de

1 Introduction

Writing code is difficult. The programmer has to reason about the actual semantics of a program. The actual semantics not only depends on the code but is influenced to a great extent by the program's environment: the underlying hardware, the runtime system, and the clients. This influence results in a complexity that virtually no programmer can fully understand. For example, to speed-up execution, modern processors employ weak memory models, like total store ordering (TSO). Under TSO, every processor is equipped with a store buffer. Memory writes are put into that buffer and eventually written to memory in batches. A processor can read from its own buffer but cannot see the other buffers. Hence, each processor may observe a different memory valuation. For another example, to avoid the runtime penalty of garbage collection (GC) many programming languages require memory to be managed manually (MM). Programmers have to explicitly reclaim memory that is no longer used to prevent the system from running out of available memory. As a consequence, programs can behave unexpectedly due to unknowingly accessing reallocated memory or crash due to accessing memory no longer available. Weak memory models and manual memory management thus introduce intricate program behaviors that lead to subtle bugs which are typically hard to debug.

The intricacy of such *weak semantics* makes reasoning about programs difficult. To still argue about correctness, programmers consider, sometimes unknowingly so, a *stronger* semantics. They assume sequential consistency (SC) and garbage collection. That is, they assume memory operations to be atomic and ignore bugs due to flawed memory management.

This approach to program development is widely spread and successful. Most industrial programs lack formal correctness proofs. Instead, programmers reason informally about the strong semantics and use tests to support their reasoning. This leads to the following conjecture: reasoning about the strong semantics is sufficient. Phrased differently, correctness of a program under the simpler strong semantics entails correctness of the program under the actual weak semantics.

Obviously, our conjecture cannot be true in general. There are programs where the difference between weak and strong semantics becomes evident [9, 28], where TSO reorderings result in behavior not present under SC and where GC prevents reallocations harmful under MM. Surprisingly, this seems to be the case exclusively for performance-critical applications where experienced programmers are proficient in the weak semantics and deliberately exploit system-specific characteristics. For the majority of programs, however, the conjecture does hold. This claim is supported by the so-called *DRF theorem* [3]. It states that TSO and SC coincide, i.e., admit the exact same behaviors, provided the program is *data race free (DRF)*. Similarly, there is a *PRF theorem* [19] stating that MM and GC coincide provided the program is free from *pointer races* — roughly speaking, if no dangling pointers are used.

Interestingly, the applicability of both the DRF and the PRF theorems can be checked in the strong semantics. This is crucial since reasoning about the weak semantics — whether it is about correctness or about race freedom — is infeasible.

This allows programmers to solely reason in the strong semantics and be sure that their reasoning carries over to the actual semantics.

Our verification approach can be formulated as automating the programmer's manual reasoning. In the strong semantics, we check (i) properties like DRF and PRF and (ii) correctness. If any of the checks fails, verification fails. That is, we consider programs that do not adhere to rules like DRF buggy. Note that automation allows us to consider properties more complex than DRF and PRF that would be hard to establish manually. This makes the approach applicable to more programs. We have successfully instantiated the approach for weak memory, concurrent data structures, and heap-manipulating programs.

For weak memory, the goal is to show that TSO does not admit more behaviors than SC. In literature, this is also known as *robustness* [8] or *stability* [4]. The main insight required for the result observes that non-robustness becomes evident in computations of a particular form [7, 8]: if a program is not robust, then there is a TSO computation which is infeasible under SC where only one thread delays stores. That the computation is infeasible under SC can be detected by searching for a happens-before path from the last load of the delaying thread to the first delayed store. Finding such a path can then be done under SC. The reason for this is that the store buffer of the delaying thread is used in a restricted way during the time frame that needs to be analysed. Hence, the store buffer can be elided altogether. To that end, the original program is instrumented: instead of putting a write to the store buffer and flushing it to memory later, we modify the thread such that it performs the write to another memory location and immediately flushes it. This memory location is fresh in that no other thread reads it. Then, the delaying thread performs reads from the elided location to mimic reads from its own store buffer. Other threads are not affected by the elision. They would not observe the delayed write in the original program and also do not observe the write in the instrumented program as they do not load the elided location. Since this instrumentation does not affect the happens-before relation involving actions of the non-delaying threads, the instrumented program contains the happens-before path in question if and only if the original program does. Altogether, this means that robustness can be checked under SC.

The above instrumentation allows to check robustness of programs. So for robust programs, correctness can be checked under the simpler SC semantics. For non-robust programs, however, verification remains impossible; correctness results of SC do not carry over to TSO and an analysis of full TSO is intractable. To overcome this limitation, one can iteratively enrich the SC computations with TSO relaxations (delayed stores) via an instrumentation [5]. To that end, one uses the robustness decision procedure from above. If it terminates with a negative answer, one extracts a TSO computation that is infeasible under SC. As described above, at the heart of such a computation is a delayed store that introduces a happens-before cycle. Then, one instruments the program to explicitly buffer that store in a local register and flush it to memory later. This introduces TSO behavior of the original program into the SC behaviors of the instrumented version. Repeating this procedure yields a semi-decision procedure. If the instrumented program is eventually found robust, it

can be used to check correctness because its SC behaviors coincide with the TSO behaviors of the original program. However, it is not guaranteed that there is a finite set of instrumentations that lead to a positive robustness result.

For heap manipulating programs, we showed the PRF theorem mentioned above [19]. Intuitively, the theorem states that if no dangling pointers are used, then a program cannot detect whether or not memory is reclaimed and reused. The reason for this is the following. In order to detect whether a memory cell has been reclaimed and reused, a program needs to acquire a pointer to that memory cell, wait for it to be reclaimed, and then observe that it is in use again. After the reclamation, however, any pointer to that memory becomes *dangling*. Those pointers remain dangling when the referenced memory is reused. We consider any subsequent usage of a dangling pointer as a bug. We call those bugs *pointer races*. They are indeed races because the access is not synchronized with the preceding deletion — the program uses a pointer after it's referenced memory was deleted and is thus prone to crash due to a *segmentation fault* or a *bus error* because the memory might as well be returned to the underlying operating system instead of being reused. However, if pointers are never dereferenced and used in conditionals after they become dangling, then a program cannot find out whether another pointer holds the same value. In particular, it cannot decide whether the result of an allocation is already referenced by a dangling pointer. Hence, it cannot know whether an allocation reuses reclaimed or fresh memory. As a consequence, the behavior is the same if memory is reused or not at all. That is, the program behaves identical under both GC and MM.

Another result for concurrent heap manipulating programs studies the interaction of threads communicating via a shared memory [22]. In particular in fault-tolerant implementations, like lock-free programs, threads typically do not rely on other threads having finished their work. This means that if a thread starts to alter a shared structure but does not finish its update due to an interruption or a failure, then any thread can *take back* the shared structure to a consistent state (wrt. to a sequential execution). Put differently, after every atomic update every thread can continue and finish its operation even though the update left some shared structure in an inconsistent state. Using standard data flow analyses one can collect a set of atomic updates that are likely to over-approximate the updates of the original program. We call a sequential program which repeatedly executes those updates a *summary*. Such a summary is then a candidate program for over-approximating the behavior of the original program on the shared heap. Hence, if correctness can be judged solely by the contents of the heap, then the sequential summary can be used for checking correctness of the concurrent program under scrutiny. For this approach to be sound, the summary must cover all possible updates of the original program. We showed that this can be done by considering the parallel composition of the summary and a single thread of the program: if the summary can mimic every shared heap update of the retained thread in this two-threaded program, then the summary indeed over-approximates the shared heap updates of the original program.

The results presented above simplify the analysis. Instead of performing an expensive analysis, one proves that a simpler analysis implies the desired results. In the automated program verification literature, a standard framework for this kind of

results is *abstract interpretation* [11]. Abstract interpretation is a powerful tool for reasoning about a weak semantics using a strong semantics. The correspondence of the semantics is then guaranteed by the underlying theory. More specifically, to use abstract interpretation one has to show — in a manual proof — that the strong semantics is a *safe approximation* of the weak semantics. That is, one has to show that every behavior of the weak semantics is *mimicked* by a behavior of the strong semantics. Note that this property is shown independently of the program under scrutiny. On the one hand, this allows the framework to be applicable to any program once shown that the strong semantics is a safe approximation. On the other hand, however, it may not leverage properties that hold only for a fragment of programs. In particular, abstract interpretation cannot reason about properties like the soundness check for summaries because it is a property of the entire state space rather than a (safety) property of an individual computation. The reason for this is that abstract interpretation shows the reduction *statically*; it holds for any program. We, however, suggest to add a *dynamic* check at verification time which ensures that the simpler analysis indeed provides a correct result. That is, we sacrifice universal applicability to provide more efficient analyses for a fragment of programs.

Our experience shows that relying on dynamic checks to judge whether a simpler analysis is correct makes verification much more efficient than classical approaches, like abstract interpretation. Moreover, we found that the concept is very flexible; even performance critical code can be handled with the concepts discussed above.

The rest of the paper is structured as follows. Section 2 introduces a general framework for reasoning about programs using strong semantics. This framework is then instantiated in Sections 3 to 5 for TSO programs, shared memory programs, and pointer programs, respectively. Section 6 concludes the paper.

2 From Weak Semantics to Strong Semantics

In this paper we consider the task of verification. More specifically, we consider the problem of checking a safety property *safe* for a set of computations $\mathscr{W}$. Intuitively, $\mathscr{W}$ contains all computations of a given program where a computation is a sequence of executed actions. Formally, we refrain from making this set any more precise; we will appropriately instantiate it as needed. The safety property is given as a predicate *safe* over computations. That is, the task is to check whether $\mathit{safe}(\tau)$ holds for every computation $\tau \in \mathscr{W}$, denoted by $\mathit{safe}(\mathscr{W})$ for brevity.

It is well known that the above verification task is undecidable for most interesting programs. To fight this theoretical limitation, abstractions and approximations are applied. A standard approach for program abstraction is *abstract interpretation* [11, 12]. The goal of abstract interpretation is to execute a given program in an *abstract semantics*. Roughly, this means that rather than using actual (concrete) values the program uses abstract values. An abstract value represents a set of concrete ones, potentially infinitely many. To guarantee soundness of such an approach, abstract interpretation requires the abstract semantics to be a *safe approximation* of

the concrete semantics. Intuitively, this means that every action a program can take in a concrete execution can be mimicked in the corresponding abstract run. That is, one has to show for all possible programs that abstract computations over-approximate the reachable control states of concrete computations.

Abstract interpretation has proven to be a powerful tool for verification. However, the state space $\mathscr{W}$ might be too large to enumerate despite aggressive finite abstractions. In particular, weak memory and concurrent shared memory programs suffer from a severe state space explosion. Our goal is to provide reductions that allow verification for cases where abstract interpretation alone cannot. To that end, we suggest to identify a simpler semantics to conduct the verification in. That is, we suggest to consider a smaller/simpler state space $\mathscr{S}$ the successful verification of which implies the desired property of $\mathscr{W}$. Intuitively, we want to identify $\mathscr{S} \subset \mathscr{W}$ such that $\mathscr{S}$ can be verified efficiently with existing abstract interpretation techniques. Technically, we do not require $\mathscr{S} \subset \mathscr{W}$. In fact, we do allow $\mathscr{S}$ to stem from another program than the one under scrutiny. In any case, the analysis of $\mathscr{S}$ may *skip* computations of $\mathscr{W}$. Hence, one has to show that the verification result of $\mathscr{S}$ carries over to $\mathscr{W}$ indeed. In the spirit of abstract interpretation, we perform an over-approximation: we require that a *successful* verification of $\mathscr{S}$ implies correctness of $\mathscr{W}$. More formally, we require that $\mathit{safe}(\mathscr{S})$ implies $\mathit{safe}(\mathscr{W})$. The reverse needs not hold.

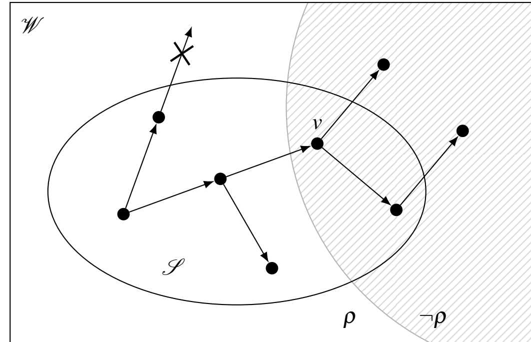

Fig. 1 A borderline computation, like v, is a computation that may *leave* $\mathscr{S}$. More specifically, v can perform actions under $\mathscr{W}$ that are not discovered by (not included in) $\mathscr{S}$. This jeopardizes sound verification. To detect such situations, we use a predicate ρ over $\mathscr{S}$. This predicate is evaluated at analysis-time, e.g., after $\mathscr{S}$ has been enumerated.

However, the above reduction idea is flawed. The problem is that we require $\mathit{safe}(\mathscr{S}) \implies \mathit{safe}(\mathscr{W})$. In a manual proof for a specific program one can reason about the $\mathscr{W}$-computations that are skipped by $\mathscr{S}$. For automated techniques, however, the above implication needs to be established for *every* program. To simplify this task, we weaken the requirement. We allow a reduction to supply a predicate ρ which judges whether or not an analysis of the simpler $\mathscr{S}$ is sound. For example, ρ could check for data races to judge whether or not the DRF theorem applies.

Intuitively, in terms of abstract interpretation, ρ characterizes those computations for which the abstract semantics is a safe approximation of the concrete semantics. That is, in computations for which ρ evaluates to true any action performed in $\mathscr{W}$ can be mimicked in $\mathscr{S}$. Consider Figure 1 for an illustration. Consequently, an analysis has to additionally check whether or not $\rho(\mathscr{S})$ holds. If it does hold, then the analysis

is guaranteed to be sound. Otherwise, there are *borderline* computations (cf. v in Figure 1) that can perform actions under $\mathscr{W}$ not captured by $\mathscr{S}$. Since ρ is checked dynamically, the result may differ from program to program. Those programs for which ρ cannot be established we deem buggy and verification fails.

Technically, we require $\rho(\mathscr{S}) \wedge \mathit{safe}(\mathscr{S}) \implies \mathit{safe}(\mathscr{W})$ to hold. As we will see in more detail later, this has two advantages. On the one hand, this weakened property can be established for arbitrary programs. It is thus amenable for automated techniques as it need not be re-proven for every program of interest. On the other hand, the soundness check $\rho(\mathscr{S})$ can be done by the actual analysis. Typically, it can be implemented efficiently, taking insignificant time compared to the actual analysis of $\mathscr{S}$. Lastly, note that we do not pose any restriction on ρ. In particular, we allow ρ to be any predicate over *sets* of computations rather than just a predicate over *single* computations. This renders ρ strictly more powerful than a safety property.

The following summarizes how we reduce the verification of $\mathscr{W}$ to $\mathscr{S}$.

Reduction I

Given: A set of computations $\mathscr{W}$ and predicate *safe*.

Task: Find a set of computations $\mathscr{S}$ and a predicate ρ over $\mathscr{S}$ such that:

$$\rho(\mathscr{S}) \wedge \mathit{safe}(\mathscr{S}) \implies \mathit{safe}(\mathscr{W}) \, . \tag{R}$$

With such a reduction at hand, we suggest the following approach: (i) enumerate $\mathscr{S}$, (ii) check $\mathit{safe}(\mathscr{S})$, and (iii) check $\rho(\mathscr{S})$. Verification fails if one of the checks fails. Otherwise it succeeds because (R) guarantees that $\mathit{safe}(\mathscr{W})$ holds.

The main goal in establishing a reduction is to prove property (R) once $\mathscr{S}$ and ρ are chosen. The following is a promising proof strategy for this task. Introduce a relation $\approx \subseteq \mathscr{W} \times \mathscr{S}$ among computations. Intuitively, $\tau \approx \sigma$ states that σ mimics in $\mathscr{S}$ the behavior of τ in $\mathscr{W}$. So if τ is a safety violation, then also σ is a safety violation. Now, if there is such a σ for every τ, denoted $\mathscr{W} \prec \mathscr{S}$, then $\mathscr{S}$ covers all behaviors of $\mathscr{W}$. That is, if there is a safety violation $\tau \in \mathscr{W}$, then there must be a safety violation $\sigma \in \mathscr{S}$. Hence, it is sufficient to verify $\mathscr{S}$.

Altogether, we suggest to refine the reduction approach from above as follows.

Reduction II

Given: A set of computations $\mathscr{W}$ and predicate *safe*.

Task: Find $\mathscr{S}$, $\prec$, and ρ such that:

$$\rho(\mathscr{S}) \implies \mathscr{W} \prec \mathscr{S} \tag{R1}$$

and

$$\neg\mathit{safe}(\mathscr{W}) \wedge \mathscr{W} \prec \mathscr{S} \implies \neg\mathit{safe}(\mathscr{S}) \, . \tag{R2}$$

As noted before, an analysis may find that the provided reduction does not allow to soundly verify a program. In the case of TSO programs, for instance, the analysis may explore the SC executions and find a data race. Following the approach so far, the analysis must deem the program incorrect even if no *proper* correctness violation was found. This makes verification imprecise. To fight the false-positives of such an analysis, one can dynamically weaken $\mathscr{S}$ to include computations that make the soundness check fail. To that end, we adapt the verification procedure suggested above. If the check $\rho(\mathscr{S})$ fails, then we extract a witness computation. Such a computation is a borderline event, as depicted in Figure 2. The computation is borderline because it allows behavior under $\mathscr{W}$ which $\mathscr{S}$ does not allow. Put differently, the borderline computation can *leave* $\mathscr{S}$ and thus harms soundness. Instead of deeming the program incorrect, we can extend $\mathscr{S}$ to include the missing behavior. The adding process can be guided by ρ: ρ detects a borderline action that can be done under $\mathscr{W}$ but not under $\mathscr{S}$. Then, we add this missing behavior to $\mathscr{S}$ and repeat the analysis. In the next iteration, either the analysis can be shown sound or ρ reveals another missing behavior. Repeating the above approach thus yields a semi-decision procedure. If a bug is found or the analysis becomes sound eventually, then the procedure stops with a definite result. However, the above process may loop indefinitely because there is no guarantee that there are only finitely many behaviors missing in $\mathscr{S}$ in general.

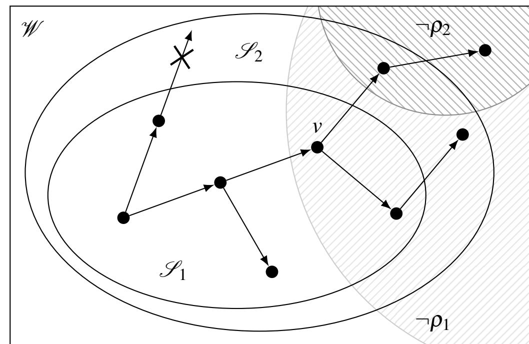

Fig. 2 One-step extension $\mathscr{S}_2$ of $\mathscr{S}_1$. The extension $\mathscr{S}_2$ includes the borderline events of $\mathscr{S}_1$. The borderline events are given by ρ_1, like v. The extended $\mathscr{S}_2$ has another borderline event according to ρ_2.

In the remainder of the paper, we discuss several instantiations of the above framework to ease reasoning about weak memory models, concurrent shared memory programs, and concurrent pointer programs.

3 From Total Store Ordering to Sequential Consistency

A natural memory model is sequential consistency (SC) [25]. Under SC, all operations appear instantaneous and atomic. That is, the effect of an operation is immediately visible to all threads. This simplicity makes it easy to reason, both manually and automatically, about SC executions. For performance reasons, however, SC is not

implemented in today's processors. Instead, weak memory models are used which may reorder or buffer operations. A widespread weak memory model is total store ordering (TSO) [34], which is used on x86 machines, for example. TSO relaxes the atomicity of SC for memory writes. Instead of writing directly to memory, store operations are placed in a per-thread FIFO *store buffer*. Eventually, the store buffer is flushed and the memory is updated. Only after the buffer has been flushed, other threads see the memory update that may have happened much earlier. However, for the thread executing the store, the processor establishes the illusion that it was already performed — technically, this *early read* lets threads read from their own store buffer. Altogether, this causes every thread to have its own local perception of the shared memory. For a program to synchronize the individual perceptions of threads, TSO provides *fence* instructions which force flushing the store buffer to memory. It is then the programmer's (or compiler's) task to introduce fences such that the program does not read stale values and behaves correctly. This task can be cumbersome and requires an understanding of how the underlying weak memory affects a programs behavior. To address this obstacle, we discuss two reductions that allow for reasoning about TSO under SC. The first reduction comes in the form of a programming guideline and is amenable for manual reasoning. The second reduction is built for automated reasoning and focuses on more complex programs that cannot be handled with the first reduction. Before we go into the details of those reductions, we give a characterization of SC and distinguish it from TSO.

Characterization. In the literature, the most common approach to reason about weak memory models is to consider the *trace* [35] of a computation, rather than the sequence of executed instructions. The trace of a computation is a graph. Its nodes correspond to the executed operations. The arcs represent the *happens-before (hb)* relation. Intuitively, $(o_1, o_2) \in hb$ means that o_2 relies on the result of o_1. Technically, *hb* is a union of four different relations: *program order (po)*, *store order (st)*, *source relation (src)*, and *conflict relation (cf)*. The program order implements the per-thread sequence of executed operations. The store order is a per-address serialization of all store operations. A store is in source relation to a load if the load reads the value written by the store. The conflict relation, intuitively, relates loads to stores if the value read was over-written by the store. With this, SC traces are guaranteed to be acyclic due to the operations being instantaneous and atomic.

Lemma 1 ([35]). *A trace is acyclic iff it is induced by an SC computation.*

On the other hand, TSO computations can produce cycles due to store buffering. More precisely, a thread can load a value that was already over-written by a store of another thread. Loading the over-written value is indeed possible under TSO because the store may have been buffered and only flushed after the load is performed — a scenario not possible under SC. For an example, consider Figure 3.

DRF Theorem. To tackle the complexity of TSO computations, one can rely on the well-known DRF theorem [3]. It states that every TSO trace is acyclic provided the program is *data race free (DRF)* under SC. A data race occurs if there are two unsynchronized accesses to the same memory location at least one of which is a

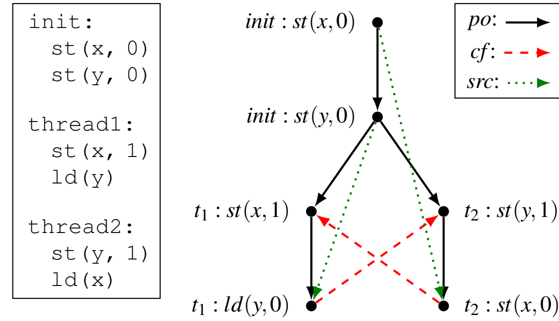

Fig. 3 A program and a possible TSO trace. The load of t_1 and the store of t_2 are in conflict relation because the load is executed after the store is put into the store buffer but before it was flushed. Hence, the load reads a stale value—the program is not properly synchronized. The behavior here is not present under SC because the store would immediately be visible and thus read by the other thread.

store. Intuitively, they are unsynchronized if at one point the program could execute either of the operations. Formally, one introduces a *synchronizes with (sw)* relation to explicate the behavior of synchronization primitives, such as fences. (We omit a more formal discussion of *sw* for brevity.) Then, two actions are unsynchronized if they are not related by $(po \cup sw)^+$. The key insight here is that conflict arcs stem from data races; indeed, the program is free to choose whether to execute the load or to flush the store involved in a conflict. So if a program is DRF, then it contains no conflict arcs. One can easily see that the absence of such arcs makes *hb* acyclic.

Theorem 1 ([3]). *Data race freedom implies that the SC and TSO traces coincide.*

With this theorem, we can come up with a reduction following the format from Section 2. For $\mathscr{W}$ and $\mathscr{S}$ we choose the sets of traces induced by TSO and SC, respectively. Then, we instantiate the predicate ρ such that it checks for data races. And we use equality as a relation among computations. Then, property (R2) follows immediately assuming there is a dedicated *unsafe* control location. For (R1) we have to show that data race freedom of a program can be judged by its SC traces. To see this, consider a shortest TSO computation τ which raises a data race. By minimality, there is exactly one conflict arc. This arc relates some load and store instruction. Again by minimality, the store is the *po* latest operation of the executing thread (because actions of the same thread are never in conflict). Hence, we can remove that store and get a DRF trace. Then, there must be an SC computation σ with the same trace. Moreover, the store can be executed after σ because trace equality implies that all threads are in the same control state. This establishes the same conflict arc in the resulting trace as in the trace for τ. Altogether, this means that scanning SC for data races suffices to show a program DRF.

Theorem 2. *A program is data race free under SC iff it is so under TSO.*

This justifies our choice of ρ and proves our reduction sound. That is, one can rely on the simpler SC semantics to check for data races and to perform an actual analysis.

Robustness against TSO. The above development works well for general purpose programs because they tend to be data race free. However, there are many areas where data races cannot be avoided, for example, in concurrency libraries and performance-critical code [13,20,26,31]. Despite the data races, we observe that there are programs that do not show non-SC behavior. This is the case because DRF does not characterize SC — it is strictly stronger. To see this, consider Figure 3: if we remove one of the conflict arcs there we retain a data race but get an acyclic and thus SC trace. So we seek a method for *precisely* identifying under SC if a program is *robust*, i.e., allows only SC behavior even if executed under TSO. We do this in two steps. First, we establish a characterization of robustness. Then, we show how to employ this characterization for an SC-based robustness check.

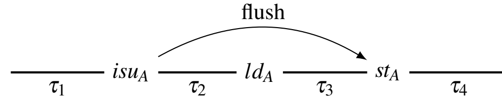

Fig. 4 A TSO attack [7]. Instructions isu_A, ld_A, and st_A are executed by the attacker. The pair (isu_A, st_A) makes visible when the delayed store is executed and flushed, respectively.

For a characterization of robustness we show that if a program is not robust, then it allows for an *attack* [6, 7]. An attack is a computation of a particular form that exhibits TSO behavior, as shown in Figure 4. The rational behind attacks is the following. If a program has non-SC behavior, then it must be due to a delayed store, st_A. Moreover, the delayed store must be on a happens-before cycle due to Lemma 1. Now, assume for a moment that there is only one delayed store. Then, this delay must overtake a load of the same thread, ld_A, for otherwise the store would be flushed immediately. Similarly, ld_A must be on the happens-before cycle since the store could be flushed immediately otherwise. This means that the happens-before cycle is of the form $ld_A \rightarrow_{hb}^{+} st_A \rightarrow_{po}^{+} ld_A$. (For Figure 4 note that isu_A represents the store being buffered and st_A the store being flushed; hence, st_a is indeed *po*-earlier than ld_A although the figure suggests otherwise; in the trace isu_A and st_A appear as one node.)

This cycle, however, is based on the assumption that there is only one delayed store. Our reasoning remains valid if only one thread delays stores. Intuitively, st_A is the first delayed store on the happens-before cycle. Any prior store can be flushed immediately because it is in the *SC prefix* of the computation. Any later delayed store cannot participate in the $ld_A \rightarrow_{hb}^{+} st_A$ part of the happens-before cycle because the FIFO property of the TSO store buffer guarantees that it is flushed only after st_A is. Interestingly, one can show the needed property, namely, if a program is not robust then there is a non-SC computation where only one thread delays stores [8]. Due to its technical nature, we refrain from elaborating on this result and refer the reader to [6, 8] instead. Altogether, this means that if a program is not robust, there is an attack. Due to the happens-before cycle in an attack, the revers holds too. The following theorem summarizes the characterization result.

Theorem 3 ([7]). *A program is robust iff there is no attack.*

Towards a practical reduction, it remains to check for attacks under SC. This requires to find a delayed store and a subsequent load that are on a happens-before cycle. Such a cycle cannot be observed under SC as postulated by Lemma 1. However, an attack makes only limited use of the store buffer: only a single thread, the *attacker*, delays stores. To mimic such delays under SC we make the store buffer of the attacker explicit [6, 7]. To that end, we instrument the program under scrutiny. To delay a store to address a of the attacker, we replace it with a store to an auxiliary address a' which is not used by any other thread. The attacker can read the value early as it knows of address a'. All other threads continue to read the stale value at address a. We do not need to handle flushes, because the instrumentation can stop execution if the happens-before cycle is *closed*. To do that, we instrument all other threads, the *helpers*. Technically, we force the computation to produce a happens-before cycle. That is, with respect to Figure 4, we require every action in τ_3 to participate in the cycle and prevent any other instruction from being executed. More precisely, we require each action act of a helper to satisfy $ld_A \rightarrow_{hb}^{+} act$. If a helper thread already contributed an action, then any subsequent action continues the happens-before path using program order. Whether an action has already been contributed can be kept in the control state of helpers threads. Otherwise, act cannot continue the happens-before path from ld_A using program order. Let a denote the address which is accessed by act. If a has been loaded on the happens-before path, then act has to be a store to a because only the conflict relation allows it to continue the hb path. Otherwise, if a has seen a store, then any access allows to continue the hb path through act: using the store relation in case of a store, and the source relation in case of a load. The access information can be kept in a per-address auxiliary address. Altogether, a helper thread closes the happens-before cycle if it accesses the address targeted by st_A. In such a case the program jumps to a dedicated goal state. So if an instrumented program reaches this goal state, then the program is guaranteed to contain a hb cycle and thus to be not robust. Otherwise, the program is robust and shows the exact same behavior as the original program under TSO.

Theorem 4 ([7]). *A program is not robust iff its instrumentation reaches the goal state.*

The above development allows us to formulate the approach as a reduction. The set $\mathscr{W}$ contains the TSO traces of the program under scrutiny. For $\mathscr{S}$ we choose the set of SC traces that stem from an instrumented version of the program. The predicate ρ simply checks if the goal state was reached. And we use equality as relation among traces. Then, both (R1) and (R2) follow from the above theorems.

Supporting Non-Robust Programs. The above method handles all robust programs and allows their verification to rely on the simpler SC semantics. Programs that are not robust, however, do not profit from that development. That is, the above method does not guarantee soundness for non-robust programs. Unfortunately, TSO resists efficient automated reasoning [23,30]. To overcome this problem, we can patch non-SC behavior into the SC semantics using the approach described in Section 2 paired with the above insights [5]. So consider a program that is not robust. The above instrumentation makes visible the delayed store which allows for a happens-before

cycle and thus non-SC behavior. More precisely, non-robustness yields as witness an attack like the one from Figure 4 — a borderline computation in the wording of Section 2. From such an attack, we can extract the attackers sequence of actions performed while st_A was in the store buffer. That is, we can extract which store is delayed and when it is flushed. Then, we instrument the program to make the delay explicit. To that end, upon arriving at the to-be-delayed store, the thread makes a non-deterministic choice. Either it continues it's original computation, or it executes the instrumentation. The instrumentation writes the store into an auxiliary register instead of writing it to memory. Then the actions from the extracted sequence follow. However, they are adapted such that early reads of the buffered value are read from the auxiliary register. After the sequence is executed, the write is flushed to memory, i.e., copied from the auxiliary register to the original target address. Then, the thread jumps back into its original program. Technically, the instrumentation has to handle further stores that are delayed during the extracted sequence. They are handled in the same way as described above. Note that since the extracted sequence is finite, the instrumentation requires only finitely many auxiliary registers. Altogether, this allows to patch finite sequences of delayed stores into SC. The stores we choose here stem from attacks making the program non-robust. Hence, the instrumented program is guaranteed to contain more behaviors than the original one.

Repeating the above patching allows to continuously enrich the SC behavior with TSO behavior. This yields a semi-decider for safety. If there is a safety violation under TSO, then this behavior is eventually added to the instrumentation and thus explored under SC. Otherwise, it may not terminate at all; there may be infinitely many non-SC behaviors that are iteratively added. So the algorithm will not converge to an instrumentation that is robust.

4 From Many Threads to Few

For concurrent programs, informal reasoning and testing techniques can only explore a fraction of all possible program behaviors and are likely to miss delicate corner cases [10]. The reason for this are the many thread interleavings and the severe state space explosion of concurrent programs. This problem becomes worse when considering lock-free code. There, correctness arguments require rigorous formal proofs due to the subtle thread interactions. Providing a manual proof is a cumbersome task and requires a deep understanding of the program to be verified [14, 15, 16, 24, 32, 33, 37]. Hence, we strive for automation. For practicality, we need techniques that can handle parametrized programs. That is, we need techniques that can establish correctness of a given lock-free program for any number of threads. Our goal is to develop a reduction from parametrized programs to programs with few threads. We focus here on lock-free programs. It is an inherent characteristic of lock-freedom that allows for the desired reduction. Towards the result, we discuss the working principles of lock-free programs first.

Lock-Freedom. Lock-freedom is a progress property of programs [20]. It guarantees that a thread can make progress even if arbitrary other threads stall or fail. To see how this influences the way in which programs are written, consider as an example a program which maintains a shared structure, like a stack or a list, that is used for inter-thread communication. A logical update of the structure, like adding or removing elements, may (and in most cases does) require multiple physical updates, i.e., multiple memory writes [14,17,20,27,28,29,38]. The naive solution would use a critical section to guard the sequence of memory updates from interference of other threads. Lock-free programs, however, do not allow the usage of locks. Now assume a thread wants to perform a sequence of updates but fails half-way through, that is, performs only some of its updates. With respect to lock-based programming, this means that the thread executed only a part of its critical section. This typically leaves the shared structure in an *inconsistent* state wrt. the invariants of the lock-based implementation. Now, to achieve lock-freedom, other threads must still be able to progress. That is, they have to anticipate the incomplete sequence of updates and either roll back or finish that sequence. In the literature, this is commonly referred to as *helping* [20]. And indeed, this is what other threads do. They first make sure that the structure is in a consistent state, helping another thread updating the structure if necessary, before they continue with their own operation. For an example, consider a lock-free singly-linked queue which maintains a shared pointer to the last list segment. Appending a new segment requires to link the new segment and to update the shared pointer. This requires two subsequent updates. Hence, a thread may find that the shared pointer does not point to the last element. If so, it will first move it to the last segment before it continues to, e.g., add another element to the end of the list.

Summaries. It is this helping that we exploit for the desired reduction. It is typically implemented in such a way that a thread, by inspecting a shared structure, can find out whether or not the structure is in a consistent state and what actions are required to make it consistent if it is not. That is, inconsistencies and the required fixes can be deduced from the shared structure itself — a thread does not need knowledge about other threads nor of the history of the structure. Hence, we say that the process of helping is *stateless* [21,22]. Consequently, those updates can be collectively executed by a single thread which we call *summary*. So assume for a moment we had such a summary thread. Then, that summary could produce any shared heap the original program can produce. Hence, if correctness can be judged by the contents of the shared heap (which is a reasonable assumption[1] in practice), then it suffices to analyse this single thread. In the following, we discuss how to generate a candidate summary and check whether it is a proper summary [21,22].

A candidate summary can be generated by inspecting the program under scrutiny. As mentioned before, lock-free programs update shared structures on a memory-word level. In practice, the prevalent pattern to update the content of a single memory location is the following [14,17,20,27,28,29,38]: (i) read the memory contents of the location to be updated, (ii) compute a new value for the location (based on the

[1] For linearizability proofs, for example, one encodes the sequence of linearization points in the shared heap [1] and checks whether a non-linearizable sequence can be produced by the program.

previously read value), and (iii) atomically write the new value to memory if the valuation has not changed (typically implemented using Compare-And-Swap) or retry otherwise. Instances of this update mechanism can be found by a syntactic analysis of the program under consideration. The summary is then simply an indefinite loop every iteration of which non-deterministically choses and atomically executes one of those update instances. It is worth stressing that the resulting program is sequential. Moreover, executing the instances of the above pattern atomically allows to apply standard compiler optimizations resulting in a summary that is much shorter and more easily understandable than the original program. While understandability of the summary is not important for an automated technique, reducing its code size can be beneficial. To sum up, we create a candidate summary by syntactically identifying shared updates in the original program assuming atomicity.

Summaries in Verification. An instantiation of the framework from Section 2 exploiting summaries is in order. To that end, let $P = \prod_{i=1}^{k} T_i$ be the original program, a parallel composition of k identical threads T_i, and let Q be a candidate summary. Then, $\mathscr{S}$ is the set of all computations of P. For $\mathscr{S}$ we choose the set of computations of the program $T_1 \parallel Q$, a parallel composition of a single thread T_1 from P and the single thread Q. Predicate ρ checks whether Q is a proper summary of P. We discuss how to do this later. As a relation among computations we choose an equivalence relation $\approx$ such that τ and σ are related if they coincide on the shared structures and the thread-local state of T_1. Property (R2) follows immediately provided safety can be concluded from the state of the shared structures as assumed initially. For (R1), we proceed as follows. Given a computation $\tau \in \mathscr{W}$, we replace all actions of threads $T_2, \ldots, T_k$ that update a shared structure by actions of Q that have the same effect and simply remove all other actions of threads $T_2, \ldots, T_k$. The replacement is feasible since Q is a proper summary due to the premise of (R1). The removal is feasible because actions that do not change the shared structures cannot influence other threads by assumption. Altogether, this process yields a computation $\sigma \in \mathscr{S}$ with $\tau \approx \sigma$, as required for (R1). The intuition behind this construction is the following: thread T_1 can be influenced by other threads only through updates of the shared structures; however, it cannot detect which entity performs the updates of those structures. Hence, T_1 cannot detect whether it runs in parallel with $T_2, \ldots, T_k$ or just with Q; it's behavior is the same. To conclude the reduction, it remains to discuss ρ.

Checking Summary Candidates. As shown by our experiments, the procedure for generating candidate summaries works well for common lock-free implementations from the literature [21, 22]. Nevertheless, the process may give a candidate summary which is *missing* updates of the original program. This can be the case, for instance, if the read value is changed back and forth such that the final check of the pattern erroneously concludes that the structure was not changed, known as the *ABA problem* in the literature [28]. Hence, we require a method for checking whether a candidate summary is indeed a proper summary. In the remainder of this section we present such a check and embed it into the framework from Section 2. For simplicity, we

assume that the only way threads can influence each other are updates of shared structures.[2]

We now turn towards checking whether or not the candidate summary Q is a proper summary, as required to be done by ρ. So let Q be a candidate summary that misses some required update of the program under scrutiny. That is, there is a shortest computation $\tau = act_1.\dots.act_n.act_{n+1}$ from $\mathscr{W}$ such that action act_{n+1} performs an update of a shared structure that Q cannot perform. Let us denote the state of the shared structures after $\tau_i = act_1.\dots.act_i$ by s_i. That is, no sequence of Q-actions can transform s_n into s_{n+1}. Wlog. the last action act_{n+1} is performed by thread T_1.[3] Since τ is the shortest such computation, Q can mimic all updates in the prefix τ_n. So we can perform for τ_n the same construction we used to establish (R1) above. That is, we construct a computation $\sigma \in \mathscr{S}$ with $\tau_n \approx \sigma$. Since act_{n+1} is performed by T_1, we get $\tau \approx \sigma.act_{n+1} \in \mathscr{S}$. This means, every missing update, like the one from s_n into s_{n+1}, can be found in $\mathscr{S}$. So, ρ can simply collect all updates from $\mathscr{S}$ and check whether Q can perform them too.

Missing Updates. Altogether, it suffices to analyse the two-threaded program $T_1 \parallel Q$ instead of the much more complicated P. This allows us to use existing techniques for programs with a fixed number of threads. Moreover, note that Q has a particular form which may allow for further optimizations.

Although our experiments showed that the summaries generated by the above approach work well in practice [22], they may miss updates. A missing update is a transformation of the shared heap. Instead of aborting the analysis in such a case, one can collect those updates in a set *Up*. Then, one repeats the analysis. This time, however, the updates from *Up* are applied in addition to the summary. Applying an update $s \rightsquigarrow s'$ from *Up* to a computation means to replace the shared heap s with s'. This procedure is repeated until saturation of *Up*. This yields a semi-decider because it is not guaranteed that there are finitely many missing updates. Nevertheless, this approach has been shown effective and efficient in [36].

5 From Manual Memory Management to Garbage Collection

A major obstacle in reasoning about programs relying on manual memory management is the fact that memory is explicitly deleted and can be reused immediately after the deletion. This allows for pointers to become *dangling* after the memory they reference is deleted. After a subsequent reuse of the deleted memory, the dangling pointer can still be used to modify the memory contents without the reusing thread knowing. As shown in the literature DBLP:conf/podc/MichaelS96 [28, 29], this can lead to corruption of program invariants and thus to undesired behavior. Under

[2] Technically, one separates the memory into a shared and a per-thread owned part and checks whether threads comply to this separation, that is, ensure that threads never read/write from/to the parts owned by other threads. The actual separation is a parameter to our result and can thus be instantiated as needed [22].

[3] This can be achieved by a simple renaming of threads.

garbage collection, this problem does not arise. Any reference would prevent the memory from being reused. Put differently, an allocation guarantees exclusive access under GC but does not do so under MM. This makes it significantly harder to reason about MM than about GC [1,2,19,39]. To tackle the problem, we establish a practical reduction from MM to GC [18, 19]. Let $\mathscr{W}$ and $\mathscr{S}$ be the sets of computations of a program under MM and GC, respectively.

General Purpose Programs. The main idea of the result is to allow dangling pointers but prevent them from begin used. Later, we generalize this result to allow for certain accesses of dangling pointers. Towards the result, we need to characterize what it means for a pointer to be dangling. Technically, it is easier to define when a pointer is *not* dangling. We call such non-dangling pointers *valid* pointers. Initially, no pointer is valid. A pointer becomes valid, if it is the target of an allocation or if it is assigned from a valid pointer. A valid pointer becomes invalid if its referenced memory location is deleted or if it is assigned from an invalid pointer. Basically, this means that allocations make pointers valid, deletions make pointers invalid (or dangling), and all other operations simply *spread* (in)validity. We stress that an allocation makes valid the receiving pointer only; every other invalid pointer remains invalid even if its referenced address is reallocated. With this definition, using an invalid pointer is said to yield a *pointer race*.

Definition 1 ([19]). A computation $\tau.act$ raises a pointer race if action *act* (i) dereferences, (ii) compares, or (iii) deletes an invalid pointer.

Now, we observe that pointer race free (PRF) programs are invariant to memory reuse. That is, a PRF program cannot detect whether the result of an allocation is fresh, i.e., has never been allocated before, or if it was already in use and has been deleted. To see this, assume a program seeks to draw such a conclusion. To do so, it requires a pointer p to address a before a is deleted and a pointer q to a that wlog. receives the address from an allocation. To tell whether or not the allocation for q reused the deleted a the program has to compare the addresses held by p and q. However, the deletion of a rendered p invalid. And it remains invalid through the reallocation involving q. So the comparison of p and q raises a pointer race. Altogether, this means that pointer race free programs indeed cannot distinguish fresh and reused memory. Our goal is to instantiate the framework from Section 2 to reduce MM to GC for pointer race free programs. We use for ρ a predicate that decides whether or not a given computation is pointer race free. We showed that such a check is effective and can be implemented efficiently with low overhead to an actual analysis [19]. Also note that this reduction allows to use existing tools which rely on exclusivity of allocated memory as discussed above.

Towards a reduction result we need to find an appropriate relation among computations as suggested by (R1). A non-trivial relation is required because in general GC yields a proper subset of MM computations. In the above example, MM allows to reallocate a for q. Under GC this is not possible. Instead, an allocation would give a fresh address b. In order to find an appropriate relation, recall that invalid pointers cannot be used. Hence, their valuation does not matter. It suffices to relate the valid pointers. Those pointers coincide with respect to an isomorphism which states how

memory reuse is elided (mapped) to fresh memory. So we introduce an equivalence relation $\approx$ such that two computations are related if they end up in the same program counter and in the same valid heap up to an isomorphism. For this relation, one can show the desired first part, (R1), of the reduction.

Theorem 5 ([19]). *If the program under scrutiny is pointer race free then* $\mathscr{W} \approx \mathscr{S}$.

To establish this result, one proceeds by induction over the structure of computations and constructs a new computation with an appropriate isomorphism to elide reallocations. For the technical details of that construction we refer the reader to [18, 19]. With this main result, the second part, (R2), of the reduction follows easily. If there is an unsafe computation in $\mathscr{W}$, then $\mathscr{W} \approx \mathscr{S}$ yields a related computation from $\mathscr{S}$. By the definition of $\approx$ both computations end up in the same program counter. Wlog. this implies that also $\mathscr{S}$ is unsafe.

Performance-Critical Programs. Unfortunately, the restriction to pointer race free programs is too strong to handle performance-critical code. Lock-free programs, for instance, typically perform memory reads optimistically to avoid synchronization overhead on the read side. Such optimistic accesses patterns, however, use dangling pointers and thus suffer from pointer races. To prevent harmful accesses that can lead to system crashes or integrity corruption, those patterns include checks to guarantee that if a dangling pointer was used it was used in a *safe* way. These checks necessarily need to compare dangling pointers. Therefore, we adapt the notion of pointer races from above to tolerate such optimistic access patterns. To that end, we mark the result of memory reads which dereference an invalid pointer as strongly invalid and require those strongly invalid values never to be used. The development then follows the one from ordinary pointer races.

Definition 2 ([19]). A computation $\tau.act$ raises a strong pointer race (SPR) if action *act* (i) deletes, or (ii) writes to memory dereferencing an invalid pointer, or if *act* (iii) contains a strongly invalid value otherwise.

The relaxation to strong pointer races does not allow for a reduction of MM to GC. The reason is, as seen before, that eliding reuse does not allow to maintain equivalence among invalid pointers. Hence, assertions involving invalid pointers (which raise no SPR) may not have the same outcome in an MM-computation from $\mathscr{W}$ and the corresponding GC-computation from $\mathscr{S}$ which elides reuses. Nevertheless, the relaxation to strong pointer races eases verification efforts. We found that SPR freedom provides an allocating thread with exclusivity [19]. This exclusivity is a write exclusivity rather than a read/write exclusivity as in GC. Exploiting this observation improves the efficiency of existing techniques [19].

In the future we want to establish a reduction for programs using optimistic access patterns. A promising approach is a *mixed semantics* where only a few memory locations can be reused like in MM and the remaining ones cannot like in GC. To guarantee soundness then, one has to show that the aforementioned comparisons involving invalid pointers can be mimicked in the mixed semantics. We believe that this boils down to showing that the program under scrutiny does not suffer from the ABA problem. This can be decided in the simpler mixed semantics.

6 Conclusion

We presented various approaches for reasoning about correctness wrt. to a complicated weak semantics by solely relying on a simpler strong semantics. The approaches exploit properties satisfied by *most but not all* programs. In other word, they facilitate that programs for a certain (application) domain typically adhere to the same principles or programming patterns. Consequently, not all programs can be handled because the usage of the pattern is not enforced, for example, by the programming language. Experiments show two interesting things. First, most programs of interest can be handled despite relying on domain specific assumptions. Second, exploiting those assumptions allows for much more efficient analyses. The stronger semantics can be verified much easier and a soundness check for the reduction can be performed without much overhead too. Altogether, this line of research highly suggests to give up completeness and focus on specific program domains in order to make verification possible where it is highly needed but not yet successful.

References

1. Abdulla, P.A., Haziza, F., Holík, L., Jonsson, B., Rezine, A.: An integrated specification and verification technique for highly concurrent data structures. In: TACAS. LNCS, vol. 7795, pp. 324–338. Springer (2013)
2. Abdulla, P.A., Jonsson, B., Trinh, C.Q.: Automated verification of linearization policies. In: SAS. LNCS, vol. 9837, pp. 61–83. Springer (2016)
3. Adve, S.V., Hill, M.D.: A unified formalization of four shared-memory models. IEEE Trans. Parallel Distrib. Syst. 4(6), 613–624 (1993)
4. Alglave, J.: A shared memory poetics. Ph.D. thesis, Université Paris 7 (2010)
5. Bouajjani, A., Calin, G., Derevenetc, E., Meyer, R.: Lazy TSO reachability. In: FASE. LNCS, vol. 9033, pp. 267–282. Springer (2015)
6. Bouajjani, A., Derevenetc, E., Meyer, R.: Checking robustness against TSO. CoRR abs/1208.6152 (2012)
7. Bouajjani, A., Derevenetc, E., Meyer, R.: Checking and enforcing robustness against TSO. In: ESOP. LNCS, vol. 7792, pp. 533–553. Springer (2013)
8. Bouajjani, A., Meyer, R., Möhlmann, E.: Deciding robustness against total store ordering. In: ICALP (2). LNCS, vol. 6756, pp. 428–440. Springer (2011)
9. Burckhardt, S., Musuvathi, M.: Effective program verification for relaxed memory models. In: CAV. LNCS, vol. 5123, pp. 107–120. Springer (2008)
10. Clarke, E.M.: The birth of model checking. In: 25 Years of Model Checking. LNCS, vol. 5000, pp. 1–26. Springer (2008)
11. Cousot, P., Cousot, R.: Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In: POPL. pp. 238–252. ACM (1977)
12. Cousot, P., Cousot, R.: Systematic design of program analysis frameworks. In: POPL. pp. 269–282. ACM Press (1979)
13. Dijkstra, E.W.: Cooperating Sequential Processes, pp. 65–138. Springer New York (2002)
14. Doherty, S., Groves, L., Luchangco, V., Moir, M.: Formal verification of a practical lock-free queue algorithm. In: FORTE. LNCS, vol. 3235, pp. 97–114. Springer (2004)
15. Elmas, T., Qadeer, S., Sezgin, A., Subasi, O., Tasiran, S.: Simplifying linearizability proofs with reduction and abstraction. In: TACAS. LNCS, vol. 6015, pp. 296–311. Springer (2010)
16. Elmas, T., Qadeer, S., Tasiran, S.: A calculus of atomic actions. In: POPL. pp. 2–15. ACM (2009)

17. Harris, T.L.: A pragmatic implementation of non-blocking linked-lists. In: DISC. LNCS, vol. 2180, pp. 300–314. Springer (2001)
18. Haziza, F., Holík, L., Meyer, R., Wolff, S.: Pointer race freedom. CoRR abs/1511.00184 (2015)
19. Haziza, F., Holík, L., Meyer, R., Wolff, S.: Pointer race freedom. In: VMCAI. LNCS, vol. 9583, pp. 393–412. Springer (2016)
20. Herlihy, M., Shavit, N.: The art of multiprocessor programming. Morgan Kaufmann (2008)
21. Holík, L., Meyer, R., Vojnar, T., Wolff, S.: Effect summaries for thread-modular analysis. CoRR abs/1705.03701 (2017)
22. Holík, L., Meyer, R., Vojnar, T., Wolff, S.: Effect summaries for thread-modular analysis - sound analysis despite an unsound heuristic. In: SAS. LNCS, vol. 10422, pp. 169–191. Springer (2017)
23. Huynh, T.Q., Roychoudhury, A.: A memory model sensitive checker for c#. In: FM. LNCS, vol. 4085, pp. 476–491. Springer (2006)
24. Jonsson, B.: Using refinement calculus techniques to prove linearizability. Formal Asp. Comput. 24(4-6), 537–554 (2012)
25. Lamport, L.: How to make a multiprocessor computer that correctly executes multiprocess programs. IEEE Trans. Computers 28(9), 690–691 (1979)
26. Lamport, L.: A fast mutual exclusion algorithm. ACM Trans. Comput. Syst. 5(1), 1–11 (1987)
27. Michael, M.M.: High performance dynamic lock-free hash tables and list-based sets. In: SPAA. pp. 73–82 (2002)
28. Michael, M.M., Scott, M.L.: Simple, fast, and practical non-blocking and blocking concurrent queue algorithms. In: PODC. pp. 267–275. ACM (1996)
29. Michael, M.M., Scott, M.L.: Nonblocking algorithms and preemption-safe locking on multi-programmed shared memory multiprocessors. J. Parallel Distrib. Comput. 51(1), 1–26 (1998)
30. Park, S., Dill, D.L.: An executable specification, analyzer and verifier for RMO (relaxed memory order). In: SPAA. pp. 34–41 (1995)
31. Peterson, G.L.: Myths about the mutual exclusion problem. Inf. Process. Lett. 12(3), 115–116 (1981)
32. da Rocha Pinto, P., Dinsdale-Young, T., Gardner, P.: Tada: A logic for time and data abstraction. In: ECOOP. LNCS, vol. 8586, pp. 207–231. Springer (2014)
33. Schellhorn, G., Derrick, J., Wehrheim, H.: A sound and complete proof technique for linearizability of concurrent data structures. ACM Trans. Comput. Log. 15(4), 31:1–31:37 (2014)
34. Sewell, P., Sarkar, S., Owens, S., Nardelli, F.Z., Myreen, M.O.: x86-tso: a rigorous and usable programmer's model for x86 multiprocessors. Commun. ACM 53(7), 89–97 (2010)
35. Shasha, D.E., Snir, M.: Efficient and correct execution of parallel programs that share memory. ACM Trans. Program. Lang. Syst. 10(2), 282–312 (1988)
36. Vafeiadis, V.: Rgsep action inference. In: VMCAI. LNCS, vol. 5944, pp. 345–361. Springer (2010)
37. Vafeiadis, V., Parkinson, M.J.: A marriage of rely/guarantee and separation logic. In: CONCUR. LNCS, vol. 4703, pp. 256–271. Springer (2007)
38. Vechev, M.T., Yahav, E.: Deriving linearizable fine-grained concurrent objects. In: PLDI. pp. 125–135. ACM (2008)
39. Yahav, E., Sagiv, S.: Automatically verifying concurrent queue algorithms. Electr. Notes Theor. Comput. Sci. 89(3), 450–463 (2003)