Scheduling Constraint Based Abstraction Refinement for Weak Memory Models

Liangze Yin
Laboratory of Software Engineering for Complex Systems
National University of Defense Technology
Changsha, Hunan, China
yinliangze@163.com

Wei Dong
Laboratory of Software Engineering for Complex Systems
National University of Defense Technology
Changsha, Hunan, China
wdong@nudt.edu.cn

Wanwei Liu
Laboratory of Software Engineering for Complex Systems
National University of Defense Technology
Changsha, Hunan, China
wwliu@nudt.edu.cn

Ji Wang
State Key Laboratory of High Performance Computing
National University of Defense Technology
Changsha, Hunan, China
wj@nudt.edu.cn

ABSTRACT
Scheduling constraint based abstraction refinement (SCAR) is one of the most efficient methods for verifying programs under sequential consistency (SC). However, most multi-processor architectures implement weak memory models (WMMs) in order to improve the performance of a program. Due to the nondeterministic execution of those memory operations by the same thread, the behavior of a program under WMMs is much more complex than that under SC, which significantly increases the verification complexity. This paper elegantly extends the SCAR method to WMMs such as TSO and PSO. To capture the order requirements of an abstraction counterexample under WMMs, we have enriched the event order graph (EOG) of a counterexample such that it is competent for both SC and WMMs. We have also proposed a unified EOG generation method which can always obtain a minimal EOG efficiently. Experimental results on a large set of multi-threaded C programs show promising results of our method. It significantly outperforms state-of-the-art tools, and the time and memory it required to verify a program under TSO and PSO are roughly comparable to that under SC.

CCS CONCEPTS
• Software and its engineering → Software verification and validation;

KEYWORDS
concurrent program verification, weak memory model, scheduling constraint, event order graph

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

ASE ’18, September 3–7, 2018, Montpellier, France
© 2018 Association for Computing Machinery.
ACM ISBN 978-1-4503-5937-5/18/09...
https://doi.org/10.1145/3238147.3238223

1 INTRODUCTION
Sequential consistency (SC) is a simple and commonly assumed memory consistency model for concurrent programs. Most existing work for concurrent program verification focuses on SC. However, to improve the performance of concurrent programs, most multi-processor architectures use weak memory models (WMMs). Programs under WMMs exhibit two kinds of nondeterminism: the inter-thread nondeterminism due to the thread interleaving, and the intra-thread nondeterminism due to the out of order effect on those memory operations by the same thread. Due to the intra-thread nondeterminism, a program under WMMs allows many more paths than that under SC, which significantly increases the complexity of writing and verifying concurrent programs.

Bounded model checking (BMC) [10] is one of the most efficient techniques to alleviate the path explosion problem of concurrent programs. However, a large formula is usually required to give an exact encoding of the program behavior. To avoid the large encoding, we have proposed a scheduling constraint based abstraction refinement (SCAR) method for verification of concurrent programs under SC [31]. The idea is that, observing that the scheduling constraint usually dominates the monolithic encoding, we initially ignore the scheduling constraint and obtain an over-approximation of the original program. If a counterexample is found, further validation of the counterexample is performed and the abstraction is refined if the counterexample is determined to be infeasible. To prune more search space in each iteration, we have proposed a notion of event order graph (EOG) and two graph-based algorithms over EOG for counterexample validation and refinement generation. We have proved that our method is both sound and complete w.r.t. the given loop unwinding depth. Our tool implementing this method has won the gold medal in the ConcurrencySafety category of both SV-COMP 2017 and 2018.
In this paper, we try to extend this method to WMMs such as TSO and PSO. We first analyzed the difference between the monolithic encodings of concurrent programs under SC and WMMs. The only difference is on the execution order of those memory operations by the same thread, which is deterministic under SC but nondeterministic under WMMs. The out of order effect of WMMs allows a following operation to execute before a previous one. In the SCAR method, the execution order of different memory operations is captured by the EOG of an abstraction counterexample. To extend this method to WMMs, we have enriched the EOG by employing a directed graph rather than a line to restrain the intra-thread execution order. An intuitive method to obtain the EOG usually generates many redundant orders, which may significantly increase the overhead of those graph analysis processes. To address this problem, we have devised a unified EOG generation algorithm, which can always obtain the minimal EOG for both SC and WMMs efficiently. Then, we obtained a unified SCAR method which can deal with programs under both SC and different WMMs.

We have implemented our method on top of Yogar-CBMC and evaluated it on the benchmarks in the ConcurrencySafety category of SV-COMP 2018. Experimental results show promising performance of our method. It has successfully verified all examples under TSO and PSO, and performs much better than CBMC, one of the most popular verifier for concurrent program verification. We also observed that our method is not sensitive to different memory models. In our experiments, the time and memory it required to verify a program under WMMs are comparable to that under SC.

The contributions of this paper are listed as follows.

1. This paper has elegantly extended the SCAR method to a unified approach for different memory models, which can deal with programs under both SC and WMMs.
2. This paper has proposed a unified algorithm for EOG generation, which is suitable for different memory models, and can always obtain the minimal EOG efficiently.
3. We have implemented our method on top of Yogar-CBMC. Experimental results on a large number of programs show promising performance of our method.
4. We have verified all examples in the ConcurrencySafety category of SV-COMP 2018 under TSO and PSO. To the best of our knowledge, we are the first to give exact results to all these examples under TSO and PSO.

The rest of this paper is organized as follows. Section 2 introduces the preliminaries. Section 3 reviews the SCAR method under SC. Sections 4 extends this method to WMMs. Section 5 enriches the EOG to WMMs and presents our EOG generation method which can obtain the minimal EOG efficiently. Section 6 provides the experimental results. Section 7 reviews the related work, and Section 8 concludes the paper.

2 PRELIMINARIES

2.1 Multi-Threaded Program

A multi-threaded program consists of multiple concurrent threads. It contains a set of variables which are partitioned into shared variables and local variables. Each thread can read/write both the shared variables and its local variables. We assume each variable access is atomic. This paper focuses on programs based on PThreads, one of the most popular libraries towards multi-threaded programming. It uses pthread_create(&t, &attrb, &f, &args) to create a new thread t, and pthread_join(t, &return) to suspend current thread until thread t terminates.

Given a multi-threaded program, we write V for the set of shared variables. An event e is a read/write access to a shared variable. We use E to denote all events. Each e ∈ E is associated with an element var(e) ∈ V and a literal guard(e), which represent the accessed variable and the guard literal, respectively. An event e can be either a read or a write of the shared variable var(e). To express the execution orders of different events, we associate each event with a unique natural number clk(e). clk(e1) < clk(e2) represents that e1 executes before e2.

The program P determines a partial order ≺pe E × E. Intuitively, e1 ≺pe e2 (or we write (e1, e2) ∈ pe) indicates that “e1 should happen before e2 according to the program order of P.” A read-write link (e1, e2) represents that “e2 reads the value written by e1.” Therefore, var(e1) = var(e2), e1 is a write, e2 is a read, and the value of e2 is equal to that of e1. In addition, there should be no other “write” of var(e1) happening between them.

2.2 Weak Memory Model

SC is the most simple and commonly assumed memory consistency model. It requires the execution order of those events by the same thread (we call it “intra-thread order” in the following) to be consistent with the program order. Consider the simplified Dekker mutual exclusion algorithm shown in Figure 1, which consists of two threads P1 and P2. It contains two shared variables x and y. Both of them are initialized to 0. Under SC, it requires the write of x to execute before the read of y for P1, and the write of y to execute before the read of x for P2. After both P1 and P2 terminate, at least one of r and s must be 1.

Under WMMs, to make full use of hardware and compiler-level optimizations, it allows the intra-thread order to be inconsistent with the program order. That is, for those events by the same thread, it allows a following event to execute before a previous one. We usually call this feature the out of order effect of WMMs. The most widely used Intel x86 and SPARC architectures use the TSO (Total Store Ordering) model [23][28], which relaxes the Write to Read program order, i.e., a following read can execute before a previous write to a different address. The PSO (Partial Store Ordering) model further relaxes the Write to Write program order, i.e., it further allows a following write to execute before a previous write to a different address. The IBM PowerPC model relaxes program order between all operations to different addresses [21]. Thus, a following read or write can execute before a previous read or write to a

\[
\text{Initially } x = y = 0
\]

\[
P1 \quad P2
\]

\[
x := 1 \quad y := 1
\]

\[
r := y \quad s := x
\]

Figure 1: A simplified Dekker algorithm.
different address. We in this paper mainly focus on TSO and PSO. Nevertheless, our method can also be applied to other models.

The out of order effect of WMMs plays a crucial role in improving the program performance. However, programs based on WMMs may exhibit many behaviors that violate the programmer’s intuition. For the example shown in Figure 1, under TSO, the read of $y$ can execute before the write of $x$ for $P1$, and the read of $x$ can execute before the write of $y$ for $P2$. Hence, $r$ and $s$ may both be 0 after $P1$ and $P2$ terminate.

3 THE SCAR METHOD FOR SC

BMC is one of the most applicable techniques to alleviate the path explosion problem of concurrent programs. Instead of explicitly enumerating all thread interleavings, it employs a symbolic representation to encode the verification problem, which is then solved by a SAT/SMT solver. If a positive answer is given, then a satisfying assignment corresponding to a feasible counterexample is acquired. Otherwise, the program is proven safe w.r.t. the given loop unwinding depth.

In BMC, a multi-threaded program under SC is usually encoded as a monolithic encoding $\sigma_{sc} := \phi_{init} \land \rho \land \xi \land \xi_{sc}$, where $\phi_{init}$ is the initial states, $\rho$ encodes each thread in isolation, $\xi$ formulates that "each read of a variable $v$ may read the result of any write of $v$", and $\xi_{sc}$ formulates the scheduling constraint, which defines the order requirements among all events of the program under SC. The $\xi_{sc}$ is constituted of two parts: $\xi_{sc}^{po}$ and $\xi_{sc}^{rf}$, where $\xi_{sc}^{po}$ requires that "the intra-thread order should be consistent with the program order", and $\xi_{sc}^{rf}$ defines that "for any read-write link $(e1, e2)$, there should be no other write of $\text{var}(e1)$ between them".

A problem for BMC of concurrent programs is that the monolithic encoding $\sigma_{sc}$ usually generates a large formula which overwhelms modern constraint solvers. Observing that $\sigma_{sc}$ is usually dominated by the scheduling constraint $\xi_{sc}$, we have proposed a scheduling constraint based abstraction refinement (SCAR) method for verification of multi-threaded programs under SC, which can avoid the large formula $[31]$. Figure 2 presents the overview of the method. Given a multi-threaded C program, it first ignores the scheduling constraint in the encoding. An over-approximated abstraction of the original program is then obtained as $\phi_{sc} := \phi_{init} \land \rho \land \xi_{sc}$. It then adds the abstraction $\phi_{sc}$ and the error states $\phi_{err}$ to the abstraction model. If it is unsatisfiable, then the property is proven safe w.r.t. the given loop unwinding depth. Otherwise, a counterexample of the abstraction is provided. Given that the scheduling constraint $\xi_{sc}$ is ignored in the abstraction, this counterexample may be infeasible and further validation is required. To validate the feasibility of an abstraction counterexample, a notion of event order graph (EOG) is proposed. We have proved that an abstraction counterexample is feasible iff the corresponding EOG is feasible. An intuitive method for EOG validation is constraint solving. If the EOG is infeasible, then the abstraction is refined by exploring the unsatisfiable core. However, this method is not effective for refinement generation. To obtain an effective refinement, we devised two graph-based algorithms over EOG for counterexample validation and refinement generation, in which a small yet effective refinement can always be obtained if the EOG is determined to be infeasible. However, this method is not complete. It can only give an infeasible answer. To make the overall method both efficient and sound, we first adopt the graph-based EOG validation method. If the EOG is determined to be infeasible, the graph-based refinement process is performed to obtain an effective refinement constraint. Otherwise, we employ the constraint-based validation process to further validate the EOG.

Abstraction Counterexample. A counterexample $\pi$ of an abstraction $\phi_i$, is a set of assignments to the variables in $\phi_i \land \phi_{err}$, where $\phi_{err}$ is the error states. It defines a set of events occurring in $\pi$ and the read-write links for each read event. Denote by $E_{\pi} \subseteq B$ the set of events occurring in $\pi$. A counterexample $\pi$ is said to be feasible if a concrete execution of the original program can be constructed from $\pi$. Given that the events occurring in a concrete execution must satisfy the order requirements defined in $\xi_{sc}^{po}$ and $\xi_{sc}^{rf}$, the crucial issue to validate a counterexample $\pi$ is to find a total order $<_{\pi}$ over $E_{\pi}$ s.t. $<_{\pi}$ obeys all the order requirements defined in scheduling constraint.

Figure 2: An overview of the SCAR method $[31]$. 

\[647\]
Event Order Graph. To define the order requirements of a counterexample, two partial orders $\prec_{\pi}$ and $\prec_{\pi}$ are defined on $\mathbb{E}_{\pi} \times \mathbb{E}_{\pi}$. The $\prec_{\pi}$ denotes the order requirements due to $\xi_{\pi}^{po}$. Any order $(e_1, e_2) \in \prec_{\pi}$ requires $e_1$ to execute before $e_2$. The $\prec_{\pi}$ denotes the order requirements due to $\xi_{\pi}^{rf}$. Any order $(e_1, e_2) \in \prec_{\pi}$ represents that $(e_1, e_2)$ is a read-write link in $\pi$, which requires $e_1$ to execute before $e_2$, and there should be no write of $\text{var}(e_1)$ happening between them. An order $\lambda \in \prec_{\pi}$ (resp. $\lambda \in \prec_{\pi}$) is called a program order (resp. read-from order) of $\pi$.

An event order graph (EOG) is a graph that captures all the order requirements of a counterexample. Given a counterexample $\pi$, the EOG $G_{\pi}$ is a triple $(\mathbb{E}_{\pi}, \prec_{\pi}, \prec_{\pi})$, where the nodes are the events in $\mathbb{E}_{\pi}$, and the edges are the orders defined in $\prec_{\pi}$ and $\prec_{\pi}$. Each node corresponds to either a “read” or a “write” of $\pi$, and each edge corresponds to either a program order or a read-from order of $\pi$. For each edge corresponding to a program order $(e_1, e_2) \in \prec_{\pi}$, it requires that $\text{clk}(e_1) < \text{clk}(e_2)$; and for each edge corresponding to a read-from order $(e_1, e_2) \in \prec_{\pi}$, it requires that $\text{clk}(e_1) < \text{clk}(e_2)$, and for any other write of $\text{var}(e_1)$ in $\mathbb{E}_{\pi}$, it requires $\text{clk}(e_2) < \text{clk}(e_1)$ or $\text{clk}(e_2) < \text{clk}(e_1)$. A counterexample $\pi$ is feasible iff there exists a total order $\prec_{\pi}$ over $\mathbb{E}_{\pi}$ s.t. $\prec_{\pi}$ obeys all the order requirements defined in $G_{\pi}$.

EOG based counterexample validation and refinement generation. Any edge $(e_1, e_2)$ of an EOG $G_{\pi}$ requires that $\text{clk}(e_1) < \text{clk}(e_2)$. Hence, an EOG must be infeasible if it contains some cycles. Given that a read-from order $(e_1, e_2) \in \prec_{\pi}$ further requires that no other write of $\text{var}(e_1)$ could happen between $e_1$ and $e_2$, some derived orders deductible from the EOG must exist. Three rules have been proposed in [31] to deduce as many derived orders as possible first. If some cycle exists in the result graph, then the EOG must be infeasible. If a counterexample is determined to be infeasible, one should add some constraints to the abstraction to prevent this counterexample from appearing again in the future search. To prune more search space, the kernel reasons that make the counterexample infeasible are analyzed, and the negations of them are then added to the next abstraction.

Consider the example presented in [31] (We will use it later). The example is shown in Figure 3, where Figure 3(a) is the original program, and Figure 3(b) is the static single assignment (SSA) statement of the program, in which the program variables are renamed such that each variable is assigned only once. Figure 4 is an EOG of a counterexample $\pi$ for this example. In the figures that describe EOGs, the white and gray nodes denote “writes” and “reads” occurring in the corresponding counterexample, respectively. A solid arrow with a triangular head from $e_1$ to $e_2$ represents a program order, which requires $e_1$ to happen before $e_2$. A dashed arrow from $e_1$ to $e_2$ represents a read-write link $(e_1, e_2)$. It requires that: 1) $e_1$ should happen before $e_2$, and 2) no write of $\text{var}(e_1)$ should happen between them. A solid arrow with a hollow head from $e_1$ to $e_2$ represents a derived order, which is derived from existing order requirements. It also requires $e_1$ to happen before $e_2$. Figure 4 shows two cycles, including $C_1: \text{clk}(x_2) < \text{clk}(y_4) < \text{clk}(y_3) < \text{clk}(x_2) < \text{clk}(x_3)$ and $C_2: \text{clk}(x_2) < \text{clk}(x_4) < \text{clk}(y_4) < \text{clk}(x_3) < \text{clk}(x_2)$. Hence, $\pi$ is infeasible. The kernel reasons of these two cycles are then analyzed to refine the abstraction, which contains only two simple CNF clauses.

4 THE SCAR METHOD FOR WMMS

For programs under WMMS, the monolithic encoding can be represented as $\phi_{\text{wmms}} := \phi_{\text{init}} \land \rho \land \zeta \land \xi_{\text{wmms}}$, where $\phi_{\text{init}}, \rho$, and $\zeta$ are the same as those for SC. The $\xi_{\text{wmms}} := \xi_{\text{sc}} \land \xi_{\text{rf}}$ formulates the scheduling constraint under WMMS, which defines the order requirements among all events of a program. Here, the $\xi_{\text{sc}}$ and $\xi_{\text{rf}}$ are the same with $\xi_{\text{sc}}$ and $\xi_{\text{rf}}$, which defines that “for any read-write link $(e_1, e_2)$, there should be no other write of $\text{var}(e_1)$ between them”. The only difference between the encodings under SC and
WMMs is on $ξ_{θ_{sc}}$ and $ξ_{θ_{wmm}}$, which capture the order requirements due to the program order of SC and WMMs, respectively. Under SC, the intra-thread order must be consistent with the program order. But under WMMs, the intra-thread order is relaxed, and a following event may execute before a previous one. Under SC, the intra-thread order is deterministic, and the encoding of $ξ_{θ_{sc}}$ is linear in the number of events. But under WMMs, the intra-thread order is nondeterministic, and the encoding of $ξ_{θ_{wmm}}$ is quadratic in the number of events.

As observed in [31], the monolithic encoding $α_{θ_{sc}}$ is dominated by the scheduling constraint $ξ_{θ_{sc}}$. For most examples in their experiments, $ξ_{θ_{sc}}$ usually accounts for more than 85% of the encoding. For programs under WMMs, the monolithic encoding is same with that under SC, except for $ξ_{θ_{wmm}}$, which is much larger than $ξ_{θ_{sc}}$. Hence, the monolithic encoding $α_{θ_{wmm}}$ is even larger than $α_{θ_{sc}}$, and the scheduling constraint $ξ_{θ_{wmm}}$ accounts for an even larger proportion than that under SC.

To avoid the large formula of $ξ_{θ_{wmm}}$, we will extend the SCAR method to WMMs. The idea is that, similar to the framework shown in Figure 2, we first ignore the scheduling constraint $ξ_{θ_{wmm}}$ from the monolithic encoding $α_{θ_{wmm}}$, and obtain an over-approximated abstraction of the original program, $φ_0 := φ_{init} ∧ p ∧ ξ$. Here, the initial abstraction $φ_0$ is exactly the same as under SC. We then add the abstraction $φ_0$ and the error states $φ_{err}$ to the abstraction model, which is then solved by a constraint solver. If it is unsatisfiable, given that the abstraction is still an over-approximation of the original program under WMMs, the property is proven safe w.r.t. the given loop unwinding depth. Otherwise, a counterexample $π$ of the abstraction is provided, and further validation of the counterexample is required. Given that the abstraction is obtained by excluding the scheduling constraint $ξ_{θ_{wmm}}$, the counterexample $π$ is feasible if there exists a total order $π$ over $π_{θ_{sc}}$ s.t. $π$ obeys all the order requirements defined in $ξ_{θ_{wmm}}$.

Again, we can employ an EOG $G_π$ to capture the order requirements that the events in $π$ must satisfy. If there is a total order of $π_{θ_{sc}}$ which satisfies all the order requirements defined in $G_π$, then $π$ is feasible. To determine the feasibility of an EOG, we first employ the graph-based EOG validation method. Particularly, we deduce as many derived orders as possible by the same rules with that under SC. If there exists some cycle in the result EOG, then we can conclude that no total order of $π_{θ_{sc}}$ can satisfy all the order requirements. In this case, we analyze the kernel reasons leading to these cycles, which are then negated to refine the abstraction. The kernel reason analysis method is exactly the same with that under SC. If no cycle is found, we employ a constraint based method to further decide the feasibility of the EOG. The method is that we encode all the order requirements of the EOG into a constraint formula, which is then solved by a constraint solver. The EOG is feasible iff a satisfiable result is returned. If an unsatisfiable result is obtained, we employ the unsatisfiable core to refine the abstraction.

The overall framework of the SCAR method can be elegantly extended to programs under WMMs. Moreover, the methods for initial abstraction generation, EOG validation, and kernel reason analysis etc. are all exactly the same with that under SC. The only difference is that, given that $ξ_{θ_{wmm}}^{po}$ is different from $ξ_{θ_{sc}}^{po}$, the EOG and the constraint formula encoding order requirements of the EOG should be different from that under SC.

5 A UNIFIED EOG FOR BOTH SC AND WMMS

An EOG is used to capture the order requirements that must be satisfied for those events of a counterexample. Given a counterexample $π$, the EOG $G_π$ is a triple $(π_{θ_{sc}}, π_{θ_{sc}}^0, π_α)$, where the nodes are the events in $π_{θ_{sc}}$, and the edges represent the order requirements defined in $π_{θ_{sc}}^0$ and $π_α$. For programs under WMMs, the $π_{θ_{sc}}$ and $π_α$ are exactly the same with that under SC. The only difference is on $π_{θ_{sc}}^0$. For ease of presentation, for an EOG, we discuss only $π_{θ_{sc}}^0$ in the following.

5.1 The EOG for WMMs

The $π_{θ_{sc}}^0$ defines the order requirements due to the program order. Under SC, it requires the intra-thread order to be consistent with the program order. According to the program order, the intra-thread order of any counterexample is deterministic. It usually forms a line for any thread of a counterexample under SC. Figure 5(a) demonstrates the $π_{θ_{sc}}^0$ under SC for the counterexample shown in Figure 4.

However, for programs under WMMs, the intra-thread order is relaxed. The TSO model allows a following read to execute before a previous write to a different address. Figure 5(b) demonstrates the $π_{θ_{sc}}^0$ under TSO for the counterexample shown in Figure 4. For thread thr1, given that $x_2$ is a write to $x$ and $y_3$ is a read to $y$, we have $y_3$ can execute before $x_2$ and there is no order requirement between $x_2$ and $y_3$. However, given that $y_2$ should execute before $y_3$, and $x_2$ should execute before $m_3$, we have the orders $(y_2, y_3)$ and $(x_2, m_3)$. The order requirements for thread thr2 are the same with those of thread thr1.

The PSO model further allows a following write to execute before a previous write to a different address. Figure 5(c) demonstrates the $π_{θ_{sc}}^0$ under PSO for the same counterexample. For thread thr1, according to the semantics of PSO, we have $m_3$ can execute before $x_2$, and $x_3$ can execute before $m_3$. However, $x_3$ must execute before $x_2$, and $y_3$ must execute before $m_3$ and $x_3$. Moreover, according to the program, $m_2$ should wait until all events in thr1 have finished. The orders shown in Figure 5(c) obey all these requirements. The order requirements for thread thr2 are the same with those of thread thr1.

From the above analysis, the $π_{θ_{sc}}^0$ under WMMs is much more complex than that under SC. The intra-thread order is deterministic and can be represented as a line under SC. However, it is nondeterministic and needs to be represented by a directed graph under WMMs. In fact, the $π_{θ_{sc}}^0$ under SC can be treated as a special case of the $π_{θ_{sc}}^0$ under WMMs.

5.2 Intuitive EOG Generation Method

Given a counterexample $π$, to generate the EOG, we should obtain $π_{θ_{sc}}$, $π_{θ_{sc}}^0$, and $π_α$. Given that both $π_{θ_{sc}}$ and $π_α$ are exactly the same with that under SC, we introduce the method to generate $π_{θ_{sc}}^0$ here. To obtain $π_{θ_{sc}}^0$, an intuitive method is that: for each event pair $(e_1, e_2)$ by the same thread where $e_2$ is a following event of $e_1$, we check whether $e_1$ should execute before $e_2$ according to the WMM effect. If it does, then we add the order $(e_1, e_2)$ to $π_{θ_{sc}}^0$. 
Given a memory model $mm$ and two events $\{e_1, e_2\}$ by the same thread where $e_2$ is a following event of $e_1$, Algorithm 1 presents the procedure $\text{MustBefore}$ which determines whether $e_1$ should execute before $e_2$. The procedure returns true if $mm$ is SC, which represents that no program order is relaxed under SC. It also returns true if $e_1$ and $e_2$ are of the same address, which represents that the program order is relaxed only on events of different addresses. Otherwise (which already implies that $e_1$ and $e_2$ are of different addresses), for TSO, it returns true if $e_1$ is not a write or $e_2$ is not a read; for PSO, it returns true if $e_1$ is not a write or $e_2$ is neither a write nor a read. Under WMMs, a fence statement requires that "any event (resp. after) this statement". The semantics of fence statements has been also considered in the $\text{MustBefore}$ procedure. We allow an event to be a fence too. According to the procedure, it will return true if either $e_1$ or $e_2$ is a fence.

**Algorithm 1: The procedure $\text{MustBefore}$**

```
Input: A memory model $mm$, and two events $\{e_1, e_2\}$ where $e_2$
Output: Return true if $e_1$ should execute before $e_2$ under $mm$, false otherwise.
if $mm = \text{SC}$ then
    return true;
end
if var($e_1$) = var($e_2$) then
    return true;
end
if $mm = \text{TSO}$ then
    return $\neg(\text{IsWrite}(e_1) \land \text{IsRead}(e_2));$
end
if $mm = \text{PSO}$ then
    return $\neg(\text{IsWrite}(e_1) \land (\text{IsRead}(e_2) \lor \text{IsWrite}(e_2)));$
end
```

Consider the $\prec_\pi^0$ of the counterexample shown in Figure 4 under TSO. According to the intuitive EOG generation method, it will generate ten orders for thread $\text{thr1}$, including $\{(y_2, x_2), (y_2, y_3), (y_2, m_3), (y_2, x_3), (x_2, m_3), (x_2, x_3), (y_3, m_3), (y_3, x_3), (m_3, x_3)\}$.

### 5.3 Our EOG Generation Method

A problem for the intuitive EOG generation method is that it may generate many redundant orders in $\prec_\pi^0$

**Definition 5.1.** An order $\lambda := (e_1, e_2)$ is said to be redundant in $\prec_\pi^0$ if $\lambda$ can be deduced by other edges in $\prec_\pi^0$.

In other words, an order $\lambda := (e_1, e_2)$ is redundant if there exists a path from $e_1$ to $e_2$ in $\prec_\pi^0 \setminus \{\lambda\}$. For the above example, the order $(y_2, x_2)$ is redundant because it can be deduced by two other orders $(y_2, x_3)$ and $(x_2, m_3)$. The existence of redundant orders may significantly increase the overhead of the graph-based counterexample validation and refinement generation processes. For example, the number of cycles in the EOG may increase exponentially. However, most of these cycles may be redundant, that is, their kernel reasons may be redundant with other cycles.

To address this problem, we have devised a $\prec_\pi^0$ generation algorithm, which can always obtain the minimal $\prec_\pi^0$ efficiently. This algorithm is suitable for all memory models, including SC, TSO and PSO, etc. Actually, it is a unified algorithm for generating $\prec_\pi^0$ of different memory models. Our algorithm is shown in Algorithm 2. The $\prec_\pi^0$ is initialized to $\emptyset$. We first use the $\text{EventsExtract}$ function to obtain $\mathbb{E}_\pi$. We denote by $\mathbb{E}_{\pi,i}$ the events of thread $P_i$. Here the events in $\mathbb{E}_{\pi,i}$ are stored in the program order. To generate the intra-thread orders of a thread $P_i$, we add the events in $\mathbb{E}_{\pi,i}$ to the EOG one by one in the program order. Let $e_1$ be the events to be added. Then we check whether we should add an order from some of its previous event $e_k$ to $e_1$ reversely. The method is that we first use the $\text{MustBefore}$ procedure shown in Algorithm 1 to determine whether $e_k$ should execute before $e_1$ according to the WMM effect of $mm$. If it does, we employ the $\text{IsRedundant}$ function to check whether the edge is redundant in current $\prec_\pi^0$. If it cannot
be deduced from $\lambda \notin \pi$, then we add the order $\lambda := (e_k, e_j)$ to $\lambda$. After the intra-thread orders of all threads have been added, we use the InterThreadOrderGeneration function to add those program orders among different threads, i.e., the order requirements due to pthread_create’s and pthread_join’s. When the algorithm terminates, we will obtain a minimal $\omega_\pi$ with no redundant orders.

Input: A counterexample $\pi$, and a memory model mm
Output: The program order $\omega_\pi$
$\omega_\pi := \emptyset$;
$E_\pi := \text{EventsExtract}(\pi)$
for each thread $P_i$
  // Generate intra-thread orders of $P_i$
  Let $E_{\pi,i} := E_\pi(P_i)$;
  for $j = 0; j < |E_{\pi,i}|; j + +$
    Let $e_j := E_{\pi,i}[j]$;
    for $k = j + 1; k > 0; k -=$
      Let $e_k := E_{\pi,i}[k]$;
      if MustBefore(mm, $e_k$, $e_j$) then
        Let $\lambda := (e_k, e_j)$;
        if !IsRedundant($\lambda$, $\omega_\pi$) then
          $\omega_\pi := \omega_\pi \cup \{\lambda\}$;
      end
  end
end

$\omega_\pi := \omega_\pi \cup \text{InterThreadOrderGeneration}();$

Algorithm 2: Our $\omega_\pi$ generation algorithm.

The performance of Algorithm 2 significantly depends on the performance of the IsRedundant function. To improve its performance, for each node of the graph, we maintain two bit-vectors to mark the predecessors and successors of each node. Whenever a new order is added to $\omega_\pi$, the two bit-vectors will be updated properly. In this manner, an order $\lambda := (e_1, e_2)$ is redundant in $\omega_\pi$ iff $e_1$ is a predecessor of $e_2$.

We prove that we always obtain a correct and minimal $\omega_\pi$ in Algorithm 2 as follows.

**Theorem 5.2.** For any required order requirement $\lambda \notin \omega_\pi$, $\lambda$ could be deduced from $\omega_\pi$.

**Proof.** In Algorithm 2, if $\lambda$ is required in the EOG but $\lambda \notin \omega_\pi$, then $\lambda$ must be redundant in $\omega_\pi$. According to Definition 5.1, it could be deduced from $\omega_\pi$.

**Theorem 5.3.** For any order requirement $\lambda \in \omega_\pi$, $\lambda$ cannot be deduced from $\omega_\pi$.

**Proof.** Let $\lambda := (e_m, e_n)$. Suppose that $\lambda$ can be deduced from $\omega_\pi \setminus \{\lambda\}$. Then there must exist a path from $e_m$ to $e_n$ in $\omega_\pi \setminus \{\lambda\}$. Suppose that the path is $e_m \rightarrow e_0 \rightarrow \cdots \rightarrow e_k \rightarrow e_n$. Given that all orders in $\omega_\pi$ must start from a previous event to a following event, $e_m$ and all $e_i (0 \leq i \leq k)$ should be predecessors of $e_n$. Hence, $\lambda$ should be redundant in $\omega_\pi$, and the IsRedundant function in Algorithm 2 should return true when $\lambda$ is considered. Hence, $\lambda$ will not be added to $\omega_\pi$, which is contradict with that $\lambda \in \omega_\pi$. Therefore, any order requirement $\lambda \in \omega_\pi$ cannot be deduced from $\omega_\pi \setminus \{\lambda\}$. 

5.4 EOG Encoding

If the graph-based EOG validation process is not sure about the feasibility of the EOG, we need to further validate its feasibility via a constraint-based EOG validation process. The method is that we directly encode all the order requirements defined in the EOG into a constraint formula. The EOG is feasible iff the constraint formula is satisfiable. Again, to avoid the encoding of those redundant orders, only those orders in $\omega_\pi$ should be considered in the encoding, such that the size of the constraint formula can be reduced.

6 EXPERIMENTAL RESULTS

We have implemented our method on top of YOGAR-CBMC [30], which has won the gold medals in the ConcurrencySafety category of SV-COMP 2017 and 2018. We use the 1047 multi-threaded programs of SV-COMP 2018 [25] as our benchmarks. Our tool supports all features of C language in the experiments.

6.1 Benchmark of SV-COMP 2018

The open-source, representative, and reproducible benchmarks of Competition on Software Verification (SV-COMP) have been widely accepted for program verification. Given that these benchmarks are devised for comparison of those state-of-the-art techniques and tools, a significant number of studies on concurrent program verification have performed their experiments on them. The concurrency benchmarks of SV-COMP 2018 include 1047 examples and cover most of the publicly available concurrent C programs that are used for verification. These programs contain hundreds of lines, 4 to 8 threads, complex structure variables with 2-dimensional pointers, and hundreds or even a thousand read/write accesses. With these complex features, these programs are challenging for state-of-the-art concurrency verification techniques and tools. Though many studies have been performed on this benchmark, these studies mainly focus on SC. Existing work for WMMs usually performs their experiments on a small subset of this benchmark [1, 2, 14].

6.2 Experimental Setup

We conduct all of our experiments using a computer with Intel(R) Core (TM) i5 - 4210M CPU 2.60GHz and 12 GB memory. A 600-second time limit is observed.

Our experiments are presented in four parts. We first present the verification results of our method under TSO and PSO. Then we compare our performance of TSO and PSO with that of SC and discuss the overhead due to WMMs. Third, we compare the performance of our method with a popular verifier CBMC [5], which also supports TSO and PSO. Different from our method, it provides an exact encoding of the scheduling constraint for program verification. We use CBMC-5.8, the latest version of CBMC, in our experiments. Lastly, we compare our method with the intuitive EOG generation method and discuss the improvement benefit from our EOG generation method.

The loop unwinding depths for both our method and CBMC are set as that of YOGAR-CBMC [30], which dynamically determines the
unwinding depth through syntax analysis. Particularly, the bound is set to 2 for programs with arrays, and \( n \) if some of the program’s for-loops are upper bounded by a constant \( n \).

### 6.3 Effectiveness and Efficiency

#### 6.3.1 Verification Results for TSO and PSO

The results of the 1047 examples under SC are known to the public, which include 246 true examples and 801 false ones. However, the results of these examples under TSO and PSO remain to be unknown. Given that the order constraints of SC, TSO and PSO are in turn weaker, a false example under SC (resp. TSO) is also false under TSO (resp. PSO), but a true example under SC (resp. TSO) may be false under TSO (resp. PSO).

We perform our method on the overall 1047 examples under SC, TSO, and PSO. TABLE 1 presents the number of true and false results we have obtained under the three memory models.

<table>
<thead>
<tr>
<th></th>
<th>SC</th>
<th>TSO</th>
<th>PSO</th>
</tr>
</thead>
<tbody>
<tr>
<td>true</td>
<td>246</td>
<td>257</td>
<td>125</td>
</tr>
<tr>
<td>false</td>
<td>801</td>
<td>810</td>
<td>922</td>
</tr>
<tr>
<td>total</td>
<td>1047</td>
<td>1047</td>
<td>1047</td>
</tr>
</tbody>
</table>

TABLE 1: Results under SC, TSO and PSO.

Yogar-CBMC has won the gold medal of SV-COMP 2017 and 2018, which can successfully verify all these examples. In our experiments, we have observed that under SC, the EOGs generated by our EOG generation method are exactly the same with those generated by the original Yogar-CBMC. Hence, our results under SC coincide with those of Yogar-CBMC. For TSO and PSO, our method can still solve all examples under these two memory models. It is observed from the experiments that, under TSO, all the 801 false examples under SC are still false, and 9 true examples flip to false. Under PSO, all false examples under TSO are still false, but there are 112 true examples becoming false. From the experimental results, it seems that allowing the reorder of two different writes is more likely to affect the program than allowing the reorder between a write and a following read.

#### 6.3.2 Performance for TSO and PSO

Given that the behaviors of a program under WMMs are much more complex than that under SC, more efforts are usually required in existing work to verify a program under WMMs. To evaluate the efficiency of our method, we compare the performance among SC, TSO and PSO. In our experiments, we have respectively spent 1568, 1937, and 1930 seconds under SC, TSO and PSO to solve all examples. Meanwhile, the memory we have consumed under the three memory models are 44, 43, and 45 GB, respectively. From these results, in our method, the time and memory it required for WMMs are just a little more than that under SC.

Figure 6 further compares the performance of the three memory models for each example, where Figure 6(a) for SC and TSO, and Figure 6(b) for TSO and PSO. In these figures, each point stands for an example. The \( x \)-coordinate of a point denotes the spent time (s) under one memory model (or for one method), and the \( y \)-coordinate denotes that of another model (or method). Among all these examples, there are 451 examples which can be verified faster under TSO than under SC, and there are 338 examples which can be verified faster under PSO than under TSO. There are only 89 (resp. 115) examples which requires half more time from SC (resp. TSO) to TSO (resp. PSO).

Though programs under WMMs have more behaviors, it also provide a higher probability to obtain a feasible counterexample in our method. Even for those true examples, due to the randomness of counterexample generation, the verification under WMMs may also performs better than that under SC. Moreover, in the SCAR method, the core kernel reasons of a cycle is more sensitive to the read-from link literals. Hence, our method is insensitive to the type of memory models.

#### 6.3.3 Comparison with CBMC

We further compare the performance of our method with that of CBMC, which supports TSO and PSO. To verify programs under these two memory models, it adopts the monolithic encoding which gives an exact encoding to the scheduling constraint.

In our experiments, under TSO, CBMC ran error for 8 examples, ran timeout for 29 examples, and gave wrong answers for 3 examples (it turned a false example to true from SC to TSO). To finish all examples, CBMC has spent 32605 seconds and consumed 128 GB of memory in total. Under PSO, CBMC ran error, ran timeout and...
gave wrong answers for those same examples under TSO. It has spent 26738 seconds and consumed 117 GB of memory to finish all examples in total.

Figure 7 further compares the performance of our method with that of CBMC, where Figure 7(a) is for TSO and Figure 7(b) is for PSO. From these two figures, we can see that our method performs better than CBMC for nearly all examples, no matter under TSO or under PSO. The SCAR method performs much better than CBMC under SC. This advantage has been brought into full play to WMMs in our method. For CBMC, given that the encoding for WMMs is much larger than that under SC, its performance is worse under WMMs than that under SC. Hence, the results coincide with our original expectation.

6.3.4 Comparison with the Intuitive EOG Generation Method. One of our major innovation is on the unified EOG generation method which can always obtain the minimal $\pi_0$ with no redundant orders. To evaluate the improvement benefit from this method, we compare the performance of our method with that of the intuitive EOG generation method.

Under TSO and PSO, the intuitive EOG generation method can solve only 138 and 139 of the 1047 examples, respectively. Actually, it cannot solve any example in the pthread_wmm directory, which contains 898 examples. For these examples, there are a large number of shared variable accesses. The intuitive EOG generation method may generate thousands of orders in $\pi_0$, which significantly increases the overhead of the graph-based counterexample validation and refinement generation processes. But with our algorithm, the number of orders contained in $\pi_0$ is reduced to about three hundred. Hence, our method performs much better than the intuitive EOG generation method on these examples.

Figure 8 further compares the performance of our method with that of the intuitive EOG generation method, where Figure 8(a) is for TSO and Figure 8(b) is for PSO. From these two figures, for those examples which generate relatively fewer redundant orders, the performance of the intuitive EOG generation method may be comparable with our improved method. There are some examples on which the intuitive method performs even a little better. The reason is that in the refinement generation process, to reduce the overhead of cycle detection, we have limited the number of detected cycles to 400. Obtaining different cycles may generate different refinement constraints, which will affect the efficiency of the verification. However, for those examples which generate many redundant orders, our improved method runs much faster than the intuitive EOG generation method.
7 RELATED WORK

Verification of concurrent programs under SC has been extensively studied in recent years. The most successful techniques to alleviate the path explosion problem include stateless model checking [2, 7, 12, 18], bounded model checking [5, 15, 19, 26], and abstraction refinement [16, 17, 20, 24], etc. The general method of stateless model checking is to employ partial order reduction (POR) to explore only non-redundant interleavings. There are also some work which reduces the search space by restricting the schedules of the program [8, 29]. Bounded model checking has been considered an efficient technique to address the interleaving problem. In SV-COMP 2017, 16 out of the 18 participants in the ConcurrencySafety category have adopted this technique [9]. However, pure BMC is not efficient enough. Many existing tools combine this method with other techniques [15, 19]. Abstraction refinement is also an important technique for concurrent program verification. Most of existing work employs predicate abstraction to address the data space explosion problem, which uses a finite number of predicates to abstract the program.

For programs under WMMs, given that the path explosion problem is still the main challenge, most of the existing work is based on the techniques for SC. Existing work for verifying programs under WMMs can be divided into two directions. The first direction is to convert a program under a WMM to another program under SC, and then employ the techniques and tools for SC to verify the result program. In the work of [6], to verify the reachability properties of a program $P$ under TSO, it first converts $P$ into a new program $P'$ under SC, such that the reachable sets of $P$ and $P'$ are the same. The work in [4] further extended this approach to PSO, RMO and PowerPC, etc. Precise conversion usually makes $P'$ much more complex than $P$. To improve the verification performance, some work tried to decrease the complexity of the result program by fence insertion. However, fence insertion may loose some precision.

To find a better trade-off between precision and efficiency, how to insert the fences is crucial. The work of [3], [13], and [22] employed CEGAR, predicate abstraction and static analysis methods respectively to help insert fences. In the work of [27], E. Tomasco et al. employed shared memory abstraction (SMA) to convert a program $P$ under TSO and PSO into a new program $P'$ under SC, which is then verified by Lazy-CSeq. Nevertheless, the obtained program $P'$ is usually much more complex than the original program $P$, which makes the verification of programs under WMMs much difficult than that under SC.

Another direction is to extend the techniques of SC to WMMs. The most studied approach is to extend stateless model checking to WMMs. Stateless model checking explicitly explores all possible paths one by one. To alleviate the path explosion problem, it employs POR to reduce the number of explored paths. Hence, the crucial issue of extending stateless model checking to WMMs is to extend POR to WMMs. In the work of [32], N. Zhang et al. relaxed the definition of enabled set and proposed a unified framework which can model both the inter-thread and intra-thread nondeterminism under WMMs. In the work of [1], P. Abdulla et al. proposed the notion of chronological trace for WMMs, which can distinguish the equivalent paths under WMMs. The POR is then extended to WMMs based on this trace. There are also some work extending BMC to WMMs [5, 11]. These work employs a logic formula to represent the behavior of programs under WMMs, which are then solved by a constraint solver. However, encoding the program behavior directly usually requires a large formula, which overwhelms modern constraint solvers. In this paper, we extend the SCAR method to WMMs, which avoids the large monolithic encoding.

8 CONCLUSION

Most multi-processor architectures adopt WMMs to improve the performance of a program. Due to the nondeterminism of the intra-thread execution order, the behavior of a program under WMMs is much more complex than that under SC. In this paper, we have enriched the EOG and elegantly extended the SCAR method to WMMs. To further improve the performance of this method, a unified EOG generation method has been devised which can always obtain the minimal EOG efficiently. We implemented our method on top of YOGAR-CBMC and performed our experiments on the benchmarks in the ConcurrencySafety category of SV-COMP 2018. Experimental results show that our method is not sensitive to memory models, and it performs much better than CBMC, one of the state-of-the-art tools for WMMs.

ACKNOWLEDGMENTS

This work was funded by the National Nature Science Foundation of China (No. 61690203, No. 61532070), and the National key R&D program of China (No. 2017YFB1001802).

REFERENCES


SCAR for Weak Memory Models


