Satisfiability Modulo Ordering Consistency Theory for SC, TSO, and PSO Memory Models

HONGYU FAN, ZHIHANG SUN, and FEI HE, Tsinghua University, China, Key Laboratory for Information System Security, MoE, China, and Beijing National Research Center for Information Science and Technology, China

Automatically verifying multi-threaded programs is difficult because of the vast number of thread interleavings, a problem aggravated by weak memory consistency. Partial orders can help with verification because they can represent many thread interleavings concisely. However, there is no dedicated decision procedure for solving partial-order constraints.

In this article, we propose a novel *ordering consistency theory* for concurrent program verification that is applicable not only under sequential consistency, but also under the TSO and PSO weak memory models. We further develop an efficient theory solver, which checks consistency incrementally, generates minimal conflict clauses, and includes a custom propagation procedure. We have implemented our approach in a tool, called Zord, and have conducted extensive experiments on the *SV-COMP 2020 ConcurrencySafety* benchmarks. Our experimental results show a significant improvement over the state-of-the-art.

CCS Concepts: • **Software and its engineering → Formal software verification**; • **Theory of computation → Logic and verification;**

Additional Key Words and Phrases: Program verification, satisfiability modulo theory, weak memory models, concurrency

ACM Reference format:
Hongyu Fan, Zhihang Sun, and Fei He. 2023. Satisfiability Modulo Ordering Consistency Theory for SC, TSO, and PSO Memory Models. *ACM Trans. Program. Lang. Syst.* 45, 1, Article 6 (March 2023), 37 pages.
https://doi.org/10.1145/3579835

1 INTRODUCTION

Shared-memory multi-threaded programs are commonly used in modern computing systems. The number of interleavings of a concurrent program makes its verification very hard in practice. It is highly desirable to develop techniques to alleviate the execution explosion problem of concurrent program verification.

A *memory consistency model* (for short, memory model) [5] restricts the execution order of shared-memory accesses from different threads. It determines what value(s) a read access can

This work was supported in part by the National Natural Science Foundation of China (No. 62072267 and No. 62021002) and the National Key Research and Development Program of China (No. 2018YFB1308601).

Authors' address: H. Fan, Z. Sun, and F. He (corresponding author), School of Software, Tsinghua University, Beijing, China, Key Laboratory for Information System Security, MoE, Beijing, China, and Beijing National Research Center for Information Science and Technology, Beijing, China; emails: fhy18@mails.tsinghua.edu.cn, sunzh20@mails.tsinghua.edu.cn, hefei@tsinghua.edu.cn.

0164-0925/2023/03-ART6 $15.00
https://doi.org/10.1145/3579835

return. The ***sequential consistency*** **(SC)** model [42] forces memory accesses in each thread to follow the program order of instructions. To make full use of hardware resources and improve the efficiency of multi-threaded program execution, *weak memory models* allow certain memory access orders to be relaxed. Compared to SC, a weak memory model allows more concurrent behaviors and further aggravates the execution explosion problem. This article examines two weak memory models, i.e., **total store order (TSO)** [47] and **partial store order (PSO)** [58].

Bounded model checking **(BMC)** [15, 17] is a verification technique that is particularly efficient in bug-finding and has been widely adopted by most verification tools. BMC sets upper bounds for loops and recursive functions to obtain a bounded program and then uses SMT solvers to verify its correctness. A promising technique for handling multi-threaded programs with BMC is to use *partial orders* to represent the happens-before relation between shared-memory access events [9, 10]. In this way, one can achieve a compact representation of the vast number of interleaving behaviors of multi-threaded programs.

The standard approach (e.g., in References [8–10, 51, 61]) for solving partial order constraints is based on *integer difference logic*. Each event is associated with an integer-valued clock, and event orders are represented as differences among these clock variables. Then, the partial order constraints can be solved by the decision procedure of integer difference logic. There are two problems with this approach. First, it determines a clock value for each event, which goes a little bit too far, because we only care about the events' order, not their exact clock values. Second, there is an important axiom (Axiom 3 in Section 4) in reasoning about multi-threaded programs, which defines the derivation rule for the so-called *from-read* orders. Existing approaches [8–10, 28, 51] encode all possible from-read constraints, irrespective of whether they are actually needed for verification. This method yields numerous from-read constraints, which significantly increases the burden on the solver and worsens its performance.

In this article, we propose a new and novel *ordering consistency* $\mathcal{T}_{ord}$ theory (see Section 4) and elaborate on its theory solver (see Section 5) for multi-threaded program verification. Using this method, we no longer need to specify all possible from-read orders in the encoding formula. One direct benefit is the significant reduction in the size of the encoding formula. Another benefit is the *on-demand* deduction of from-read orders. With a specialized theory propagation procedure (see Section 5.5), a from-read order is derived only when the relevant variables get assigned. In this way, we avoid the generation of massive useless from-read constraints.

We develop an efficient theory solver for $\mathcal{T}_{ord}$ and integrate it into the DPLL(T) framework [26]. Given a partial assignment, the solver judges whether this assignment is consistent with the theory axioms, which can further be reduced to detecting cycles on a so-called *event graph*. In particular, we use an incremental consistency checking algorithm (see Section 5.2) that utilizes the previously computed results and attains better efficiency. We also devise a conflict clause generation algorithm (see Section 5.3) for finding the minimal reasons for inconsistency. The complexity of this algorithm is linear in the number of conflict clauses and the number of edges in the event graph.

Last but not least, inspired by the idea of *unit clause propagation*, we propose a novel technique for theory propagation. We attempt to find the so-called *unit edges* and use these edges to enforce values of some unassigned variables (see Section 5.5.1). With this technique, the decision iterations of DPLL(T) are greatly reduced, and the whole performance is significantly improved.

We have implemented the proposed approach in CBMC [41] and Z3 [21] and conducted experiments on 1,061 benchmarks in the `ConcurrencySafety` category of `SV-COMP 2020`. We compare our approach with state-of-the-art concurrent verification tools, including CBMC [9, 41], Lazy-CSeq [36], Lazy-SMA [54], CPA-Seq [13, 14], and Dartagnan [27], under SC, TSO, and PSO, respectively:

- Under SC,[1] our approach solves 38, 119, and 897 more cases than CBMC, CPA-Seq, and Dartagnan, respectively, and 6 less cases than Lazy-CSeq; counting on both-solved cases, our approach runs 2.44×, 90.04×, 139.47×, and 7.20× faster, consumes 20.8%, 99.6%, 99.0%, and 94.5% less memory, than CBMC, CPA-Seq, Dartagnan, and Lazy-CSeq, respectively.
- Under TSO,[2] our approach solves 47, 925, and 533 more cases than CBMC, Dartagnan, and Lazy-SMA, respectively; counting on both-solved cases, our approach is 2.47×, 174.18×, 4.49× faster, and consumes 31.8%, 98.2%, 92.6% less memory than CBMC, Dartagnan, and Lazy-SMA, respectively.
- Under PSO, our approach solves 50, 890, and 273 more cases than CBMC, Dartagnan, and Lazy-SMA, respectively; counting on both-solved cases, our approach runs 2.44×, 163.43×, 11.29× faster, and consumes 31.0%, 98.4%, 93.2% less memory, than CBMC, Dartagnan, and Lazy-SMA, respectively.

We have also compared our approach with state-of-the-art **stateless model checking (SMC)** tools, namely, Nidhugg [2, 4] and GenMC [40] on nine benchmarks from the Nidhugg suite. Experimental results show that as the program's scale (measured by the number of traces) increases, our approach is superior to these tools in most cases.

In summary, our main contributions are:

- We propose a new *ordering consistency* theory $\mathcal{T}_{ord}$ for multi-threaded program verification under SC, TSO, and PSO memory models.
- We elaborate on an efficient theory solver for $\mathcal{T}_{ord}$, which realizes incremental consistency checking, minimal conflict clause generation, and specialized theory propagation to improve the efficiency of SMT solving.
- We implement our approach in CBMC and Z3. Experimental results on `SV-COMP` concurrent benchmarks demonstrate orders of magnitude improvements of our method over state-of-the-art verification tools.

This article is an extended and revised version of a previous conference paper [32]. Compared to Reference [32], this article makes the following new contributions: First, only the SC memory model is supported in Reference [32]; in this article, we extend our approach to support the TSO and PSO memory models. Second, the $\mathcal{T}_{ord}$ theory in Reference [32] assumes no atomicity constraints. Noting that atomic operations are commonly specified in concurrent programs, this article extends $\mathcal{T}_{ord}$ to support atomicity constraints. This extension leads to a new atomicity relation $\approx$ and adaptations in $\mathcal{T}_{ord}$ and the $\mathcal{T}_{ord}$-solver (see Section 4). Third, this article also reports two new sets of experimental results (see Section 6) for evaluating our approach under TSO and PSO memory models.

The rest of the article is organized as follows: Section 2 introduces some background knowledge. Section 3 demonstrates our symbolic encoding of multi-threaded programs. Section 4 proposes the new $\mathcal{T}_{ord}$ theory. Section 5 develops a theory solver for $\mathcal{T}_{ord}$. We report experimental results in Section 6 and discuss related work in Section 7. Section 8 concludes this article.

2 PRELIMINARIES

2.1 Notions

In *first-order logic*, a *term* is a variable, a constant, or an n-ary function applied to n terms; an *atom* is $\bot$, $\top$, or an n-ary predicate applied to n terms; a *literal* is an *atom* or its negation. A *first-order*

[1]Lazy-SMA does not support SC and is excluded from the experiment under SC.
[2]CPA-Seq and Lazy-CSeq do not support TSO and PSO and are excluded from the corresponding experiments.

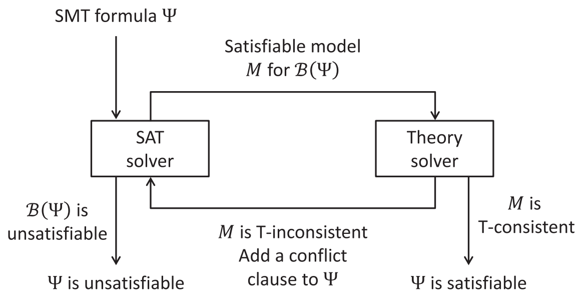

Fig. 1. Flow of DPLL(T).

formula is built from literals using Boolean connectives and quantifiers. A *model* M consists of a non-empty object set $dom(M)$, called the *domain* of M, an assignment that maps each variable to an object in $dom(M)$, and an interpretation for each constant, function and predicate, respectively. A formula Φ is *satisfiable* if there exists a model M, $M \models \Phi$; Φ is *valid* if for any model M, $M \models \Phi$.

A *first-order theory* $\mathcal{T}$ is defined by a *signature* and a set of *axioms*. The *signature* consists of constant symbols, function symbols, and predicate symbols allowed in $\mathcal{T}$; the *axioms* prescribe the intended meanings of these symbols. A $\mathcal{T}$*-model* is a model that satisfies all axioms of $\mathcal{T}$. A formula Φ is $\mathcal{T}$*-satisfiable* if there exists a $\mathcal{T}$-model M so $M \models \Phi$; Φ is $\mathcal{T}$*-valid* if it is satisfied by all $\mathcal{T}$-models.

2.2 Satisfiability Modulo Theory and DPLL(T)

The ***satisfiability modulo theories (SMT)*** problem [11, 21, 22] is a decision problem for formulas in some combination of first-order background theories. A *theory solver* is required for each background theory $\mathcal{T}$, called $\mathcal{T}$-solver, with which the $\mathcal{T}$-satisfiability of any conjunction of literals in $\mathcal{T}$ can be determined.

DPLL(T) is the standard framework for solving SMT instances. It extends the classical DPLL algorithm [20, 45] with dedicated theory solvers. Figure 1 shows a high-level overview of DPLL(T). Given an SMT formula Ψ, DPLL(T) first replaces each atom with a fresh Boolean variable. This process is called *Boolean abstraction*, because the resulting formula, denoted by $\mathcal{B}(\Psi)$, is an over-approximation of the original formula Ψ with respect to satisfiability. The satisfiability of $\mathcal{B}(\Psi)$ can be determined by a SAT solver. If $\mathcal{B}(\Psi)$ is unsatisfiable, then so is Ψ; but the reverse may not hold. If $\mathcal{B}(\Psi)$ is satisfiable and M is the satisfying model returned by the SAT solver, then we need to go ahead to check whether M is consistent with the underlying first-order theories.

A theory solver can be integrated with DPLL(T) in an *online* or *offline* scheme. Let M be the current (partial) assignment to $\mathcal{B}(\Psi)$. In the online scheme, $\mathcal{T}$-solver checks $\mathcal{T}$-consistency of M as long as M is updated (even when M is a partial assignment); in the offline scheme, consistency checking is involved only when M is a satisfying model of $\mathcal{B}(\Psi)$. If M is $\mathcal{T}$-inconsistent, then $\mathcal{T}$-solver attempts to generate a *conflict clause* and adds it to the clause set to prevent the solver from repeating the same inconsistency in the future. A typical theory solver also supports *theory propagation*, which deduces values of unassigned literals by theory axioms. Our method is integrated with DPLL(T) in an *online* scheme.

2.3 Concurrent Execution as Partial Order

A multi-threaded program comprises multiple threads running in parallel. It contains a set of variables that can be divided into *local variables* accessible to a specific thread only and *shared variables* accessible to all threads. Our approach extends the framework of Alglave et al. [9], which models executions of multi-threaded programs using partial order.

An *event* is either a read or a write access to a shared variable and has the following attributes:

- $type(e)$: the type of e, i.e., `W` if e is a write access, and `R` if e is a read access,
- $addr(e)$: the memory address that e accesses,
- $guard(e)$: the guard condition on which e is enabled.

Let $\mathbb{E}$ be the set of all events. There are some relations over events in $\mathbb{E}$. The ***program order*** (***PO***) relation $\prec_{po}$ is a total order of events from the same processor. The *atomicity* relation $\approx$ is an equivalence relation such that $e_1 \approx e_2$ iff e_1 and e_2 are contained in the same atomic operation. The ***write serialization*** (***WS***) relation $\prec_{ws}$ is a total order of writes with the same address. The ***read-from*** (***RF***) relation $\prec_{rf}$ links a write event e_1 (with $type(e_1) = \mathtt{W}$) to a read event e_2 (with $type(e_2) = \mathtt{R}$), so e_2 reads the value written by e_1. Moreover, given a pair of write events e_1, e_2 (with $type(e_1) = type(e_2) = \mathtt{W}$) and a read event e_3 (with $type(e_3) = \mathtt{R}$) so $e_1 \prec_{ws} e_2$ and $e_1 \prec_{rf} e_3$, we know that e_1 happens before e_2, and e_3 reads from e_1. To ensure that e_3 does not read from e_2, e_3 must happen before e_2; we call such relation the ***from-read*** (***FR***) relation $\prec_{fr}$.

A weak memory model allows the order of certain pairs of memory access events to be relaxed. In this article, we focus on ***total store order*** (**TSO**) [47] and ***partial store order*** (**PSO**) [58]: the former relaxes the write-to-read program orders, and the latter further relaxes the write-to-write program orders. Formally, consider a program order $e_1 \prec_{po} e_2$, TSO relaxes this order if $type(e_1) = \mathtt{W}$, $type(e_2) = \mathtt{R}$ and $addr(e_1) \neq addr(e_2)$, PSO relaxes this order if $type(e_1) = \mathtt{W}$ and $addr(e_1) \neq addr(e_2)$. Note that PSO can relax more program orders than TSO. We use $\prec_{ppo}$ to represent the ***preserved program order*** (***PPO***) relation after relaxation. Especially, $\prec_{ppo}$ is identical to $\prec_{po}$ under SC.

A *concurrent execution* of the multi-threaded program can be represented as a set of partial orders over access events. Note that $\prec_{ppo}$ is determined by the program and the architecture, while $\prec_{ws}$, $\prec_{rf}$ and $\prec_{fr}$ are specified by executions. A concurrent execution is *valid* if $\prec_{rf} \cup \prec_{ws} \cup \prec_{fr}$ is consistent with $\prec_{ppo}$ and $\approx$; that is, there is a *linearization* of events on this execution that respects $\prec_{ppo}$ and the accesses contained in each atomic operation are executed consecutively, not interrupted by other accesses.

Lemma 1 ([50]). *The relation $\prec_{rf} \cup \prec_{ws} \cup \prec_{fr}$ is consistent with the partial order $\prec_{ppo}$ and the equivalence relation $\approx$, if and only if:*

(1) $(\prec_{ppo} \cup \prec_{rf} \cup \prec_{ws} \cup \prec_{fr}) \cap \approx$ has no cycles, and
(2) all cycles in $\prec_{ppo} \cup \prec_{rf} \cup \prec_{ws} \cup \prec_{fr} \cup \approx$ are contained in $\approx$.

If there is no equivalence relation in the program, then $\prec_{rf} \cup \prec_{ws} \cup \prec_{fr}$ is consistent with $\prec_{ppo}$ iff $\prec_{ppo} \cup \prec_{rf} \cup \prec_{ws} \cup \prec_{fr}$ has no cycles. An execution is *correct* if it satisfies the correctness condition. An incorrect execution is also called a *counterexample.* A program is *correct* iff it does not contain any valid counterexample.

3 SYMBOLIC ENCODING OF MULTI-THREADED PROGRAMS

In this section, we use a simple example to introduce our symbolic encoding, discuss its differences with other approaches, and establish its correctness.

3.1 Symbolic Encoding

Consider the program in Figure 2(a), which contains three threads, i.e., `main`, t_1, and t_2. Our goal is to verify that m and n cannot be both equal to 1 at the end of the execution.

We first convert the original program to its ***static single assignments (SSA)*** form [19], shown in Figure 2(b), where each occurrence (no matter write or read) of each shared variable is replaced

```
int x = 0, y = 0, m = 0, n = 0;
void* thr1(void* arg) {
   if(x == 1)  m = 1;
   else m = x;
   y = x + 1;
}
void* thr2(void* arg) {
   if(y == 1) n = 1;
   else n = y;
   x = y + 1;
}
int main() {
   pthread_t t1, t2;
   pthread_create(&t1, 0, thr1, 0);
   pthread_create(&t2, 0, thr2, 0);
   pthread_join(t1, 0);
   pthread_join(t2, 0);
   assert(!(m == 1 && n == 1));
}
```

(a) The original program

```
int x1 = 0, y1 = 0, m1 = 0, n1 = 0;
void* thr1(void* arg) {
   if(x2 == 1)  m3 = 1;
   else m4 = x3;
    y2  = x4 + 1;
}
void* thr2(void* arg) {
    if(y3 == 1)  n3 = 1;
    else n4 = y4;
    x5 = y5 + 1;
}
int main() {
    pthread_t t1, t2;
    pthread_create(&t1, 0, thr1, 0);
    pthread_create(&t2, 0, thr2, 0);
    pthread_join(t1, 0);
    pthread_join(t2, 0);
    assert(!(m2 == 1 && n2 == 1));
}
```

(b) The SSA form

Fig. 2. A three-threaded program.

with a fresh copy of this variable. A similar SSA transformation procedure is adopted in References [9, 51, 61].

SSA Variables and Access Events. Given an SSA variable x_i, we write $(\!|x_i|\!)$ for its corresponding access event. Especially, we write $(\!|x_i|\!)^w$ for a write access and $(\!|x_i|\!)^r$ for a read access. With respect to the attributes, we have $type((\!|x_i|\!)^w) = \mathtt{W}$, $type((\!|x_i|\!)^r) = \mathtt{R}$, and $addr((\!|x_i|\!)^w) = addr((\!|x_i|\!)^r) = x$.

Considering x in the program (Figure 2(a)), there are five accesses to this variable. Five fresh variables, i.e., x_1, x_2, x_3, x_4, x_5, are introduced in the SSA form (Figure 2(b)). Note that x_1, x_5 represent write accesses and x_2, x_3, x_4 represent read accesses; their corresponding events are represented as $(\!|x_1|\!)^w$, $(\!|x_2|\!)^r$, $(\!|x_3|\!)^r$, $(\!|x_4|\!)^r$, and $(\!|x_5|\!)^w$, respectively.

Value Assignment Encoding. Value assignments of variables in each thread can be encoded by directly interpreting SSA statements. The encoding $\rho_{va}^{t_1}$ of thread t_1's value assignment is:

$$(x_2 = 1 \rightarrow m_3 = 1) \wedge (\neg(x_2 = 1) \rightarrow m_4 = x_3) \wedge (y_2 = x_4 + 1).$$

In a similar way, we get encodings $\rho_{va}^{t_2}$ and ρ_{va}^{main} for value assignments of threads t_2 and `main`, respectively. The value assignment encoding of the whole program is:

$$\rho_{va} := \rho_{va}^{t_1} \wedge \rho_{va}^{t_2} \wedge \rho_{va}^{main}.$$

Error Condition. We use ρ_{err} to encode the error condition of the program. The error condition of the example program is

$$\rho_{err} := (m_2 = 1) \wedge (n_2 = 1).$$

Note that $\rho_{va} \wedge \rho_{err}$ is not sufficient for verifying the program's correctness. A satisfying model of $\rho_{va} \wedge \rho_{err}$ does not necessarily represent a *valid* execution.

Considering the example program in Figure 2, a satisfying model of $\rho_{va} \wedge \rho_{err}$ is:

$$\{x_1 \mapsto 0, x_2 \mapsto 1, m_3 \mapsto 1, x_4 \mapsto 0, y_2 \mapsto 1, y_3 \mapsto 1, n_3 \mapsto 1, y_5 \mapsto 0, x_5 \mapsto 1, m_2 \mapsto 1, \ldots\}.$$

The execution corresponding to this model is, however, invalid. Let us consider the variable x. Recall that $(\!|x_1|\!)^w$ and $(\!|x_5|\!)^w$ are write accesses, $(\!|x_2|\!)^r$ and $(\!|x_4|\!)^r$ are read accesses. By $x_1 = x_4 \neq x_5$, $(\!|x_4|\!)^r$ must read from $(\!|x_1|\!)^w$, and $(\!|x_4|\!)^r$ must happen before $(\!|x_5|\!)^w$ (otherwise, $(\!|x_4|\!)^r$ should read from $(\!|x_5|\!)^w$ instead). Then, $(\!|x_2|\!)^r$ should also happen before $(\!|x_5|\!)^w$ (since $(\!|x_2|\!)^r$ happens before $(\!|x_4|\!)^r$), and therefore should also read from $(\!|x_1|\!)^w$, which contradicts the fact that $x_1 \neq x_2$.

The main reason is that $\rho_{va} \wedge \rho_{err}$ does not restrict the order of memory accesses. In the following, we formulate order constraints of concurrent executions.

Program Order Constraints. We use ρ_{po} to encode the **program order (*PO*)** constraints, which represents the natural order of access events in each thread. The program orders of thread t_1, t_2, and *main* are

$$\rho_{po}^{t_1} := (\!|x_2|\!)^r \prec_{po} (\!|m_3|\!)^w \prec_{po} (\!|x_3|\!)^r \prec_{po} (\!|m_4|\!)^w \prec_{po} (\!|x_4|\!)^r \prec_{po} (\!|y_2|\!)^w,$$
$$\rho_{po}^{t_2} := (\!|y_3|\!)^r \prec_{po} (\!|n_3|\!)^w \prec_{po} (\!|y_4|\!)^r \prec_{po} (\!|n_4|\!)^w \prec_{po} (\!|y_5|\!)^r \prec_{po} (\!|x_5|\!)^w,$$
$$\rho_{po}^{main} := (\!|x_1|\!)^w \prec_{po} (\!|y_1|\!)^w \prec_{po} (\!|m_1|\!)^w \prec_{po} (\!|n_1|\!)^w \prec_{po} (\!|m_2|\!)^r \prec_{po} (\!|n_2|\!)^r.$$

Moreover, since t_1 and t_2 are child threads of `main`, all events in t_1 and t_2 should happen between the invocations to `pthread_create` and `pthread_join`, respectively. As a result, we have the following program order constraint:

$$\rho_{po}^{spawn} := (\!|n_1|\!)^w \prec_{po} (\!|x_2|\!)^r \wedge (\!|n_1|\!)^w \prec_{po} (\!|y_3|\!)^r \wedge (\!|y_2|\!)^w \prec_{po} (\!|m_2|\!)^r \wedge (\!|x_5|\!)^w \prec_{po} (\!|m_2|\!)^r.$$

Let ρ_{po} be the conjunction of the above program order constraints, i.e.,

$$\rho_{po} := \rho_{po}^{t_1} \wedge \rho_{po}^{t_2} \wedge \rho_{po}^{main} \wedge \rho_{po}^{spawn}.$$

Note that some program orders are relaxed under weak memory models. Taking thread t_1 as an example, TSO relaxes $(\!|m_3|\!)^w \prec_{po} (\!|x_3|\!)^r$ and $(\!|m_4|\!)^w \prec_{po} (\!|x_4|\!)^r$ (write-to-read program orders), and PSO relaxes these two program orders plus $(\!|m_4|\!)^w \prec_{po} (\!|y_2|\!)^w$ (write-to-write program order). In the following, we use ρ_{ppo} to represent the ***preserved program order* (*PPO*)** constraint. Note that no program order is relaxed under SC, thus $\rho_{ppo} \equiv \rho_{po}$ for SC.

Atomicity Constraints. We use $\rho_{\approx}$ to represent the atomicity constraints. Intuitively, $e_1 \approx e_2$ means that e_1 and e_2 are in the same atomic operation, so they are executed indivisibly, not interrupted by other accesses. There is no atomic operation specified in the example program in Figure 2 (to simplify other discussions in this article). Assuming that the last statement in thread t_1, i.e., $y_2 = x_4 + 1$, is declared as an atomic operation,[3] we have $(\!|x_4|\!)^r \approx (\!|y_2|\!)^w$.

Read-from Variables and Constraints. Note that a read event $(\!|x_i|\!)^r$ reads a value written by a write event to the same address. Let $\pi((\!|x_i|\!)^r)$ be the set of write accesses that $(\!|x_i|\!)^r$ may read from. Because of thread interactions, $\pi((\!|x_i|\!)^r)$ may contain write accesses in other threads. Consider the read event $(\!|x_2|\!)^r$ in the example program:

$$\pi((\!|x_2|\!)^r) = \{(\!|x_1|\!)^w, (\!|x_5|\!)^w\}.$$

For each write event $(\!|x_j|\!)^w \in \pi((\!|x_i|\!)^r)$, we define a Boolean variable $rf_{j,i}^x$, called a ***read-from* (*RF*)** variable, to specify whether $(\!|x_i|\!)^r$ reads its value from $(\!|x_j|\!)^w$. For each *RF* variable, we have the following constraints:

- *RF-Val constraint*: if $rf_{j,i}^x$ is *true*, $(\!|x_i|\!)^r$ and $(\!|x_j|\!)^w$ are enabled and their values are equal, i.e.,

$$rf_{j,i}^x \rightarrow guard((\!|x_i|\!)^r) \wedge guard((\!|x_j|\!)^w) \wedge (x_i = x_j);$$

[3]SV-COMP benchmarks use `atomic_begin` and `atomic_end` to declare atomic operations.

- *RF-Ord constraint*: if $rf^x_{j,i}$ is *true*, the write event $(\!|x_j|\!)^w$ must happen before the read event $(\!|x_i|\!)^r$, i.e.,
$$rf^x_{j,i} \rightarrow (\!|x_j|\!)^w \prec_{rf} (\!|x_i|\!)^r;$$
- *RF-Some constraint*: if the read event $(\!|x_i|\!)^r$ is enabled, it must obtain its value from a certain write event in $\pi((\!|x_i|\!)^r)$, i.e.,
$$guard((\!|x_i|\!)^r) \rightarrow \bigvee_{(\!|x_j|\!)^w \in \pi((\!|x_i|\!)^r)} rf^x_{j,i}.$$

In the following, we use $\rho_{rf\text{-}val}$, $\rho_{rf\text{-}ord}$, and $\rho_{rf\text{-}some}$ to represent the conjunctions of *RF-Val*, *RF-Ord*, and *RF-Some* constraints over all *RF* variables, respectively. Considering the read event $(\!|x_2|\!)^r$ in the example program, we have:

$$\begin{aligned}
\rho_{rf\text{-}val} &:= (rf^x_{1,2} \rightarrow (x_2 = x_1)) \wedge (rf^x_{5,2} \rightarrow (x_2 = x_5)),\\
\rho_{rf\text{-}ord} &:= (rf^x_{1,2} \rightarrow (\!|x_1|\!)^w \prec_{rf} (\!|x_2|\!)^r) \wedge (rf^x_{5,2} \rightarrow (\!|x_5|\!)^w \prec_{rf} (\!|x_2|\!)^r),\\
\rho_{rf\text{-}some} &:= rf_{1,2} \vee rf_{5,2}.
\end{aligned}$$

Write-serialization Variables and Constraints. For each variable x, let $\gamma(x)$ be the set of write accesses to x. We need to determine a total order among all enabled write accesses in $\gamma(x)$. To this end, for each pair of write accesses $(\!|x_i|\!)^w$, $(\!|x_j|\!)^w$ in $\gamma(x)$, we define a Boolean variable $ws^x_{i,j}$, called a ***write-serialization* (*WS*)** variable, to represent whether $(\!|x_i|\!)^w$ happens before $(\!|x_j|\!)^w$.

For each *WS* variable, we have the following constraints:

- *WS-Cond constraint*: if $ws^x_{i,j}$ is *true*, then both $(\!|x_i|\!)^w$ and $(\!|x_j|\!)^w$ are enabled, i.e.,
$$ws^x_{i,j} \rightarrow guard((\!|x_i|\!)^w) \wedge guard((\!|x_j|\!)^w);$$
- *WS-Ord constraint*: if $ws^x_{i,j}$ is *true*, then the write event $(\!|x_i|\!)^w$ must happen before $(\!|x_j|\!)^w$, i.e.,
$$ws^x_{i,j} \rightarrow (\!|x_i|\!)^w \prec_{ws} (\!|x_j|\!)^w;$$
- *WS-Some constraint*: if $(\!|x_i|\!)^w$ and $(\!|x_j|\!)^w$ are enabled, then one must happen before the other, i.e.,
$$guard((\!|x_i|\!)^w) \wedge guard((\!|x_j|\!)^w) \rightarrow ws^x_{i,j} \vee ws^x_{j,i}.$$

In the following, we use $\rho_{ws\text{-}cond}$, $\rho_{ws\text{-}ord}$, and $\rho_{ws\text{-}some}$ to represent the conjunctions of *WS-Cond*, *WS-Ord*, and *WS-Some* constraints over all *WS* variables, respectively. Considering two write events $(\!|m_1|\!)^w$ and $(\!|m_3|\!)^w$ in the example program, we have:

$$\begin{aligned}
\rho_{ws\text{-}cond} &:= (ws^m_{1,3} \rightarrow (x_2 = 1)) \wedge (ws^m_{3,1} \rightarrow (x_2 = 1)),\\
\rho_{ws\text{-}ord} &:= (ws^m_{1,3} \rightarrow (\!|m_1|\!)^w \prec_{ws} (\!|m_3|\!)^w) \wedge (ws^m_{3,1} \rightarrow (\!|m_3|\!)^w \prec_{ws} (\!|m_1|\!)^w),\\
\rho_{ws\text{-}some} &:= (x_2 = 1) \rightarrow ws^m_{1,3} \vee ws^m_{3,1}.
\end{aligned}$$

From-read Constraints. Considering one read access $(\!|x_i|\!)^r$ and two write accesses $(\!|x_j|\!)^w$, $(\!|x_k|\!)^w$ to the same variable x, if $(\!|x_j|\!)^w$ happens before $(\!|x_k|\!)^w$ and $(\!|x_i|\!)^r$ reads from $(\!|x_j|\!)^w$, then $(\!|x_i|\!)^r$ must also happen before $(\!|x_k|\!)^w$; otherwise, $(\!|x_k|\!)^w$ is closer than $(\!|x_j|\!)^w$ to $(\!|x_i|\!)^r$, and $(\!|x_i|\!)^r$ should read from $(\!|x_k|\!)^w$ instead of $(\!|x_j|\!)^w$. Formally, this rule can be formulated as the following ***from-read* (*FR*)** constraint:
$$rf^x_{j,i} \wedge ws^x_{j,k} \rightarrow (\!|x_i|\!)^r \prec_{fr} (\!|x_k|\!)^w.$$
Let ρ_{fr} denote the conjunction of all *FR* constraints.

Most existing techniques [9, 51, 56] for concurrent program verification include all *from-read* constraints in their encoding formulas. This is a safe choice to ensure the SMT encoding's correctness. However, it is not practical. For each *FR* constraint $rf^x_{j,i} \wedge ws^x_{j,k} \rightarrow (\!|x_i|\!)^r \prec_{fr} (\!|x_k|\!)^w$, only when both $rf^x_{j,i}$ and $ws^x_{j,k}$ are evaluated *true* do we need to consider $(\!|x_i|\!)^r \prec_{fr} (\!|x_k|\!)^w$. At the beginning of SMT solving, all *RF* and *WS* variables are unassigned—no *FR* orders need to be considered. For most of the time, only a small portion of these *FR* constraints take effect. Maintaining such a large set of (unnecessary for most of the time) constraints is expensive for the SMT solver. As a result, the efficiency of SMT solving degenerates (see Section 6.3 for more details).

The Whole Encoding Formula. The whole encoding formula for a program is:

$$\Psi := \Phi_{ssa} \wedge \Phi_{ord}, \tag{1}$$

where

$$\Phi_{ssa} = \rho_{va} \wedge \rho_{err} \wedge \rho_{rf\text{-}val} \wedge \rho_{rf\text{-}some} \wedge \rho_{ws\text{-}cond} \wedge \rho_{ws\text{-}some} \tag{2}$$

represents the data and control flow of the program, and

$$\Phi_{ord} = \rho_{ppo} \wedge \rho_{\approx} \wedge \rho_{ws\text{-}ord} \wedge \rho_{rf\text{-}ord} \tag{3}$$

represents the ordering constraints[4] of the program.

Note that ρ_{fr} is excluded from our encoding formula. Instead of adding all *FR* constraints into the SMT formula, we prefer adding them during SMT solving in an "*online*" schema—an *FR* order is derived and activated only when the corresponding *RF* and *WS* variables are evaluated *true*.

Let X_{ssa}, X_{rf}, and X_{ws} be the sets of SSA, *RF*, and *WS* variables, respectively. *RF* and *WS* variables are also called *ordering variables*. The formula Φ_{ssa} is over $X_{ssa} \cup X_{rf} \cup X_{ws}$, and Φ_{ord} is over $X_{rf} \cup X_{ws}$. Actually, Φ_{ord} is a "pure" formula that contains only ordering variables and ordering literals; Φ_{ssa} is a formula that does not include any ordering literal. To decide the satisfiability of Φ_{ssa}, we can use any existing solver that supports a sufficiently rich fragment of first-order logic. To decide the satisfiability of Φ_{ord}, we intend to develop a dedicated theory solver.

3.2 Constraints Generation

In this section, we detail the algorithms for generating *PPO*, *atomicity*, *RF*, and *WS* constraints under various memory models, respectively.

3.2.1 Preserved Program Order and Atomicity Constraints. Algorithm 1 shows the pseudo code for generating the *preserved program order* ρ_{ppo} and *atomicity* constraints $\rho_{\approx}$. The algorithm starts to process each thread P_i from line 2. Let $\mathbb{E}_i$ be the list of events in thread P_i, sorted by their occurrences in P_i (line 3). The key challenge here is to avoid encoding transitive closure to obtain a minimal number of constraints. More specifically, an order (e_i, e_j) is added to $\prec_{ppo}$ only if this order cannot be implied by the transitive closure of the current $\prec_{ppo}$, i.e., there does not exist an event e_k such that both (e_i, e_k) and (e_k, e_j) are in the current $\prec_{ppo}$. To this end, we define a set *succ_set* (initially $\{e\}$) for each event e in $\mathbb{E}_i$ (line 4) to keep all successor events of e in the current $\prec_{ppo}$.

Algorithm 1 traverses all events in $\mathbb{E}_i$ in reverse order (line 5); and from line 6, the inner loop traverses all subsequent events e' of e in the list $\mathbb{E}_i$ from near to far. The order (e, e') must be in the transitive closure of $\prec_{po}$. If $e \prec_{po} e'$ is preserved under *mm* (line 7), then we need go further to check whether e' is already in the *succ_set* of e (line 8)—if it is, then the order (e, e') is implied by the current ρ_{ppo}, i.e., there exists another event e'' such that both $e \prec_{ppo} e''$ and $e'' \prec_{ppo} e'$ are in the current ρ_{ppo} (note that the *succ_set* of e'' should be updated before the outer loop traverses e); otherwise, $e \prec_{ppo} e'$ is explicitly added to ρ_{ppo} (line 9), and the *succ_set* of e should be extended with *succ_set* of e' (line 10). Moreover, if e and e' belong to the same atomic operation, then $e \approx e'$ is added to $\rho_{\approx}$ (line 12). Finally, each child thread is spawned by `thread_create` and stopped by `thread_join`. All events of this child thread must happen after the call of `thread_create` and

[4]The atomicity constraints can be regarded as bi-directed ordering constraints.

ALGORITHM 1: Generation of *preserved program order* and *atomicity* constraints

```
Data: a memory model $mm$
Output: the preserved program order constraint $\rho_{ppo}$ and the atomicity constraint $\rho_{\approx}$
$\rho_{ppo} \leftarrow \top, \rho_{\approx} \leftarrow \top$;
foreach thread $P_i$ do
    Let $\mathbb{E}_i$ be the list of events in thread $P_i$, sorted by their occurrences in $P_i$;
    foreach $e \in \mathbb{E}_i$ do $e.succ\_set \leftarrow \{e\}$ ;
    foreach $e \in \mathbb{E}_i$ from last to first do                    // reverse traversal
        foreach subsequent $e'$ of $e$ in $\mathbb{E}_i$ from near to far do    // forward traversal
            if $Is\_Preserved(e, e', mm)$ then
                if $e' \notin e.succ\_set$ then
                    $\rho_{ppo} \leftarrow \rho_{ppo} \wedge (e \prec_{ppo} e')$;
                    $e.succ\_set \leftarrow e.succ\_set \cup e'.succ\_set$;
                    if $e, e'$ in the same atomic operation then
                        $\rho_{\approx} \leftarrow \rho_{\approx} \wedge (e \approx e')$;
$\rho_{ppo} \leftarrow \rho_{ppo} \cup Interthread\_Order\_Generation()$;
return $\rho_{ppo}$, $\rho_{\approx}$;
```

ALGORITHM 2: *Is_Preserved*(e, e', mm)

```
Input: two events $e, e'$ and a memory model $mm$
Output: $true$ if the order $(e, e')$ is preserved under $mm$, and $false$ otherwise
if $addr(e) \neq addr(e')$ and $e \not\approx e'$ and $(e, e')$ is not preserved by fences then
    if $mm = TSO \wedge type(e) = W \wedge type(e') = R$ then
        return false;
    if $mm = PSO \wedge type(e) = W$ then
        return false;
return $true$;
```

before the call of `thread_join`; we use `Interthread_Order_Generation` (line 13) to encode such ordering constraints between the `main` and child threads.

Algorithm 2 determines whether the program order $e \prec_{po} e'$ is preserved under the memory model *mm*. If $addr(e) = addr(e')$, i.e., they access the same memory address; or $e \approx e'$, i.e., they belong to the same atomic operation; or (e, e') is preserved by *fences* [9],[5] then this program order must be preserved. There are only two situations that the program order $e \prec_{po} e'$ are relaxed: (1) $type(e) = \mathtt{W}$ and $tyep(e') = \mathtt{R}$ (i.e., write-to-read program order) under TSO (lines 2 and 3), and (2) $type(e) = \mathtt{W}$ (i.e., write-to-read/write program order) under PSO (lines 4 and 5).

The key challenge of Algorithm 1 is to avoid redundant transitive closure of $\prec_{ppo}$ to obtain a small number of constraints. Let ρ_{ppo} be the output of Algorithm 1. We first prove that ρ_{ppo} is *sufficient* for representing all orders in $\prec_{ppo}$.

Lemma 2. *$\forall (e, e') \in \prec_{ppo}$, it can be deduced from ρ_{ppo}.*

Proof. By $e \prec_{ppo} e'$, the order (e, e') is preserved under the memory model. If $e' \notin e.succ\_set$, by line 9 of Algorithm 1, the order (e, e') is explicitly encoded in ρ_{ppo}, the lemma holds for this case. If $e' \in e.succ\_set$, then we prove that $e \prec_{ppo} e'$ can be deduced from ρ_{ppo} by induction:

[5] A concurrent program can use *fence* instructions to prevent non-SC behaviors.

- Suppose there is only one pair of events $(e_1, e_2) \in \prec_{ppo}$, $e_1 \prec_{ppo} e_2$ must be in ρ_{ppo} (base case).
- Suppose there is a set of events $e_1, e_2, \ldots, e_n$ ($n > 2$) following their occurrence in P_i and $\forall e_i, i \in 2, \ldots, n.(e_1, e_i) \in \prec_{ppo}$ and $e_i \in e_1.succ\_set$, it can be deduced from ρ_{ppo} (induction hypothesis).
- Then, for *PO*-successor of e_n, named e_{n+1}, s.t. $(e_1, e_{n+1}) \in \prec_{ppo}$ and $e_{n+1} \in e_1.succ\_set$, we show that $(e_1, e_{n+1}) \in \prec_{ppo}$ can be deduced from ρ_{ppo}. If $e_1 \prec_{ppo} e_{n+1}$ is in ρ_{ppo}, then the inductive step trivially holds. Otherwise, since $\bigcup_{i=2}^{n} e_i.succ\_set \subseteq e_1.succ\_set$, let j be the maximal i s.t. $e_{n+1} \in e_j.succ\_set$, we have the *PPO* constraint $e_j \prec_{ppo} e_{n+1}$ in ρ_{ppo}. Meanwhile, by induction hypothesis, there is a set of *PPO* constraints in ρ_{ppo} (abbreviated as *S*), which implies $(e_1, e_j) \in \prec_{ppo}$ using transitivity. Therefore, $S \cup \{e_j \prec_{ppo} e_{n+1}\}$ deduces $(e_1, e_{n+1}) \in \prec_{ppo}$ (inductive step).

Therefore, the above lemma holds. □

Next, we prove that ρ_{ppo} is *minimal* for representing $\prec_{ppo}$. For simplicity, we write $c_{e,e'}$ for an ordering constraint between e and e', ρ_{ppo} can be regarded as a set of such constraints.

Lemma 3. *$\forall c_{e,e'} \in \rho_{ppo}$, $c_{e,e'}$ cannot be deduced from $\rho_{ppo} \backslash \{c_{e,e'}\}$.*

Proof. We prove this lemma by contradiction. Assume $e \prec_{ppo} e'$ can be derived from $\rho_{ppo} \backslash \{c_{e,e'}\}$, and let $c_{e_1,e_2}, c_{e_2,e_3}, \ldots, c_{e_{n-1},e_n}$ where $e_1 = e$, $e_n = e'$, is the set of constraints in $\rho_{ppo} \backslash \{c_{e,e'}\}$ that form the derive path $e_1 \prec_{ppo} e_2 \ldots \prec_{ppo} e_n$. By line 10 of Algorithm 1, $e_{i+1}.succ\_set$ is integrated into $e_i.succ\_set$ for $i \in 1, 2, \ldots, n-1$. Therefore, e_n is in $e_1.succ\_set$ and Algorithm 1 skips to encode $c_{e,e'}$, which contradicts to the premise of $c_{e,e'} \in \rho_{ppo}$. □

Let $\rho_{\approx}$ be another output of Algorithm 1. It can be concluded from Algorithm 1 that atomicity constraints are generated for each pair of *PO*-adjacent events in the same atomic operation. The following lemma shows that $\rho_{\approx}$ is *sufficient* and *minimal* for representing all atomicity orders in $\approx$. The proof is trivial and is omitted.

Lemma 4. *$\forall (e, e') \in \approx$, it can be deduced from $\rho_{\approx}$; $\forall c_{e,e'} \in \rho_{\approx}$, it cannot be deduced from $\rho_{\approx} \backslash \{c_{e,e'}\}$.*

3.2.2 Read-from and Write-serialization Constraints. For each shared variable x, we first obtain the set *reads* (or *writes*) of all read (or write) accesses to x. Basically, a read r in *reads* may get its value from any write w in *writes*, except when $r \prec_{po} w$ (since r cannot read from a later issued write). We then generate *read-from* constraints for all such pairs of read and write events.

Moreover, for each subset $\{w_1, w_2\} \subseteq$ *writes*, either $w_1 \prec_{ws} w_2$, or $w_2 \prec_{ws} w_1$. Especially, if w_1, w_2 belong to the same thread and $w_1 \prec_{po} w_2$, we use a Boolean variable *ws* to imply $w_1 \prec_{ws} w_2$; meanwhile, $w_2 \prec_{ws} w_1$ can never hold (otherwise, there forms a cycle $w_1 \prec_{po} w_2 \prec_{ws} w_1$). If w_1, w_2 are from different threads, then we use two Boolean variables *ws* and *ws'* for representing these two cases, respectively. We generate *write-serialization* constraints with these *WS* variables.

3.3 Comparison to Other Approaches

Our encoding is built on References [9, 10, 51, 56, 61]. Compared to their encoding formulas, the most significant difference is that our encoding does not include *FR* constraints, which has already been discussed in the preceding section (also see Section 6.3 for experimental results).

Second, the way we model ordering constraints is also different. The existing techniques (e.g., References [9, 28, 51]) use integer-valued clocks to model the time of occurrence for each event and use differences between clock values to model the order among events. Moreover, they use clock equalities to represent the atomicity of events in the same atomic operation. Then, they can rely on the integer difference logic to solve ordering constraints. However, note here, we do not need to compute the exact occurrence time of each event, but only their order. We thus intend to develop a dedicated solver for ordering consistency theory.

Finally, compared to the encoding in Reference [9], the generation of each *RF* and *WS* constraint is slightly different, i.e, in our encoding, if an *RF* or a *WS* variable is assigned *true*, the two related events must both be enabled, while the encoding in Reference [9] has no such requirement. As a result, the derivation of *FR* orders with our encoding need not consider guard conditions anymore. This change is quite important, since the guard conditions often involve arithmetic computation and data structures, which can hardly be handled by a dedicated theory solver for order constraints.

3.4 Correctness of the Encoding

Given a concurrent program P and a memory model mm, the symbolic encoding procedure outputs a formula $\Psi := \Phi_{ssa} \wedge \Phi_{ord}$. Even though our encoding is slightly different from that in Reference [9], we can prove that $\Psi \wedge \rho_{fr}$ is equi-satisfiable with the encoding in Reference [9]; we thus have the following theorem:

THEOREM 1. *The formula $\Psi \wedge \rho_{fr}$ is* satisfiable *iff there is a valid counterexample in the program.*

4 ORDERING CONSISTENCY THEORY

This section presents our ordering consistency theory. We first introduce its definition, then discuss a data structure that is useful for its reasoning.

4.1 Theory Definition

The *theory of ordering consistency* $\mathcal{T}_{ord}$ has the signature

$$\Sigma_{ord} : \{e_1, e_2, \ldots, \prec_{ppo}, \prec_{ws}, \prec_{rf}, \prec_{fr}, \approx\},$$

where

- $e_1, e_2, \ldots$ are constants, intended to represent the access events in $\mathbb{E}$,
- $\prec_{ppo}, \prec_{ws}, \prec_{rf}, \prec_{fr}$ are binary predicates, intended to represent different orders among access events, and
- $\approx$ is a binary predicate, intended to represent the atomicity relation of the program.

A Σ_{ord}-atom is either a Boolean variable or a predicate $e_1 \circ e_2$, where $\circ \in \{\prec_{ppo}, \prec_{rf}, \prec_{ws}, \prec_{fr}, \approx\}$. A Σ_{ord}-formula is constructed from Σ_{ord}-atoms using Boolean connectives. Recall the ordering constraints ρ_{ppo}, $\rho_{rf\text{-}ord}$, $\rho_{ws\text{-}ord}$, and $\rho_{\approx}$ (in Section 3). They are all Boolean combinations of Σ_{ord}-atoms, and thus are Σ_{ord}-formulas; the formula $\Phi_{ord} = \rho_{ppo} \wedge \rho_{rf\text{-}ord} \wedge \rho_{ws\text{-}ord} \wedge \rho_{\approx}$ is also a Σ_{ord}-formula.

Each predicate symbol in Σ_{ord} defines a *binary relation* over $\mathbb{E}$. We use the same symbol for a predicate and the binary relation it defines. Now, we discuss the axioms of $\mathcal{T}_{ord}$.

AXIOM 1 (PARTIAL ORDER). *Predicates $\prec_{ppo}, \prec_{ws}, \prec_{rf}, \prec_{fr}$ in Σ_{ord} represent* partial orders*, and*

- *$\prec_{ws}, \prec_{rf}, \prec_{fr}$ are over accesses to the same memory address;*
- *$\forall e_1, e_2.\ e_1 \prec_{ws} e_2 \rightarrow type(e_1) = type(e_2) = \mathtt{W}$;*
- *$\forall e_1, e_2.\ e_1 \prec_{rf} e_2 \rightarrow type(e_1) = \mathtt{W} \wedge type(e_2) = \mathtt{R}$;*
- *$\forall e_1, e_2.\ e_1 \prec_{fr} e_2 \rightarrow type(e_1) = \mathtt{R} \wedge type(e_2) = \mathtt{W}$.*

AXIOM 2 (EQUIVALENCE RELATION). *The predicate $\approx$ represents an equivalence relation.*

AXIOM 3 (*FR* DERIVATION). *For any two write events $e_1, e_2 \in \mathbb{E}$ and a read event $e_3 \in \mathbb{E}$ with $addr(e_1) = addr(e_2) = addr(e_3)$, $type(e_1) = type(e_2) = \mathtt{W}$, and $type(e_3) = \mathtt{R}$, we have:*

$$e_1 \prec_{rf} e_3 \wedge e_1 \prec_{ws} e_2 \Rightarrow e_3 \prec_{fr} e_2.$$

Axiom 4 (Consistency). *The union $\prec_{ws} \cup \prec_{rf} \cup \prec_{fr}$ needs to be consistent with $\prec_{ppo}$ and $\approx$.*

The above axioms define the intended semantics of $\prec_{ppo}$, $\prec_{ws}$, $\prec_{rf}$, $\prec_{fr}$, $\approx$, as we understand them in the preceding sections. Note that Axiom 4 should hold after any number of applications of Axiom 3, i.e., after deriving any number of $\prec_{fr}$ orders.

4.2 Event Graph

Let $\alpha : X_{rf} \cup X_{ws} \to \{true, false, unassigned\}$ be the current assignment to ordering variables, and $\prec_\alpha$ the set of induced *RF*, *WS*, and *FR* orders by α. Let $\prec$ be the union $\prec_{ppo} \cup \approx \cup \prec_\alpha$. The event set $\mathbb{E}$ and the order set $\prec$ can be represented as a graph, called an *event graph*, where events are represented as nodes and orders as edges.[6]

At the beginning of SMT solving, all ordering variables are *unassigned*; no *RF* or *WS* edges are drawn on the event graph. Since *FR* orders are derived from *RF* and *WS* orders, there are no *FR* edges, either. Therefore, only *PPO* and *atomicity* edges present in the graph at that moment. The event set $\mathbb{E}$, the preserved program order *PPO*, and the *atomicity* constraints make up the *skeleton* of the event graph. Later, along with variable assignments, more edges are added to the graph.

According to the axioms of $\mathcal{T}_{ord}$, on each edge addition, we need to check whether this new edge leads to a cycle. If this is the case, then we say the current variable assignment is *invalid*—we then need to analyze the event graph to find the inconsistency reason. Otherwise, if there is no cycle, then we go ahead to apply Axiom 3 to derive *FR* edges. Note that if any *FR* edge is derived, then we need to check the consistency of the current variable assignment again.

We associate each edge with a Boolean expression, called *derivation reason* (abbreviated as *reason*), to indicate what this edge is derived from.

- The *reason* for a *PPO* or an *atomicity* edge is *true*, for this edge always presents in the graph.
- The *reason* for an *RF* or a *WS* edge is the corresponding ordering variable, for this edge is directly derived from this variable.
- The *reason* for an *FR* edge is the conjunction of *reasons* for the *RF* edge and the *WS* edge that derive this *FR* edge.

The concept of *derivation reason* can be lifted to a path. Let $e_1 \prec e_2 \prec \cdots \prec e_n$ be a path on the event graph; the *reason* for this path is the conjunction of *reasons* for each edge it passes, i.e.,

$$reason(e_1 \prec e_2 \prec \cdots \prec e_n) = \bigwedge_{i=1}^{n-1} reason(e_i \prec e_{i+1}).$$

Figure 3 shows several event graphs that may occur during SMT solving of the program in Figure 2. To differentiate event types, we use *grey* and *white* nodes to represent write and read events, respectively. Preserved program orders are drawn as solid lines, while others are drawn as dashed lines. In the beginning, the event graph contains only *PPO* edges, as shown in Figure 3(a). After some assignments, more edges are added to the graph; Figure 3(b) shows an updated event graph of Figure 3(a) during SMT solving, in which $(\!|n_1|\!)^w \prec_{rf} (\!|n_2|\!)^r$ and $(\!|n_1|\!)^w \prec_{ws} (\!|n_4|\!)^w$ derive $(\!|n_2|\!)^r \prec_{fr} (\!|n_4|\!)^w$. Finally, the red dashed edges in Figure 3(c) form a cycle, indicating a $\mathcal{T}_{ord}$-inconsistent execution.

An event graph may be different under weak memory models. Figure 3(a) shows an event graph under SC; Figures 3(d) and 3(e) show event graphs of the same program under TSO and PSO, respectively. Edges of these three graphs are quite different—some edges are deleted and some are added. However, in comparison with their transitive closures (the *PPO* relation is transitive), the graphs of TSO and PSO always contain less program orders. For example, considering the edge

[6]Equivalence relations are regarded as undirected edges.

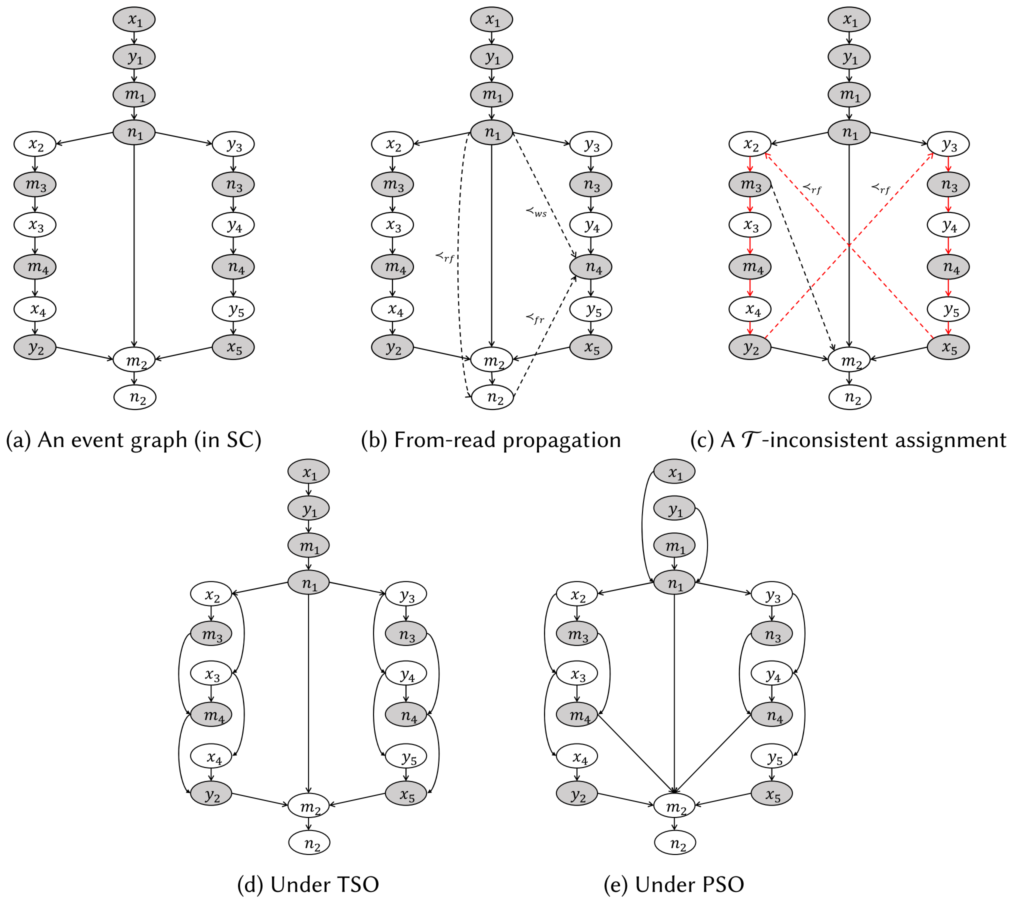
(a) An event graph (in SC)
(b) From-read propagation
(c) A $\mathcal{T}$-inconsistent assignment
(d) Under TSO
(e) Under PSO
$\prec_{rf}$
$\prec_{ws}$
$\prec_{fr}$

Fig. 3. Updates of the event graph in SMT solving.

$(\!|x_2|\!)^r \prec_{ppo} (\!|x_3|\!)^r$ added in both Figures 3(d) and 3(e), the corresponding program order is also contained in the transitive closure of Figure 3(a). In contrast, for edge $(\!|m_3|\!)^w \prec_{ppo} (\!|x_3|\!)^r$, which is deleted in Figures 3(d) and 3(e), the corresponding program order is also relaxed.

Similar structures to our event graph were defined in References [9, 51, 61]. Note that the events discussed in this article can hold symbolic values, and thus our event graph is actually a "symbolic" event graph. In Reference [9], a so-called *symbolic event structure* is defined, which, however, is used to depict program order only. Moreover, the *event order graph* defined in Reference [61] represents a counterexample instead of a program. In Reference [51], an *interference skeleton* is defined, which equals the *skeleton* of our event graph.

5 THEORY SOLVER FOR $\mathcal{T}_{ord}$

In this section, we present $\mathcal{T}_{ord}$-solver with emphasis on algorithms for consistency checking, conflict clause generation, and theory propagation.

5.1 Overview

Figure 4 shows an overview of $\mathcal{T}_{ord}$-solver. Each time an ordering variable is assigned in the SAT solver, $\mathcal{T}_{ord}$-solver performs consistency checking (see Section 5.2) to detect if a cycle exists after the corresponding edge addition to the event graph.

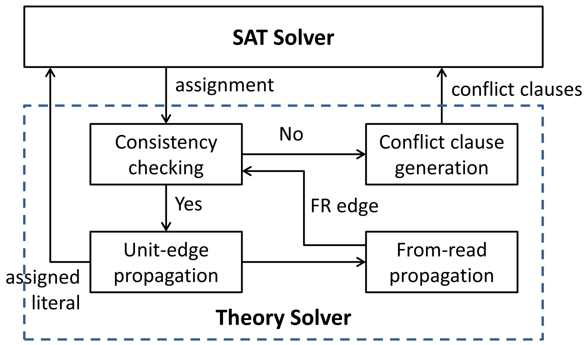

Fig. 4. $\mathcal{T}_{ord}$-solver with DPLL(T).

If the current assignment is $\mathcal{T}_{ord}$-consistent, then $\mathcal{T}_{ord}$-solver proceeds to: (1) determine values of unassigned literals by using axioms of $\mathcal{T}_{ord}$ (called *unit-edge propagation*; see Section 5.5.1), and (2) deduce all possible *FR* edges with respect to the assignment (called *from-read propagation*; see Section 5.5.2). If any unassigned literal is assigned, then this assignment should be returned to the SAT solver; if any *FR* edge is deduced, then the consistency checking needs to be invoked again.

If the current assignment is $\mathcal{T}_{ord}$-inconsistent, then $\mathcal{T}_{ord}$-solver computes conflict clauses (called *conflict clause generation*; see Section 5.3) to record the inconsistency reason, returns it to DPLL(T) to prevent the solver from going down the same path in the future.

5.2 Consistency Checking

Each time an ordering variable is assigned *true*, $\mathcal{T}_{ord}$-solver inserts the corresponding edge into the event graph and performs consistency checking. Consistency checking can be reduced to cycle detection on the event graph.

Let $\prec_{ppo}$ be the preserved program order of the program, $\approx$ the atomicity relation of the program, and $\prec_\alpha$ the set of induced *RF*, *WS*, and *FR* orders by the current assignment α. The basic routine for consistency checking is as follows:

- If $\prec_{ppo} \cup \prec_\alpha \cup \approx$ has no cycles, then so does its subset $(\prec_{ppo} \cup \prec_\alpha) \cap \approx$. Then, both conditions of Lemma 1 are satisfied. By Lemma 1, $\prec_\alpha$ is consistent with $\prec_{ppo}$ and $\approx$.
- If $\prec_{ppo} \cup \prec_\alpha \cup \approx$ has a cycle that is not contained in $\approx$, then this cycle must involve at least two equivalence classes of $\approx$, indicating that condition (2) of Lemma 1 is violated. By Lemma 1, $\prec_\alpha$ is not consistent with $\prec_{ppo}$ and $\approx$.
- Otherwise, all cycles of $\prec_{ppo} \cup \prec_\alpha \cup \approx$ are contained in $\approx$, i.e., condition (2) of Lemma 1 is satisfied. We continue to check:
  - If $(\prec_{ppo} \cup \prec_\alpha) \cap \approx$ has cycles, then condition (1) of Lemma 1 is violated. By Lemma 1, $\prec_\alpha$ is not consistent with $\prec_{ppo}$ and $\approx$.
  - Otherwise, by Lemma 1, $\prec_\alpha$ is consistent with $\prec_{ppo}$ and $\approx$.

The following theorem ensures the correctness of consistency checking:

Theorem 2. *The above consistency checking procedure finds no cycle if $\prec_\alpha$ is consistent with $\prec_{ppo}$ and $\approx$; otherwise, it can find cycle (s).*

The efficiency of consistency checking is critical for the overall performance, because the model checker performs many consistency checks. First, DPLL(T) must perform many assignments to the ordering variables to reason about the complicated thread interactions, each of which leads to a consistency check. Second, *from-read propagation* generates further checks whenever it inserts edges into the event graph.

Previous works [9, 28] perform a fresh cycle detection on each consistency check, which is inefficient. We found it better to perform cycle detection incrementally for two reasons. First, the event graph must be acyclic before an edge addition—otherwise, it must have been recognized in the previous consistency checking. Therefore, we can reuse the topological order of the previous acyclic graph and try to build a new acyclic graph incrementally. Second, incremental cycle detection has been shown to be efficient for sparse graphs [12], and the event graph is typically sparse: $\prec_{ppo}$ only relates events within the same thread (as well as thread creating/joining), while $\prec_{\alpha}$ only relates events that access the same shared variable.

5.2.1 Incremental Cycle Detection. We employ an ***incremental cycle detection (ICD)*** algorithm [7, 12] to check $\mathcal{T}_{ord}$-consistency. ICD algorithms are based on the topological order in directed graphs (including the event graph). A topological order exists in a directed graph iff the graph is acyclic: Each node in the graph is labeled with an integer-valued level such that for any edge, say, from node e_i to node e_j, the level of e_i (written $lv(e_i)$) is smaller than that of e_j (written $lv(e_j)$). Once a new edge is inserted into the event graph, the algorithm reuses the previous topological order and attempts to compute a new topological order incrementally. If a new topological order is computed, then the graph is acyclic; therefore, the current assignment is $\mathcal{T}_{ord}$-consistent. Otherwise, the algorithm finds a cycle; therefore, a $\mathcal{T}_{ord}$-inconsistency is reported.

We employ an ICD algorithm for sparse graphs [12]. This algorithm relaxes the topological order into a *pseudo-topological order* such that for any edge from node e_i to node e_j, $lv(e_i) \leq lv(e_j)$. The pseudo-topological order can handle atomicity by assigning the same integer-valued level to all events in an atomic block. For each node e, except of its pseudo-topological level $lv(e)$, the ICD algorithm also keeps $in(e)$ and $out(e)$, where $in(e)$ stores e's incoming edges whose start nodes have the same level as e, and $out(e)$ stores e's outgoing edges. During the process of inserting an edge into the graph, the pseudo-topological levels, outgoing edges sets, and incoming edges sets are incrementally updated, and these processes are described in detail in the following text.

We show the basic routine of the employed ICD algorithm below. When adding an edge, say, $e_i \prec e_j$, into the event graph, we first check whether $lv(e_i) < lv(e_j)$. If so, then e_j cannot reach e_i; thus, no cycle exists, and the previous pseudo-topological order is valid; therefore, $\mathcal{T}_{ord}$-solver can safely insert this edge and update $out(e_i)$. Otherwise, $lv(e_i) \geq lv(e_j)$, we need to check whether there is a path from e_j to e_i. If it does exist, this path, together with $e_i \prec e_j$, forms a cycle, indicating that the current edge insertion causes $\mathcal{T}_{ord}$-inconsistency. Otherwise, e_j cannot reach e_i (i.e., no cycle exists), the ICD algorithm reports the current edge insertion being $\mathcal{T}_{ord}$-consistent. Meanwhile, the previous pseudo-topological order and the related *in*/*out* sets are updated in the search process.

The standard approach to finding a path from e_j to e_i is to search forward exhaustively from the outgoing edges of e_j, e.g., Tarjan's ***strongly connected component*** **(SCC)** algorithm [53]. However, this approach totally costs $O(m^2)$ time to construct a graph with m edges. In contrast, the employed ICD algorithm [12] in $\mathcal{T}_{ord}$-solver performs cycle detection based on an elaborate *two-way search*, i.e., *backward search* and *forward search*. Bender et al. [12] prove that by setting a threshold $\Delta = min\{n^{\frac{2}{3}}, m^{\frac{1}{2}}\}$ to bound the maximum number of steps in the backward search, the employed ICD algorithm can achieve $O(m \times min\{n^{\frac{2}{3}}, m^{\frac{1}{2}}\})$ time complexity in constructing a graph with n nodes and m edges. Since thread interleaving may cause numerous edge insertions, the employed ICD algorithm can achieve higher efficiency than Tarjan's SCC algorithm. In Section 6.3, we report the comparison results between the ICD algorithm and Tarjan's SCC algorithm.

We detail the two-way search below. When adding an edge, say, $e_i \prec e_j$, into the event graph and $lv(e_i) \geq lv(e_j)$, before the forward search for a path from e_j to e_i, a backward search starts from e_i first. Recall that $in(e_i)$ only stores e_i's incoming edges $e \prec e_i$ with $lv(e) = lv(e_i)$. Following the *in* sets, we search for nodes that can reach e_i and have the same level of e_i backwardly. We use a set $\mathbb{B}$ to record the visited nodes during the backward search. The backward search ends in any of the following situations:

- If e_j is visited, then the algorithm reports a cycle and returns.
- If the backward search completes before visiting Δ edges, then the ICD algorithm runs out of the edges that should be traversed in the backward search and fails to find a path from e_j to e_i. Since $lv(e_i) \geq lv(e_j)$, there are two situations:
  - If $lv(e_i) = lv(e_j)$, then no cycle exists. We prove this by contradiction. Assume there is a path from e_j to e_i. Since $lv(e_i) = lv(e_j)$, all nodes along the path should have the same level, and the backward search should be able to find e_j. But the backward search completes without finding e_j, which is a contradiction. Therefore, there is no path from e_j to e_i. The algorithm thus reports acyclic and returns.
  - If $lv(e_i) > lv(e_j)$, then for each visited $e_b \in \mathbb{B}$, $lv(e_b) = lv(e_i) > lv(e_j)$. In this case, the algorithm does not know if there is a path from e_j to e_b—so we set $lv(e_j) := lv(e_i)$ and attempt to compute a new pseudo-topological order by invoking a forward search from e_j.
- If Δ edges are visited, we stop backward search, set $lv(e_j) := lv(e_i) + 1$, and then attempt to compute a new pseudo-topological order by invoking a forward search from e_j.[7]

The forward search explores nodes reachable from e_j. Among outgoing edges in $out(e_j)$, we consider only edges $e_j \prec e_k$ such that $lv(e_j) \geq lv(e_k)$:

- if $lv(e_j) = lv(e_k)$, then add e_j to $in(e_k)$;
- if $lv(e_j) > lv(e_k)$, then set $lv(e_k) := lv(e_j)$, clear $in(e_k)$, and then add e_j to $in(e_k)$.

After e_k is visited, we continue to check the outgoing edges of e_k in the same way. Nodes visited in the forward search are stored in a set $\mathbb{F}$. We then check if $\mathbb{F} \cap \mathbb{B}$ produces an empty set. If not, then any node $e \in \mathbb{F} \cap \mathbb{B}$ witnesses a cycle, composed of the path segment from e_j to e (by $e \in \mathbb{F}$), the segment from e to e_i (by $e \in \mathbb{B}$), and the inserted edge $e_i \prec e_j$. Otherwise, if the forward search completes with $\mathbb{F} \cap \mathbb{B} = \emptyset$, then we confirm the absence of any cycle.

Note that even if we skip the backward search but only perform an exhaustive forward search, the algorithm is still correct but degenerates into a fresh cycle detection on each edge insertion. From this perspective, the key insight of the employed ICD algorithm is to integrate a backward search—bounded by a "magic" threshold $\Delta = min\{n^{\frac{2}{3}}, m^{\frac{1}{2}}\}$—to limit the forward search. The following lemma is proved in Reference [12], which is crucial for analyzing the complexity of the ICD algorithm:

Lemma 5 ([12]). *For a graph with n nodes and m edges, with the ICD algorithm, no node's pseudo-topological level is greater than* $\Delta + 2$.

Complexity of the ICD algorithm can be analyzed as follows: First, the backward search of a single edge insertion visits at most Δ edges, so the backward search during m edge insertions visits at most $m \times \Delta$ edges. Second, the forward search is invoked when the algorithm updates the pseudo-topological level of a node, which, according to Lemma 5, happens at most $\Delta + 2$ times per edge during the whole m edge insertions. Therefore, forward search during m edge insertions visits at most $m \times (\Delta + 2)$ edges. Finally, each edge takes $O(1)$ time to visit. Thus, the ICD algorithm's complexity is $O(m \times \Delta)$.

[7]If the graph contains no cycles, then we can always construct a new topological order at the end of the search.

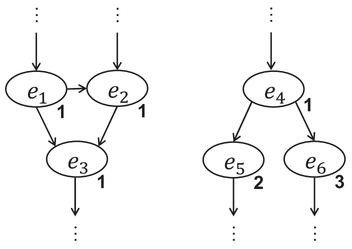

Fig. 5. The original graph.

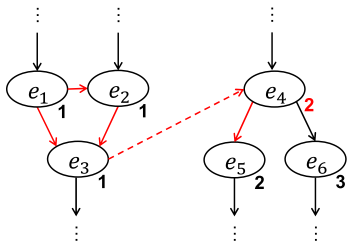

Fig. 6. The graph after inserting $e_3 \prec e_4$.

Example. Figure 5 shows parts of nodes and edges of a graph, where the digit alongside each node represents the pseudo-topological level of that node. Assume Δ is currently 3, and the edge $e_3 \prec e_4$ (the red dashed arrow in Figure 6) is inserted.

The algorithm first checks whether $lv(e_3) < lv(e_4)$—the result is not, thus the backward search is invoked from e_3. The backward search visits $e_1 \prec e_3$, $e_2 \prec e_3$, and $e_1 \prec e_2$ (marked as red solid arrows in Figure 6) in turn; and then stops (since the number of visited edges reaches Δ), and sets $lv(e_4) = lv(e_3) + 1 = 2$. The set $\mathbb{B} = \{e_1, e_2, e_3\}$ records nodes visited in the backward search.

Then, the forward search starts from e_4. Note that $lv(e_4)$ is now 2, equaling $lv(e_5)$, the forward search visits $e_4 \prec e_5$ and adds e_4 to $in(e_5)$; moreover, since $lv(e_4) < lv(e_6)$, the forward search ignores $e_4 \prec e_6$. The set $\mathbb{F} = \{e_4, e_5\}$ records nodes visited in the forward search. Finally, we confirm there is no cycle, since $\mathbb{B} \cap \mathbb{F} = \emptyset$.

5.3 Conflict Clause Generation

If a $\mathcal{T}_{ord}$-inconsistency occurs, then we need to find the inconsistency reason and report it to the SAT solver. To find the inconsistency reason, it is sufficient to consider *critical cycles* [50]. Formally, a cycle is *critical* if it is *simple* (i.e., no duplicate nodes) and has no chords in $\prec_{ppo} \cup \approx$ [50]. Recall that $\mathcal{T}_{ord}$-solver records a *derivation reason* for each edge, and the *derivation reason* of a path can be calculated accordingly (see Section 4.2). The *derivation reason* of critical cycles can be returned as the inconsistency reason. To ease the following discussions, we call the edges corresponding to *PPO* and atomicity orders *static* edges (they are fixed during SMT solving), and the edges corresponding to *RF*, *WS*, and *FR* orders *induced* edges (they are induced by variable assignments).

When a $\mathcal{T}_{ord}$-inconsistency occurs, the event graph may contain many critical cycles; we prefer those with the shortest *width* (defined as the number of induced edges on the cycle). Their *derivation reasons* contain the minimal number of ordering literals and can be used to prune more search space. If there are multiple critical cycles with the shortest *width*, then we generate them all.

An important fact is that the event graph must be acyclic before the current edge insertion. Therefore, the newly added edge should present in all cycles. Let $e_i \prec e_j$ be the newly added edge; the conflict clause generation needs to find all *derivation reasons* of $e_j \prec^+ e_i$ with the shortest *width*.

Let $\mathbb{E}_{j-i}$ be the set of nodes that occur on any path of $e_j \prec^+ e_i$, including e_j and e_i themselves. For each $e_n \in \mathbb{E}_{j-i}$, denote $reasons(e_n)$ the set of all *derivation reasons* of $e_j \prec^+ e_n$ with the shortest *width*. We compute $reasons(e_n)$ in the following routine:

Step 1 (Subgraph construction). We first construct subgraph $\mathbb{E}_{j-i}$. Remember that in consistency checking, set $\mathbb{B}$ contains visited nodes from incoming edges of e_i and set $\mathbb{F}$ contains visited nodes from outgoing edges of e_j. Actually, for each node in $\mathbb{B}$ or $\mathbb{F}$, we also record its parents (e.g., if edges $e_1 \prec e_3$ and $e_2 \prec e_3$ are visited in the forward search, then e_3's parents are $\{e_1, e_2\}$). In consistency checking, once a cycle is detected when visiting a node (assumed to be e_k), then e_k

must be in both $\mathbb{B}$ and $\mathbb{F}$. We can find all nodes on path $e_j \prec^+ e_i$ by tracking back to e_k's parents; we add these nodes to $\mathbb{E}_{j-i}$. We construct this subgraph of the event graph by removing all nodes other than $\mathbb{E}_{j-i}$ and deleting induced edges that have a chord in $\prec_{ppo} \cup \approx$ (e.g., $(\!|n_1|\!)^w \prec_{rf} (\!|n_2|\!)^r$ in Figure 3(b)).

Step 2 (Iterative solving). We traverse the subgraph in topological order, starting from the outgoing edges of e_j. Let e_n be the current node to be visited. Without loss of generality, when there are multiple edges waiting to be visited, we first visit *PPO* edges. There are two situations:

- When visiting *PPO* edge $e_p \prec_{ppo} e_n$, we append $reasons(e_p)$ to $reasons(e_n)$. However, if node e_n has been visited once (so $width(e_n)$ has been calculated) and $width(e_p) > width(e_n)$, then we skip this visit.
- We visit induced edges only when no *PPO* edges are waiting to be visited. When visiting induced edge $e_p \prec e_n$ (either $\prec_{rf}$, $\prec_{ws}$, or $\prec_{fr}$), we append $reasons(e_p)$ to $reasons(e_n)$ and add $reason(e_p \prec e_n)$ to each newly appended reason. However, if node e_n has been visited once and $width(e_p) > width(e_n) - 1$, then we skip this visit.

In this traversal order, we can always find the shortest *width* of a node at its first visit, i.e., we first visit all nodes whose $width = 0$, then all nodes whose $width = 1$, and so on.

Formally, suppose the predecessors of e_n are $e_{p1}, \dots, e_{pa}, e_{q1}, \dots, e_{qb}$ where $e_{pi} \prec_{ppo} e_n$ and $e_{qj} \prec e_n$ (induced edges). According to the procedure described above, we have:

$$width(e_n) = min\{width(e_{p1}), \dots, width(e_{pa}), width(e_{q1}) + 1, \dots, width(e_{qb}) + 1\},$$

so $\forall e_{pi}\ width(e_{pi}) \geq width(e_n)$ and $\forall e_{qj}\ width(e_{qj}) \geq width(e_n) - 1$.

Denote $SP(e_n)$ the set of *shortest predecessors* of e_n such that the paths $e_j \prec^+ SP(e_n) \prec e_n$ have the shortest *width*. We lift $\wedge$ operator to sets, and compute $reasons(e_n)$ as

$$\bigcup_{e_p \in SP(e_n)} reasons(e_p) \wedge reason(e_p \prec e_n).$$

After the traversal, $reasons(e_i)$ records the set of shortest *derivation reasons* of $e_j \prec^+ e_i$. A path in $e_j \prec^+ e_i$ and $e_i \prec e_j$ forms a cycle. We append $reason(e_i \prec e_j)$ to each reason in $reasons(e_i)$ and return them as conflict clauses.

Example. Consider the event graph in Figure 7. Let $e_3 \prec_{fr} e_1$ be the newly added edge. The consistency checking reports $\mathcal{T}_{ord}$-inconsistency, i.e., cycle is detected in the event graph.

- First, we construct the subgraph by keeping all nodes that appear on any path from $e_1 \prec^+ e_3$. There are two paths: $e_1 \prec_{ppo} e_2 \prec_{rf} e_3$ and $e_1 \prec_{rf} e_4 \prec_{fr} e_5 \prec_{ppo} e_2 \prec_{rf} e_3$; therefore, e_6 and e_7 are removed.
- Second, we traverse the subgraph in topological order, starting from the outgoing edges of e_1. Note that *width* can be computed as the number of $\prec_{rf}$, $\prec_{ws}$, and $\prec_{fr}$ edges in the cycle; Figures 8 and 9 show two cycles with $width = 2$ and $width = 4$, respectively. Since we try to find cycle(s) with the shortest *width*, we return the conflict clause: The conjunction of ordering literals that imply $e_2 \prec_{rf} e_3$ and $e_3 \prec_{fr} e_1$.

We show the correctness and complexity of our conflict clause generation algorithm by the following theorems.

THEOREM 3. *The above conflict clause generation algorithm finds all conflict clauses with the shortest width.*

PROOF. Using mathematical induction, we prove that for each node e_n with predecessors $e_{p1}, \dots, e_{pa}, e_{q1}, \dots, e_{qb}$ where $e_{pi} \prec_{ppo} e_n$ and $e_{qi} \prec e_n$ (induced edges), $reasons(e_n)$ contains

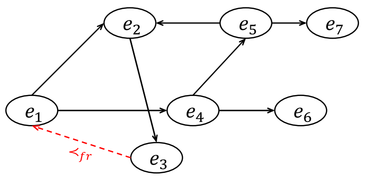

Fig. 7. The original graph.

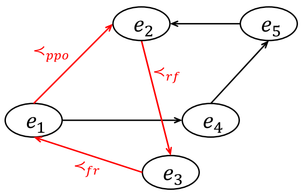

Fig. 8. Cycle α: $width = 2$.

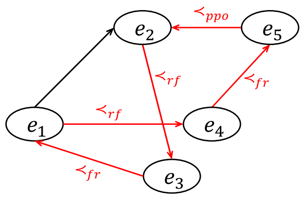

Fig. 9. Cycle β: $width = 4$.

all reasons for $e_j \prec^+ e_n$ with the shortest *width*, where soundness means there is no shorter reason for $e_j \prec^+ e_n$; and completeness means there is no reason for $e_j \prec^+ e_n$ with the same *width* but not contained in $reasons(e_n)$. Both soundness and completeness trivially hold for e_j, for the graph is acyclic before the edge addition.

Soundness: Assume there exists a path $path_{err}$ from e_j to e_n corresponding to $reason_{err}$ of $e_j \prec^+ e_n$ whose $width_{err} < width(e_n)$, If its last step is $e_{pi} \prec e_n$, then $reasons(e_{pi})$ must include this $reason_{err}$, which contradicts $width(e_{pi}) \geq width(e_n)$; otherwise, if the last step is $e_{qj} \prec e_n$, then $reasons(e_{qj})$ must include $reason_{err}$ without the last reason(whose *width* is $width_{err} - 1$), which contradicts $width(e_{qj}) + 1 \geq width(e_n)$. Thus, we prove the soundness of n from the soundness of its predecessors.

Completeness: Assume there exists a path $path_{err}$ from e_j to e_n corresponding to $reason_{err}$ of $e_j \prec^+ e_n$ whose *width* is $width(e_n)$ but not included in $reasons(e_n)$. If the last step of $path_{err}$ is $e_{pi} \prec e_n$, then $reason_{err}$ is also a reason of $e_j \prec^+ e_{pi}$, thus, by soundness of e_{pi}, $width(e_{pi}) \leq width(e_n)$, which gives $width(e_{pi}) = width(e_n)$. By completeness of e_{pi}, $reason_{err}$ is included in $reasons(e_{pi})$ and should be collected by $reasons(e_n)$, which gives a conflict; otherwise, if the last step is $e_{qj} \prec e_n$, then $reason_{err}$ without the last reason is a reason of $e_j \prec^+ e_{qj}$, thus $width(e_{qj}) \leq width(e_n) - 1$, which gives $width(e_{qj}) = width(e_n) - 1$. Then, $reason_{err}$ without the last reason is included in $reasons(e_{qj})$ and should be appended to the last reason and collected by $reasons(e_n)$, which also gives a conflict. Thus, we prove the completeness of e_n.

The graph has a topological order, since it is acyclic. By mathematical induction, we prove that the algorithm is sound and complete. □

THEOREM 4. *The time complexity of our conflict clause generation algorithm is $O(c \times m')$, where c is the number of computed conflict clauses and m' is the number of edges in the constructed subgraph.*

PROOF. A reason is a conjunction of literals, usually implemented as a vector or set. To reduce the time cost of copying reasons, when a reason is copied from $reasons(d)$ to $reasons(e)$, we only copy its pointer. When a *new_reason* is constructed by appending r to an existing *reason*, we represent the *new_reason* with r appending to the pointer of *reason*.

In this manner, any *reason* whose last element is r is constructed when visiting nodes e where $reason(d \prec e) = r$. Thus, each *reason* is constructed only once and we can give each *reason* a unique ID to distinguish different *reasons*, i.e., it is easy to identify and remove duplicate *reason* when computing $reasons(e)$. $reasons(e)$ has at most c elements, so for each edge $e_x \prec e_y$ in the subgraph, at most c reasons are copied from $reasons(e_x)$ to $reason(e_y)$. As shown above, either copy or construction of reasons cost $O(1)$ time. Thus, the whole algorithm takes $O(c \times m')$ time. □

5.4 Backtracking

After the SAT solver being noticed of the conflict clauses, it backtracks to a previous state to get rid of the current inconsistency. On this occasion, $\mathcal{T}_{ord}$-solver also needs to restore the event graph to the correlating previous state, i.e., to remove all edges added after this state. Each edge deletion takes $O(1)$ time. Moreover, $\mathcal{T}_{ord}$-solver needs to restore the pseudo-topological levels, which is necessary to preserve ICD's complexity analyzed in Reference [12]. The restoration of each node also takes $O(1)$ time. Therefore, in the worst case (all edges are removed), $\mathcal{T}_{ord}$-solver costs $O(m+n)$ time on backtracking an event graph with n nodes and m edges. In practice, the time complexity is often much smaller, since usually only a small portion of edges are removed.

5.5 Theory Propagation

We employ theory propagation to derive more ordering constraints. In detail, $\mathcal{T}_{ord}$-solver deduces values of unassigned literals by *unit-edge propagation* and derives from-read orders by *from-read propagation.*

5.5.1 Unit-edge Propagation. For ease of implementation, we pre-create an edge for each ordering variable in $X_{rf} \cup X_{ws}$. Each of these edges has two states, `active` and `inactive` (initially `inactive`). Only `active` edges are present in the event graph. An `inactive` edge is *activated* when the corresponding ordering variable is set to *true*. An `active` edge is *inactivated* if the corresponding ordering variable is unassigned (due to backtracking of DPLL(T)).

Let $e_i \prec e_j$ be an `inactive` edge for an ordering variable v. It is a *unit edge* if there already exists a path from e_j to e_i in the event graph. In other words, once the ordering variable v is assigned *true*, a cycle $e_i \prec e_j \prec^+ e_i$ forms. To prevent this cycle, v must be set to *false*. In this way, we deduce the value of an unassigned variable. We call this *unit-edge propagation.*

Unit-edge propagation is performed after incremental cycle detection. Let $\mathbb{B}$ and $\mathbb{F}$ be the node sets obtained in the backward and forward search of ICD, respectively. For any node $e_b \in \mathbb{B}$ and any node $e_f \in \mathbb{F}$, there must be a path from e_b to e_f that passes the newly added edge. We enumerate each such node pair and check if (e_f, e_b) corresponds to an `inactive` edge; if it does, then the corresponding `inactive` edge is a unit edge.

Figure 3(c) shows a cycle led by assignments $rf^y_{2,3} \mapsto$ *true* and $rf^x_{5,2} \mapsto$ *true*. Assuming that $rf^x_{5,2} \mapsto$ *true* is assigned first: the edge $(\!|x_5|\!)^w \prec_{rf} (\!|x_2|\!)^r$ is added; then there forms a path from $(\!|y_3|\!)^r$ to $(\!|x_5|\!)^w$ (by *PPO* edges), to $(\!|x_2|\!)^r$ (by this added edge), and to $(\!|y_2|\!)^w$ (by *PPO* edges). According to our *unit-edge propagation*, $(\!|y_2|\!)^w \prec_{rf} (\!|y_3|\!)^r$ is a unit edge, so the value of $rf^y_{2,3}$ is enforced to *false*. In this way, our *unit-edge propagation* can prevent the $\mathcal{T}_{ord}$-inconsistency shown in Figure 3(c).

5.5.2 From-read Propagation. *FR* constraints are not included in our encoding formula. We depend on $\mathcal{T}_{ord}$-solver to deduce *FR* orders.

When adding an *RF* edge $(\!|x_i|\!)^w \prec_{rf} (\!|x_j|\!)^r$, $\mathcal{T}_{ord}$-solver seeks outgoing *WS* edges of node $(\!|x_i|\!)^w$. For each of such edges, say, $(\!|x_i|\!)^w \prec_{ws} (\!|x_k|\!)^w$, $\mathcal{T}_{ord}$-solver derives $(\!|x_j|\!)^r \prec_{fr} (\!|x_k|\!)^w$ and instantly adds it to the event graph. Similarly, when adding a *WS* edge $(\!|x_i|\!)^w \prec_{ws} (\!|x_j|\!)^w$, $\mathcal{T}_{ord}$-solver seeks outgoing *RF* edges of $(\!|x_i|\!)^w$, say, $(\!|x_i|\!)^w \prec_{rf} (\!|x_k|\!)^r$, and derives $(\!|x_k|\!)^r \prec_{fr} (\!|x_j|\!)^w$.

Figure 3(b) shows an example of *from-read propagation*, where $(\!|n_1|\!)^w \prec_{ws} (\!|n_4|\!)^w$ is added prior to $(\!|n_1|\!)^w \prec_{rf} (\!|n_2|\!)^r$. During the addition of $(\!|n_1|\!)^w \prec_{ws} (\!|n_4|\!)^w$, since $(\!|n_1|\!)^w$ has no outgoing *RF* edges yet, *from-read propagation* obtains nothing. Then, while adding $(\!|n_1|\!)^w \prec_{rf} (\!|n_2|\!)^r$, there is an outgoing *WS* edge $(\!|n_1|\!)^w \prec_{ws} (\!|n_4|\!)^w$ from $(\!|n_1|\!)^w$. By *from-read propagation*, we deduce $(\!|n_2|\!)^r \prec_{fr} (\!|n_4|\!)^w$.

In $\mathcal{T}_{ord}$ theory solving, only unit-edge propagation interacts with the outer SAT solver, since it can propagate values of some Boolean variables. In contrast, when from-read propagation deduces a new edge, the theory solver inserts the edge and calls the consistency checking procedure, all inside the $\mathcal{T}_{ord}$-solver.

Example. Let us consider how to verify the example program in Figure 2 using $\mathcal{T}_{ord}$-solver integrated with DPLL(T). Given the encoding formula Ψ, DPLL(T) first applies unit-clause propagation and theory propagation to make as many as possible deductions.

Before deciding any literal, DPLL(T) assigns *PPO* order constraints *true* for they occur alone in the CNF-formulated input SMT formula. *Unit-edge propagation* assigns:

$$ws^x_{5,1}, ws^y_{2,1}, ws^m_{3,1}, ws^m_{4,1}, ws^m_{4,3}, ws^n_{3,1}, ws^n_{4,1}, ws^n_{4,3}$$

to *false*. Then, $ws^x_{1,5}$, $ws^y_{1,2}$ are deduced to be *true* by DPLL(T). Ordering variables related to variable m and n are not derived for some *guards* not necessarily hold.

Assuming DPLL(T) chooses $rf^m_{3,2}$ and decides its value to *true*, we perform deduction as follows:

$$\begin{aligned}
rf^m_{3,2} &\Longrightarrow x_2 = 1 && \text{(Guard holds)}\\
&\Longrightarrow rf^x_{5,2} && \text{(RF-Val, RF-Some)}\\
&\Longrightarrow \neg rf^y_{2,3} \wedge \neg rf^y_{2,4} && \text{(Unit-edge)}\\
&\Longrightarrow rf^y_{1,3} \wedge \neg rf^y_{2,4} && \text{(RF-Some)}\\
&\Longrightarrow y_3 = 0 \wedge \neg rf^y_{2,4} && \text{(RF-Val)}\\
&\Longrightarrow rf^y_{1,4} && \text{(RF-Some)}\\
&\Longrightarrow y_4 = 0 && \text{(RF-Val).}
\end{aligned}$$

Then, we decide from which write ($(\!|n_1|\!)^w$, $(\!|n_3|\!)^w$, or $(\!|n_4|\!)^w$) the access $(\!|n_2|\!)^r$ obtains its value. First, $(\!|n_3|\!)^w$ is excluded from consideration, since its guard condition ($y_3 = 1$) conflicts with the current assignment ($y_3 = 0$). Second, $(\!|n_2|\!)^r$ is also excluded, since the values of n_2 (equals 1) and n_1 (equals 0) are not equal. Third, if $(\!|n_2|\!)^r$ reads from $(\!|n_4|\!)^w$, then $n_2 = n_4 = y_4 = 1$, conflicting with the current assignment ($y_4 = 0$), too. Therefore, the deduction from $rf^m_{3,2} = true$ gets to a contradiction.

Then, DPLL(T) backtracks and assigns $rf^m_{3,2}$ to *false*, i.e., $(\!|m_2|\!)^r$ cannot read from $(\!|m_3|\!)^w$. Note that $(\!|m_2|\!)^r$ also cannot read from $(\!|m_1|\!)^w$ for $m_2 \neq m_1$. Thus, $rf^m_{4,2}$ is the only choice. The subsequent deduction is as follows:

$$\begin{aligned}
rf^m_{4,2} &\Longrightarrow x_3 = m_4 = m_2 = 1 && \text{(RF-Val)}\\
&\Longrightarrow \neg rf^x_{1,3} && \text{(RF-Val)}\\
&\Longrightarrow rf^x_{5,3} && \text{(RF-Some)}\\
&\Longrightarrow \neg rf^y_{2,3} \wedge \neg rf^y_{2,4} && \text{(Unit-edge).}
\end{aligned}$$

Now the deduction gets to the same point as in the third line of the first deduction procedure. The same as in the first deduction, it also leads to a contradiction.

From the deduction procedures above, we conclude that the encoding formula for the program is unsatisfiable. Therefore, the safety property of this program holds.

6 EXPERIMENTAL EVALUATION

This section introduces the implementation of our approach and reports the comparative results with some state-of-the-art verification tools.

6.1 Implementation and Setup

We implemented our techniques on top of CBMC [41] and Z3 [21]. CBMC is powerful and flexible bounded model checker for C/C++ programs, and Z3 is a well-known and widely adopted SMT solver. In our implementation, CBMC and Z3 act as the front and back ends, responsible for generating and solving SMT formulas, respectively. We enhance CBMC by consulting our $\mathcal{T}_{ord}$-theory, and extend Z3 with our $\mathcal{T}_{ord}$-solver. We follow the same strategy as in Reference [59] for loop unrolling. All generated SMT formulas are in the SMT-LIB-v2.6 format. In the following, we call our implementation Zord.[8]

All experiments were conducted on a computer with an Intel(R) Core(TM) i7-8700 CPU and 32 GB DDR4 memory. The operating system is ArchLinux-5.11.10. The time limit for each verification task is 900 seconds.

6.2 Experiment on SV-COMP Benchmarks

We collect benchmarks from the `ConcurrencySafety` category of `SV-COMP 2020`.[9] This category is divided into 11 sub-categories, namely, *ldv-races* (12), *pthread* (38), *atomic* (11), *C-DAC* (4), *complex* (5), *divine* (16), *driver-races* (21), *ext* (53), *lit* (11), *nondet* (6), and *wmm* (898), where the number adhered to each sub-category represents the number of programs it contains. There are 14 programs in *divine* sub-category that cannot be compiled by CBMC and are thus excluded from the benchmark set. In total, we get 1,061 test cases.

We compare Zord with the following tools:

- CBMC[10]: a tool that implements the standard partial-order-based verification algorithms [9], with Z3 as the underlying SMT solver.
- Lazy-CSeq[11]: a tool that verifies concurrent programs under SC memory model using the lazy sequentialization schema [36, 37].
- Lazy-SMA[12]: a tool that verifies concurrent programs under TSO and PSO memory models using the lazy sequentialization schema [37, 54].
- CPA-Seq[13]: a configurable program verification platform [13, 14] with sequentially combined analysis strategies.
- Dartagnan[14]: a bounded model checker [27] for concurrent program verification under various memory models.

Results under SC. The experimental results under SC memory model are summarized in Table 1. The first column *#Solved* shows the number of cases successfully solved by each tool. The following columns list results for both-solved cases (cases that can be correctly verified by both Zord and the tool being compared): *Num* gives the number of cases solved by Zord and the baseline tool; *True* and *False* show the number of solved cases that satisfy and violate the desired properties, respectively; *CPU_time* and *Memory* show the time and memory consumption of the baseline tool and Zord, respectively.

In total, Zord solves 38 more cases than CBMC, 119 more cases than CPA-Seq, and 897 more cases than Dartagnan. The only exception is Lazy-CSeq, which solves 6 more cases than ours. Considering that Lazy-CSeq is a highly optimized tool (winner of the `ConcurrencySafety` category of `SV-COMP` 2020), this result is acceptable. Considering the both-solved cases, Zord is 2.44× faster than CBMC, 90.04× faster than CPA-Seq, 139.47× faster than Dartagnan, and 7.20× faster than Lazy-CSeq. Meanwhile, Zord uses 20.8% less memory than CBMC, 99.6% less memory than CPA-Seq, 99.0% less memory than Dartagnan, and 94.5% less memory than Lazy-CSeq.

We notice that programs in `wmm` sub-category are all very small ones with instrumentations to model the weak memory semantics. One may not consider them as representative concurrent

[8]https://thufv.github.io/research/zord.html.
[9]https://sv-comp.sosy-lab.org/2020/.
[10]CBMC-v5.52.0: https://github.com/diffblue/cbmc/releases/tag/cbmc-5.52.0.
[11]Lazy-CSeq-v2.1: https://github.com/omainv/cseq/releases/tag/SVCOMP2021.
[12]Lazy-SMA: http://users.ecs.soton.ac.uk/gp4/cseq/fmcad16.zip.
[13]CPA-Seq-v2.0: https://gitlab.com/sosy-lab/sv-comp/archives-2021/raw/svcomp21/2021/cpa-seq.zip.
[14]Dartagnan-v2.0.7: https://gitlab.com/sosy-lab/sv-comp/archives-2021/raw/svcomp21/2021/dartagnan.zip.

Table 1. Results on 1,061 SV-COMP Benchmarks under SC

Tool	**#Solved**	**#Both-solved**				
		Num	**True**	**False**	**CPU_time (s)** (-/Zord)	**Memory (GB)** (-/Zord)
Zord	1,051	-	-	-	-	-
CBMC	1,013	1,013	224	789	3,091/1,255	7.87/6.21
CPA-Seq	932	930	165	765	33,771/363	1,321.46/5.41
Dartagnan	154	154	130	24	6,572/51	72.33/0.69
Lazy-CSeq	1,057	1,050	248	802	14,066/1,943	115.45/6.55

Table 2. Results on 163 SV-COMP Benchmarks under SC
(i.e., with wmm Sub-category Excluded)

Tool	**#Solved**	**#Both-solved**				
		Num	**True**	**False**	**CPU_time (s)** (-/Zord)	**Memory (GB)** (-/Zord)
Zord	153	-	-	-	-	-
CBMC	115	115	80	35	2,731/989	1.94/1.53
CPA-Seq	40	38	23	15	1,135/102	30.03/0.77
Dartagnan	31	31	7	24	2,968/21	54.77/0.23
Lazy-CSeq	159	152	104	48	10,528/1,693	72.64/1.66

programs. Table 2 lists the summary results with wmm sub-category excluded. In total, Zord solves 38, 113, and 122 more cases than CBMC, CPA-Seq, and Dartagnan, respectively, and 6 less cases than Lazy-CSeq. Counting on the both-solved non-wmm cases, Zord is 2.76×, 11.06×, 157.47×, and 6.24× faster and uses 21.5%, 97.5%, 99.6%, and 97.7% less memory than CBMC, CPA-Seq, Dartagnan, and Lazy-CSeq, respectively.

Figure 10 compares Zord with CBMC on the CPU time of each verification task. A point below (or above) the diagonal represents a case that Zord is superior (inferior) to CBMC. Programs in wmm sub-category are all simple, so their accumulated CPU time by Zord and CBMC are 269 s and 338 s, respectively; we only draw a single point in the figure to represent the whole wmm sub-category. There is a cluster of points at the bottom left of Figure 10, which indicates that these cases are solved extremely fast by both tools, and CBMC even solves slightly faster on some tasks. This is because either these tasks are trivial or counterexamples occur at a low depth. When the cases become complex, our method starts to show its strength.

Figures 11 and 12 compare Zord with Lazy-CSeq, and Zord with CPA-Seq (blue points) and Dartagnan (orange points) on each case, respectively. These results conform to those in Table 1. Zord is remarkably superior to these three tools in most cases. Among all cases, Zord outperforms Dartagnan. Only in 14 and 3 cases is Zord inferior to Lazy-CSeq and CPA-Seq, respectively.

Results under TSO and PSO. Because CPA-Seq has no configuration for weak memory models, we compare Zord with CBMC, Dartagnan, and Lazy-SMA. The experimental results under TSO and PSO are summarized in Tables 3 and 4, respectively. The meaning of each column is the same as Table 1.

In TSO, Zord solves 47 more cases than CBMC, 925 more cases than Dartagnan, and 533 more cases than Lazy-SMA. Note that the number of cases Lazy-SMA solves under the weak memory model is much less than Lazy-CSeq under SC. This is because Lazy-CSeq's support for the weak memory model (called Lazy-SMA) has not been updated and maintained in the follow-up, and the performance is worse than the competition version (for SC). Considering both-solved cases, Zord is 2.47×, 174.18×, and 4.49× faster than CBMC, Dartagnan, and Lazy-SMA, respectively.

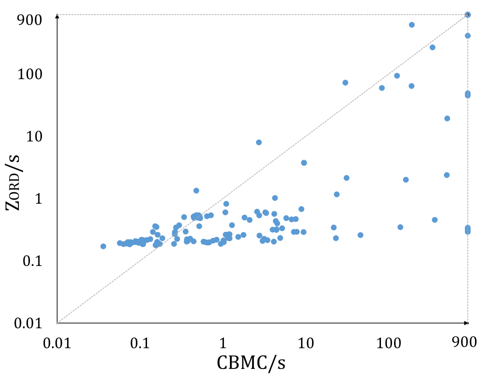

Fig. 10. ZORD vs. CBMC in SC.

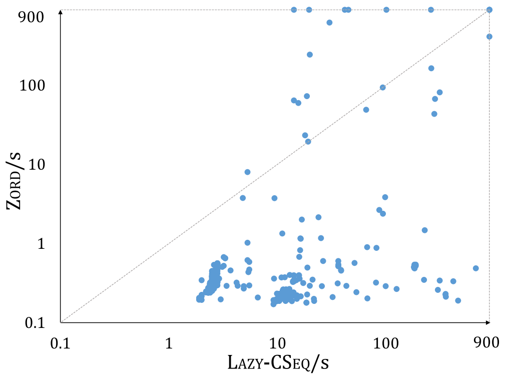

Fig. 11. ZORD vs. LAZY-CSEQ in SC.

Table 3. Summary Results on 1,061 SV-COMP Benchmarks under TSO

Tool	#Solved (TSO)	#Both-solved (TSO)				
		Num	True	False	CPU_time (s) (-/ZORD)	Memory (GB) (-/ZORD)
ZORD	1,048	-	-	-	-	-
CBMC	1,001	1,000	217	783	3,428/1,387	13.55/9.27
DARTAGNAN	123	123	92	31	5,946/37	88.34/1.69
LAZY-SMA	515	515	153	362	992/219	32.53/2.42

Table 4. Summary Results on 1,061 SV-COMP Benchmarks under PSO

Tool	#Solved (PSO)	#Both-solved (PSO)				
		Num	True	False	CPU_time (s) (-/ZORD)	Memory (GB) (-/ZORD)
ZORD	1,049	-	-	-	-	-
CBMC	999	998	97	901	4,576/1,877	11.44/7.93
DARTAGNAN	159	159	15	144	7,689/48	87.63/1.56
LAZY-SMA	776	776	57	719	2,741/245	28.66/1.98

Meanwhile, ZORD uses 31.8%, 98.2%, and 92.6% less memory than CBMC, DARTAGNAN, and LAZY-SMA, respectively.

In PSO, ZORD solves 50, 890, and 273 more cases than CBMC, DARTAGNAN, and LAZY-SMA, respectively. Considering both-solved cases, ZORD is 2.44×, 163.43×, and 11.29× faster than CBMC, DARTAGNAN, and LAZY-SMA, respectively. Meanwhile, ZORD uses 31.0%, 98.4%, and 93.2% less memory than CBMC, DARTAGNAN, and LAZY-SMA, respectively.

From the statistics in Tables 3 and 4, as the memory model changes from SC to TSO and PSO, all the *false* tasks in SC are still *false* in TSO and PSO, whereas some *true* tasks flip to *false*. From the experimental results, relaxing some ordering constraints in TSO and PSO causes more safety property violations, especially when allowing the reordering of two write events that access different addresses.

Tables 5 and 6 list the summary results with wmm sub-category excluded under TSO and PSO, respectively. In total, ZORD solves 33, 114, and 111 more cases in TSO, and 44, 121, and 111 more cases in PSO, than CBMC, DARTAGNAN, and LAZY-SMA, respectively. Counting on both-solved cases, in TSO, our approach is 2.77×, 211.80×, faster and consumes 53.5%, 98.2% less memory than

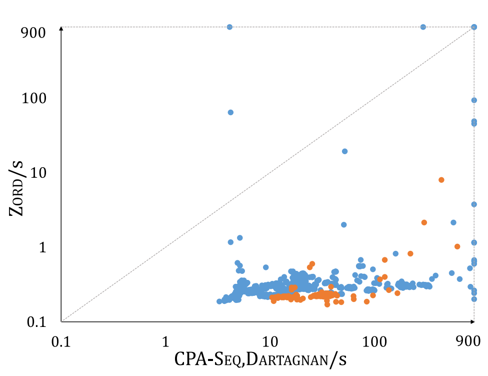

Fig. 12. ZORD vs. CPA-SEQ (blue) and DARTAGNAN (orange) in SC.

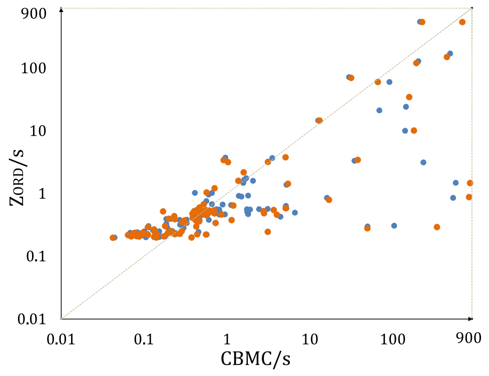

Fig. 13. ZORD vs. CBMC under TSO (blue) and PSO (orange).

Table 5. Summary Results on 163 SV-COMP Benchmarks under TSO (with wmm Sub-category Excluded)

Tool	#Solved (TSO)	#Both-solved (TSO)				
		Num	True	False	CPU_time (s) (-/ZORD)	Memory (GB) (-/ZORD)
ZORD	150	-	-	-	-	-
CBMC	117	116	73	43	3,121/1,119	3.893/1.82
DARTAGNAN	36	36	15	21	3,155/16	68.79/0.55
LAZY-SMA	39	39	27	12	22/29	11.21/0.26

Table 6. Summary Results on 163 SV-COMP Benchmarks under PSO (with wmm Sub-category Excluded)

Tool	#Solved (PSO)	#Both-solved (PSO)				
		Num	True	False	CPU_time (s) (-/ZORD)	Memory (GB) (-/ZORD)
ZORD	151	-	-	-	-	-
CBMC	107	106	54	52	4,339/1,608	1.74/1.60
DARTAGNAN	30	30	15	15	7,656/46	62.88/0.34
LAZY-SMA	40	40	21	19	18/25	9.20/0.21

CBMC and DARTAGNAN; in PSO, our approach runs 2.71×, 212.28× faster and consumes 8.0%, 99.4% less memory than CBMC and DARTAGNAN. Once we exclude the programs in wmm sub-category, LAZY-SMA only solves—39 simple cases in TSO and 40 simple cases in PSO. Considering these both-solved cases , ZORD is slightly inferior to LAZY-SMA. However, ZORD consumes 97.6% and 97.8% less memory than LAZY-SMA.

Figure 13 compares ZORD with CBMC on the CPU time of each verification task under TSO (blue) and PSO (orange). A point below (or above) the diagonal represents a case that ZORD is superior (inferior) to CBMC. Similar to Figure 10, programs in wmm sub-category are all simple so their accumulated CPU time by ZORD and CBMC are—278 s and 333 s in TSO, 272 s and 306 s in PSO, respectively; we only draw a single point for each memory model in the figure to represent the whole wmm sub-category. From Figure 13, our approach is superior to CBMC in most cases, since most of the points are below the diagonal.

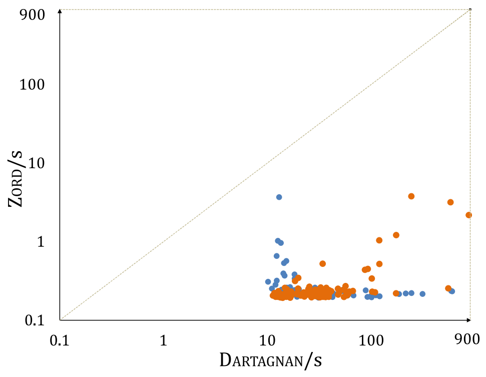

Fig. 14. ZORD vs. DARTAGNAN under TSO (blue) and PSO (orange).

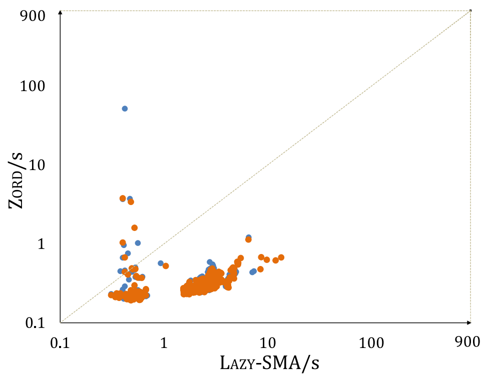

Fig. 15. ZORD vs. LAZY-SMA under TSO (blue) and PSO (orange).

Table 7. Summary Results of FR Propagation

Tool	#Solved	#Both-solved				
		Num	Total_time (s)	Encoding_time (s)	SMT_size (GB)	Solving_time (s)
ZORD⁻	1,049	1,049	3,090.5	642.0	6.53	2,448.4
ZORD	1,051	1,049	2,040.5	491.9	2.28	1,548.6
Ratio	-	-	1.52×	1.31×	2.86×	1.58×

Figures 14 and 15 compare ZORD with DARTAGNAN and LAZY-SMA on each case under TSO (blue) and PSO (orange). These results are consistent with those in Tables 3 and 4. ZORD significantly outperforms DARTAGNAN among all cases and is superior to LAZY-SMA in most cases. In only 7 and 5 cases is ZORD inferior to LAZY-SMA under TSO and PSO, respectively.

6.3 Experiment on Strategies of $\mathcal{T}_{ord}$-solver

This experiment evaluates strategies of $\mathcal{T}_{ord}$-solver by testing their effects on the whole performance of program verification. Experiment settings are identical to 6.2, and we set the memory model to SC for brevity.

Effects of FR Propagation. This experiment compares the effect of the front-end *FR* constraints propagation. ZORD does not encode *FR* constraints into the SMT formula. Instead, the $\mathcal{T}_{ord}$-solver conducts the derivation of *FR* constraints. An alternative strategy, called ZORD$^-$, forces the front-end to encode all *FR* constraints into the SMT formula and lets $\mathcal{T}_{ord}$-solver skip their derivation.

Figure 16 shows the time comparison between ZORD$^-$ and ZORD on each verification task. In most cases, ZORD is more efficient than ZORD$^-$. Table 7 summarizes the comparison result of ZORD$^-$ and ZORD. The columns *Total_time*, *Encoding_time*, and *Solving_time* show the total CPU time, encoding time, and constraint solving time, respectively. The column *SMT_size* lists the size of the generated SMT formulas. In the *Ratio* row, we compute the ratio of ZORD$^-$ to ZORD: A ratio greater than 1.0 indicates ZORD outperforming ZORD$^-$ on the selected measurement.

Among 1,061 available cases, ZORD solves 1,051 cases, and ZORD$^-$ solves 1,049 cases. ZORD$^-$ has two more timeout cases, whereas ZORD solves them within 95.4 s and 248.3 s, respectively. On 1,049 both-solved cases, ZORD$^-$ spends 3,090.5 s while ZORD spends 2,040.5 s—these cases only take ZORD 66.0% time as much as ZORD$^-$. Since ZORD ignores *FR* constraints when generating the SMT formula, ZORD encodes a smaller formula than ZORD$^-$. The size of the encoding formula of ZORD$^-$

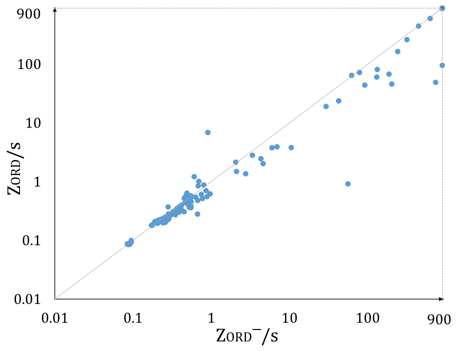

Fig. 16. ZORD vs. ZORD^{-}.

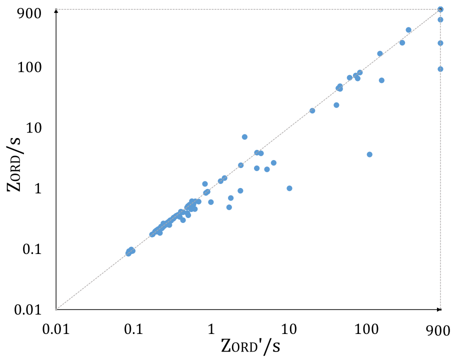

Fig. 17. ZORD vs. ZORD$'$.

and ZORD is 6.53 GB and 2.28 GB, respectively—a 65.1% reduction for ZORD over ZORD^{-}. In more detail, in 439 of 1,061 available cases, the size of generated encoding formula by ZORD is less than 60% of that by ZORD^{-}. Besides, ZORD reduces the encoding time by 150.1 s compared to ZORD^{-}.

Note that the encoding time of ZORD^{-} and ZORD only accounts for 20.8% and 24.1% of total time, respectively. In contrast, constraint solving takes up most of the verification time. Compared to ZORD^{-}, ZORD reduces the solving time by 899.8 s; ZORD shows promising improvement in constraints solving. In more detail, ZORD^{-} uses unit clause propagation at the SAT level to assign *FR* constraints, whereas $\mathcal{T}_{ord}$-solver performs *from-read propagation* on the event graph to derive *FR* constraints with $O(1)$ complexity—our application of the from-read axiom is more efficient than ZORD^{-}. The statistics in Table 7 also show that the overall efficiency improvement mainly comes from the *FR* axiom in constraint solving, not just from the simplified encoding. In summary, it is more efficient to pass the derivation of *FR* constraints to $\mathcal{T}_{ord}$-solver than to encode all FR constraints into the SMT formula.

Effects of Unit-edge Propagation. This experiment evaluates the effect of *unit-edge propagation. Unit-edge propagation* enables $\mathcal{T}_{ord}$-solver to search for unit edges and propagate their *derivation reasons* to *false* as many as possible. Note that *unit-edge propagation* only affects the efficiency of theory propagation but not its correctness.

An alternative strategy is to disable *unit-edge propagation*, signed as ZORD$'$. Figure 17 shows time comparison between ZORD and ZORD$'$ on each verification task. Among 1,061 available cases, ZORD$'$ verifies 1,048 cases correctly, and ZORD succeeds in 1,051 cases—by applying *unit-edge propagation*, we solve 3 more cases within 93.0 s, 606.3 s, and 245.0 s, respectively. On 1,048 both-solved cases, ZORD$'$ spends 1590.2 s and ZORD spends 1389.3 s—the time reduction is 12.6%. Moreover, compared to ZORD$'$, ZORD reduces memory consumption, the number of decisions, propagations, and conflicts to 97.5%, 84.4%, 90.1%, and 79.0%, respectively. ZORD reports 15.6% fewer decisions and 21% fewer conflicts than ZORD$'$. Therefore, we confirm that *unit-edge propagation* helps DPLL(T) avoid possible cycles in advance and make fewer decisions to reduce the search space. In summary, *unit-edge propagation* is an obvious optimization for $\mathcal{T}_{ord}$-solver.

Effects of ICD Algorithm. We also implement Tarjan's non-incremental cycle detection algorithm in our $\mathcal{T}_{ord}$-solver and compare it with the ICD algorithm. Figure 18 shows the CPU time of SMT solving with these two algorithms on each verification task. In small cases (<1 s), the performance is similar, whereas in most complicated cases, the ICD algorithm is more efficient than Tarjan's algorithm. Considering both-solved cases, the total CPU time with Tarjan's algorithm is

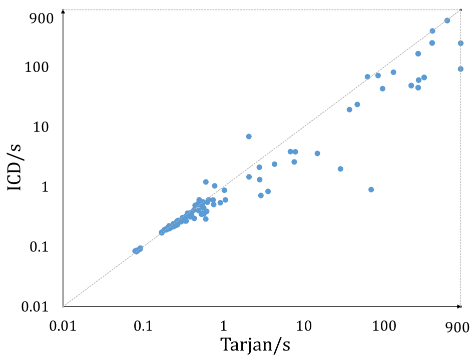

Fig. 18. ICD vs. Tarjan's algorithm.

4813 s, and that with ICD is 2368 s—SMT solving with the ICD algorithm is 2.03× times faster than that with Tarjan's algorithm. For an event graph with n nodes and m edges, the complexity of Tarjan's' algorithm is $O(m \times (n + m))$ and that of ICD is $O(m \times min\{n^{\frac{2}{3}}, m^{\frac{1}{2}}\})$. Generally, the event graph of a multi-threaded program is sparse ($m << n^2$). As programs become complex, incremental cycle detection starts to show its efficacy.

6.4 Comparison with Stateless Model Checking

Stateless model checking **(SMC)** [29] is another successful technique for multi-threaded program verification. It systematically explores all interleaving traces of the concurrent program. Meanwhile, various *partial order reduction* techniques [1, 16, 35, 62] are developed to alleviate the path space explosion problem. In this experiment, we compare Zord with two state-of-the-art SMC tools:

- Nidhugg[15]: an SMC tool that implements the reads-from equivalence exploration algorithm [4] in SC (`rfsc`) and supports various weak memory models [2, 3].
- GenMC[16]: an SMC tool that implements an optimal algorithm to build execution graphs dynamically [40]. GenMC supports various access modes in **Repaired C/C++11 memory model (RC11)** [39].

Originally, we wanted to perform this experiment on `SV-COMP` benchmarks. However, Nidhugg and GenMC currently do not support many library functions used in `SV-COMP` benchmarks, e.g., `abort`(), `strcmp`(), `strcpy`(), and some of `__VERIFIER_*`. As a result, among the 1,061 SV-COMP verification tasks, Nidhugg and GenMC only solve 45 and 38 cases, respectively.

Therefore, we decide to use benchmarks from Nidhugg.[17] The original benchmark set contains 83 multi-threaded C programs, many of which do not contain assertions, since they are prepared for testing ***Partial Order Reduction*** **(POR)** techniques [1, 2]. We select benchmarks using the following rules: (1) gcc-compilable, (2) contain at least one assertion, (3) parameterizable, and (4) verifiable by Nidhugg.

Finally, we get nine examples with parameter N:

- `CO-2+2W`(N) and `float_r`(N) are two examples that check the safety of atomic read and atomic write in a multi-threaded environment with N threads.

[15]Nidhugg-rfsc: https://github.com/nidhugg/nidhugg/tree/master/benchmarks.
[16]GenMC v8.0 https://github.com/MPI-SWS/genmc/releases/tag/v0.8.
[17]Nidhugg benchmarks: https://github.com/nidhugg/nidhugg/tree/master/benchmarks,commit6af46b8.

- `airline`(N) implements a ticket sale example from Reference [35], where N is the number threads.
- `fib_bench`(N) implements a concurrent version of Fibonacci number; `szymanski`(N) implements Szymaski's critical section algorithm; `lamport`(N) implements the Lamport's algorithm; `cir_buf`(N) implements a circular buffer. The above all four programs using two threads, where N determines their unrolling bound of loops.
- `parker`(N) implements the Parker lock algorithm using one thread. N is used in a `assume` statement.
- `account`(N) depicts a bank account system [35]. It contains N deposit threads and one withdraw thread.

Given that Zord is implemented on top of CBMC, we also make a comparison with CBMC on Nidhugg benchmarks. In general, we compare Zord with CBMC and Nidhugg under SC, TSO, and PSO, respectively. Moreover, since GenMC does not currently support TSO and PSO, we make a comparison with GenMC[18] under SC.

Results. Table 8 details the experimental results, where the *Traces* column lists the number of traces explored by GenMC. There is an obvious positive correlation between the time efficiency of stateless model checking tool and the number of program paths it searches. If a verification result is returned within 900 seconds, then the number reported in *Traces* is a measurement of the size of the trace space. For each verification tool (e.g., Nidhugg), we show its verification time under SC, TSO, and PSO, respectively. The abbreviation TO means timeout, i.e., no verification result returned within 900 seconds.

In `CO-2+2W`(N) and `float_r`(N), there is no branching statement; the `main` thread contains only one read event; and each child thread contains one visible atomic write. As a result, the number of traces in each program is equal (or nearly equal) to the number of child threads (i.e., N). Compared to SMC tools, Nidhugg is efficient under SC, since it employs the `rfsc` strategy to reduce equivalent traces. However, as N increases, Nidhugg fails to verify these programs under TSO and PSO. In contrast, Zord is obviously faster than Nidhugg under TSO and PSO. GenMC verifies these programs within a few tenths of a second. This is understandable, since the largest `float_r(100)` contains 101 traces only. GenMC only explores $N + c$ (c is a non-negative integer and $c \ll N$) traces and returns instantly. The time consumption of CBMC and Zord increase gradually on the increment of N because the size of the encoding formula grows accordingly. However, Zord is obviously faster than CBMC under both SC, TSO, and PSO. Our ordering theory and elaborated theory solver help Zord achieve higher efficiency.

`Airline`(N), `fib_bench`(N), and `szymanski`(N) contain branching statements, with N representing either the number of threads or the unrolling bound of loops. As N increases, Nidhugg and GenMC slow down rapidly. Basically, the verification time of Nidhugg and GenMC is proportional to the number of traces in the program. From Table 8, Zord is significantly superior to CBMC, Nidhugg, and GenMC.

In `lamport`(N) and `cir_buffer`(N), Zord is slightly inferior to Nidhugg and GenMC. The main reason is that these two examples contain numerous array operations. As a result, the array theory solver is involved in SMT solving, which is not as efficient as the bit-vector solver. Nevertheless, Zord is the only tool that solves `cir_buffer(13)` under SC.

In `parker`(N), a `_parker` procedure is invoked N times. As N increases, Nidhugg and GenMC slow down rapidly, in line with the fast growth of traces. In contrast, Zord verifies each program

[18]GenMC has two configurations: *modification order* (MO) and *writes-before* (WB). The latter is much superior to the former [40]. In our experiments, GenMC is configured using WB.

Table 8. Experiment Results on Nidhugg Benchmarks

Files	Traces	Nidhugg			GenMC	CBMC			Zord		
		SC	TSO	PSO	SC	SC	TSO	PSO	SC	TSO	PSO
`CO-2+2W(5)`	5	0.10	13.22	13.86	0.02	0.35	6.04	5.89	0.31	0.22	0.17
`CO-2+2W(15)`	15	0.12	TO	TO	0.03	33.89	494.10	557.24	1.35	1.35	1.34
`CO-2+2W(25)`	25	0.10	TO	TO	0.04	322.61	TO	TO	6.57	7.31	7.29
`float_r(10)`	11	0.08	TO	TO	0.03	3.182	287.63	344.37	0.21	0.18	0.14
`float_r(50)`	51	0.11	TO	TO	0.04	TO	TO	TO	1.68	2.12	2.44
`float_r(100)`	101	0.30	TO	TO	0.10	TO	TO	TO	4.90	TO	TO
`airline(3)`	27	0.09	2.23	2.36	0.03	0.19	0.21	0.21	0.22	0.22	0.22
`airline(7)`	8.2×10^5	161.21	TO	TO	28.03	3.73	51.56	40.32	0.32	0.36	0.24
`airline(9)`	-	TO	TO	TO	TO	20.03	245.40	114.06	0.51	0.41	0.25
`fib_bench(4)`	3.4×10^4	2.74	1.18	1.59	0.72	31.02	112.36	121.62	0.35	0.35	0.38
`fib_bench(5)`	5.3×10^5	37.53	21.21	18.80	12.98	88.29	558.70	511.04	0.82	0.61	1.17
`fib_bench(6)`	8.1×10^6	537.98	283.31	242.18	264.65	TO	TO	TO	1.53	1.66	1.55
`szymanski(2)`	-	4.07	2.96	2.73	TO	15.41	13.13	7.81	1.69	1.38	1.29
`szymanski(4)`	-	153.92	106.65	92.96	TO	258.15	29.65	16.62	6.14	5.11	4.77
`szymanski(6)`	-	TO	TO	TO	TO	792.42	63.34	61.62	12.52	7.89	8.63
`lamport(2)`	1.7×10^4	2.46	3.82	3.48	1.18	TO	TO	TO	4.15	2.09	1.33
`lamport(6)`	1.3×10^6	282.10	286.89	256.74	65.42	TO	TO	TO	379.21	163.39	139.99
`lamport(10)`	-	TO	TO	TO	TO	TO	TO	TO	TO	TO	TO
`cir_buf(5)`	252	0.19	0.16	0.18	0.06	11.19	27.97	35.78	1.70	0.17	0.18
`cir_buf(9)`	4.9×10^4	25.90	16.69	18.33	11.84	176.43	213.89	TO	52.73	69.10	62.54
`cir_buf(13)`	-	TO	TO	TO	TO	TO	TO	TO	897.44	TO	TO
`parker(12)`	7.0×10^4	2.94	2.05	1.99	15.58	114.63	125.47	114.41	0.33	0.35	0.32
`parker(20)`	4.8×10^5	19.21	11.75	10.98	89.56	TO	TO	TO	0.36	0.35	0.37
`parker(28)`	1.7×10^6	52.60	38.21	36.50	458.29	TO	TO	TO	0.38	0.37	0.36
`account(5)`	1	0.01	0.01	0.09	0.01	0.67	0.72	0.66	0.25	3.34	0.05
`account(15)`	1	0.01	0.09	0.09	0.02	87.14	688.33	843.31	43.30	98.28	6.44
`account(25)`	1	0.13	0.34	0.08	0.04	TO	TO	TO	TO	TO	TO

in less than 0.4 second, and the time does not notably change as N increases. The main reasons are: (1) this example contains only one thread, and (2) Zord encodes the path condition of each memory event into the SMT formula. As a result, increasing N has no influence on the size of the SMT formula. Moreover, Zord significantly outperforms CBMC under three memory models.

In program `account(`N`)`, Nidhugg obviously outperforms others. Note that all variations of this example are buggy. It happens that Nidhugg finds the safety violation by exploring one trace. In `account(5)`, the performance of Zord, CBMC, and GenMC is similar; but as N increases to 15, Zord is obviously faster than CBMC and GenMC.

Summary. First, as N increases, Zord outperforms CBMC significantly, demonstrating the efficiency of our $\mathcal{T}_{ord}$-theory solver. Second, SMC is suitable for programs with simple branches, while Zord is more suitable for complex programs. Third, arrays and complex data structures can slow down the SMT solving. In general, as the scale (measured by the number of traces) of the program increases, our approach is superior to these tools in most cases.

6.5 Threats to Validity and Limitations

The main threats to validity are whether the performance improvements are due to our tactic and whether our implementation and experiments are credible.

First, since we added support for $\mathcal{T}_{ord}$ in CBMC and elaborated on its theory solver in Z3, we compare Zord with them instantly. The improvements over CBMC must come from our tactic. Second, the experimental results of *from-read generation* and *unit-edge propagation* are consistent with the theoretical analysis, which confirms that the improvements are indeed from these strategies. Third, we compare the ICD algorithm to Tarjan's SCC algorithm during theory solving. The above three aspects show that the performance improvements are due to our tactic.

Implementation of $\mathcal{T}_{ord}$ in CBMC is simple and clear, and the implementation of $\mathcal{T}_{ord}$-solver is loosely coupled with the overall framework of Z3. Benchmarks are collected from the `ConcurrencySafety` category of `SV-COMP` 2020 and Nidhugg. Many studies perform their experiments on these benchmarks to demonstrate the effectiveness of their method. Finally, we performed an extensive comparison with CBMC, Lazy-CSeq, Lazy-SMA, CPA-Seq, Dartagnan, and two SMC techniques implemented in Nidhugg and GenMC. The detailed results and analysis of these tools show that Zord is correct and competitive. We are thus confident in the effectiveness of Zord.

Another threat to the validity is whether our approach can be generalized to other SMT solvers than Z3. We model multi-threaded programs and encode them into SMT formulas, which are independent of the SMT solver. If we follow the axioms of multi-threaded program verification and implement these rules into other SMT solvers, then they are also suitable for solving these SMT formulas. Therefore, our approach can be implemented in other DPLL(T)-based SMT solvers.

The main limitations of our approach are summarized below. First, our method is designed for verifying safety properties. We need to support more modeling techniques and elaborate on domain-specific solving strategies for other concurrent verification problems (e.g., data race detection). Second, our method can not handle the different memory order modes for atomic access and *acquire/release* operations. Third, our method only supports SC, TSO, and PSO memory models. Other mainstream weak memory models with richer semantics and operations need to be supported in the future. Finally, we use CBMC's original pointer analysis component. It is developed based on the classical Steensgaard-style algorithm [52], which is flow-insensitive and is applicable for symbolic encoding in TSO and PSO. However, the may-alias analysis in handling pointers may cause spurious counterexamples. Therefore, more advanced strategies are required for analyzing pointers in multi-threaded programs.

7 RELATED WORK

There is much research on improving the availability and efficiency of multi-threaded program verification by constraint solving. In this section, we discuss representative and related techniques in recent years.

Multi-threaded program verification has been comprehensively studied in recent years. Alglave et al. [9, 10] use a partial order relation on memory events to represent possible executions caused by thread interleaving. All partial order constraints are encoded into a formula. If no counterexample is found, then the multi-threaded program is safe. Otherwise, we need to check that the counterexample is not spurious, i.e., that the corresponding execution does not form a cycle. The above technique inspires our method. We propose a new theory to denote order constraints of memory events. Then, we elaborate on the theory solver of $\mathcal{T}_{ord}$ in Z3 [21]. It can benefit from optimizations of Z3 inherently and is fully compatible with DPLL(T) framework. By utilizing axioms of $\mathcal{T}_{ord}$, $\mathcal{T}_{ord}$-solver is responsible for verifying whether the current order constraints are valid, i.e., whether the current order is acyclic. In general, we develop a new partial order theory to extend the basic idea of References [9, 10] with SMT-based constraint solving.

Horn et al. [33, 34] provide a different approach for representing and solving partial order constraints. They propose a novel *partial-string theory* where order constraints are represented as *partial strings* and operators on *partial strings* are defined accordingly.

$\mathcal{T}_{ord}$-solver's decision procedure is closely related to **incremental cycle detection (ICD)**, a mature algorithm with a long research history. Based on online topological ordering [41], research on ICD begins with Reference [43]. When an edge is inserted, it considers updating the topological order by reusing the previous order instead of calculating a new one. The authors propose a method that takes amortized $O(n)$ time for each edge insertion in a graph with n nodes, faster than the $O(n + m)$ offline algorithm (with m representing the number of edges). Later works cited in References [6, 7, 12, 31, 38, 48] aim at proposing new cycle detection algorithms; some work better on sparse graphs (assuming $m = O(n)$) and some better on dense graphs (assuming $m = O(n^2)$).

To the best of our knowledge, the fastest algorithms are proposed by Bender et al. [12]. They propose two $O(min\{m^{1/2}, n^{2/3}\}m)$ algorithms for sparse graphs and one $O(n^2 log(n))$ algorithm for dense graphs. Among them, we apply the two-way-search algorithm for sparse graphs using pseudo-topological order in our approach. This algorithm achieves $O(min\{m^{1/2}, n^{2/3}\})$ for each edge insertion by using pseudo-topological order and setting a well-designed limit to the order. However, it can only find a strongly connected component when a cycle occurs. Therefore, we extend the algorithm to search for all critical cycles with the shortest width.

Nieuwenhuis et al. [46] design an SMT implementation of difference logic, where literal $x - y \leq k$ is represented with an edge from x to y with weight k and $x - y < k$ represented with an edge with weight $k - \epsilon$ where $\epsilon = 1$ in integer difference logic and ϵ can be computed in linear time in real difference logic [49]. After adding an edge from x to y, a traverse of outgoing edges from y and a traverse of incoming edges from x are performed to check whether this edge addition forms a cycle. If a cycle on which weights of edges sum negative is found, then an inconsistency occurs. Their difference logic solver's behavior after each edge addition is similar to ours but far from incremental. It does not maintain a topological order, so it cannot make use of previous information. Thus, each traverse is exhaustive and takes $O(n + m)$ time in a graph with n nodes and m edges.

Ge et al. [28] propose another method to solve order constraints. They only define one sort of partial order relation, i.e., the "happens-before" relation, and unite other sorts of partial order relations to "happens-before." Then, they check if there is a cycle among ordering constraints. In this manner, their theory cannot handle the intricate relation between different sorts of partial orders for concurrent program verification, which, in our understanding, is very important for the efficiency of SMT solving. In comparison, our method can model various order relations to formulate a multi-threaded program's possible executions. After each edge addition, their method calls Tarjan's algorithm to find cycles. We notice the high frequency of edge addition and deletion while solving an SMT formula with order constraints. Instead, we develop a new **ordering consistency (OC)** theory specifically for multi-threaded program verification and elaborate on its theory solver with incremental cycle detection to achieve higher efficiency. Using incremental cycle detection, m additions in a graph with n nodes take only $O(m \times min\{n^{\frac{2}{3}}, m^{\frac{1}{2}}\})$ time, better than Tarjan's algorithm that spends $O(m \times (n + m))$.

Multi-threaded program verification is complicated because of the uncertainty caused by thread interleaving. Too many possible execution paths may result in the state explosion problem. The most efficient techniques to alleviate this problem include but are not limited to bounded model checking [15, 17, 25, 27, 55], partial order reduction [1, 24, 57], abstraction refinement [23, 30, 51], and stateless model checking [2, 3, 39].

Moreover, to improve the availability of multi-threaded program verification, many researchers combine BMC with other techniques. Inverso et al. [37] propose a new approach named *Lazy Sequentialization*, which transforms a multi-threaded program into a sequential program under a specific unrolling bound and execution round. This method simulates thread interactions and is efficient in bug-finding. Yin et al. [59, 61] develop a SAT-based verification framework called **scheduling constraints-based abstraction refinement (SCAR)** for verifying multi-threaded programs under several memory models. They ignore the order constraints at first; instead, they add conflict clauses in the iterations of abstraction refinement to enhance the formula. They use the transitive closure to check the consistency of order relation, which achieves high efficiency. Cordeiro et al. [18] combine BMC with **Satisfiable Modulo Theory (SMT)**. They encode all possible executions and utilize theory conflict to abstract thread interleaving. They implemented their tactics in ESCBMC, an SMT-based verification tool.

Stateless model checking (SMC) [29] checks correctness by enumerating all possible execution traces. Since thread interleaving results in numerous traces, traces inducing the same order between conflicting events are sorted into an equivalence class, namely, a Mazurkiewicz trace [44]. Mazurkiewicz traces can be further weakened to equivalence classes of traces with identical read-from relations [4]. Efficient traversal algorithms are developed to discover all Mazurkiewicz traces consistent with the memory model and prune inconsistent ones. Popular stateless model checkers, e.g., Nidhugg [2], GenMC [40], RCMC [39], DC-DPOR [16], are based on this technique.

Weak memory models are another issue for multi-threaded program verification, since they bring intra-threaded uncertainty, making concurrent program verification even harder. Lots of groundbreaking research extends their approach designed for SC to weak memory models. Tomasco [54] extends *Lazy Sequentialization* to TSO and PSO (called Lazy-SMA). They replace memory access with operations on a shared memory abstraction and verify their validity under SC. Yin [60] extends their SCAR technique under SC to TSO and PSO by relaxing ordering constraints between neighboring events. Gavrilenko et al. [27] combine BMC with relation analysis and employ the backend SMT solver to verify concurrent programs under weak memory models. In this article, we propose a new modeling technique and extend our $\mathcal{T}_{ord}$-theory and $\mathcal{T}_{ord}$-solver to weak memory models (e.g., TSO, PSO). We performed extensive experiments with CBMC, Lazy-SMA, Dartagnan, and several state-of-the-art SMC tools (e.g., Nidhugg, GenMC) under various memory models. The experimental results illustrate that our approach is competitive and efficient.

8 CONCLUSION AND FUTURE WORK

In this article, we presented a novel SMT-based approach for verifying multi-threaded programs. We proposed a dedicated ordering consistency theory for multi-threaded program verification under SC, TSO, and PSO. We also elaborated on its theory solver, which realizes incremental consistency checking, minimal conflict clause generation, and specialized theory propagation to improve the efficiency of SMT solving. We implemented our techniques on top of CBMC and Z3 and conducted experiments on `SV-COMP` benchmarks and `Nidhugg` benchmarks to evaluate its effectiveness and efficiency. The experimental results show that our approach has significant improvements over the state-of-the-art verification techniques.

Our approach is designed for multi-threaded program verification using SMT. We are planning to extend our approach to SAT-based framework in the future.

REFERENCES

[1] Parosh Abdulla, Stavros Aronis, Bengt Jonsson, and Konstantinos Sagonas. 2014. Optimal dynamic partial order reduction. In *Proceedings of the 41st ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'14)*. Association for Computing Machinery, New York, NY, 373–384. DOI: https://doi.org/10.1145/2535838.2535845

[2] Parosh Aziz Abdulla, Stavros Aronis, Mohamed Faouzi Atig, Bengt Jonsson, Carl Leonardsson, and Konstantinos Sagonas. 2015. Stateless model checking for TSO and PSO. In *Tools and Algorithms for the Construction and Analysis of Systems*, Christel Baier and Cesare Tinelli (Eds.). Springer Berlin, 353–367. DOI: https://doi.org/10.1007/978-3-662-46681-0_28

[3] Parosh Aziz Abdulla, Mohamed Faouzi Atig, Bengt Jonsson, and Carl Leonardsson. 2016. Stateless model checking for POWER. In *Computer Aided Verification*, Swarat Chaudhuri and Azadeh Farzan (Eds.). Springer International Publishing, Cham, 134–156. DOI: https://doi.org/10.1007/978-3-319-41540-6_8

[4] Parosh Aziz Abdulla, Mohamed Faouzi Atig, Bengt Jonsson, Magnus Lång, Tuan Phong Ngo, and Konstantinos Sagonas. 2019. Optimal stateless model checking for reads-from equivalence under sequential consistency. *Proc. ACM Program. Lang.* 3, OOPSLA (Oct. 2019), 29 pages. DOI: https://doi.org/10.1145/3360576

[5] Sarita V. Adve and Kourosh Gharachorloo. 1996. Shared memory consistency models: A tutorial. *Computer* 29, 12 (1996), 66–76. DOI: https://doi.org/10.1109/2.546611
[6] Deepak Ajwani and Tobias Friedrich. 2007. Average-case analysis of online topological ordering. In *Proceedings of the 18th International Conference on Algorithms and Computation (ISAAC'07)*. Springer-Verlag, Berlin, 464–475. DOI: https://doi.org/10.1007/978-3-540-77120-3_41
[7] Deepak Ajwani, Tobias Friedrich, and Ulrich Meyer. 2008. An O(N2.75) algorithm for incremental topological ordering. *ACM Trans. Algor.* 4, 4 (Aug. 2008). DOI: https://doi.org/10.1145/1383369.1383370
[8] Jade Alglave, Daniel Kroening, Vincent Nimal, and Daniel Poetzl. 2014. Don't sit on the fence. In *Computer Aided Verification*, Armin Biere and Roderick Bloem (Eds.). Springer International Publishing, Cham, 508–524. DOI: https://doi.org/10.1007/978-3-319-08867-9_33
[9] Jade Alglave, Daniel Kroening, and Michael Tautschnig. 2013. Partial orders for efficient bounded model checking of concurrent software. In *Computer Aided Verification*, Natasha Sharygina and Helmut Veith (Eds.). Springer Berlin, 141–157. DOI: https://doi.org/10.1007/978-3-642-39799-8_9
[10] Jade Alglave, Luc Maranget, Susmit Sarkar, and Peter Sewell. 2010. Fences in weak memory models. In *Computer Aided Verification*, Tayssir Touili, Byron Cook, and Paul Jackson (Eds.). Springer Berlin, 258–272. DOI: https://doi.org/10.1007/978-3-642-14295-6_25
[11] Clark Barrett and Cesare Tinelli. 2018. *Satisfiability Modulo Theories*. Springer International Publishing, Cham, 305–343. DOI: https://doi.org/10.1007/978-3-319-10575-8_11
[12] Michael A. Bender, Jeremy T. Fineman, Seth Gilbert, and Robert E. Tarjan. 2015. A new approach to incremental cycle detection and related problems. *ACM Trans. Algor.* 12, 2 (Dec. 2015). DOI: https://doi.org/10.1145/2756553
[13] Dirk Beyer, Thomas A. Henzinger, and Grégory Théoduloz. 2007. Configurable software verification: Concretizing the convergence of model checking and program analysis. In *Computer Aided Verification*, Werner Damm and Holger Hermanns (Eds.). Springer Berlin, 504–518. DOI: https://doi.org/10.1007/978-3-540-73368-3_51
[14] Dirk Beyer and M. Erkan Keremoglu. 2011. CPAchecker: A tool for configurable software verification. In *Computer Aided Verification*, Ganesh Gopalakrishnan and Shaz Qadeer (Eds.). Springer Berlin, 184–190. DOI: https://doi.org/10.1007/978-3-642-22110-1_16
[15] Armin Biere, Alessandro Cimatti, Edmund Clarke, and Yunshan Zhu. 1999. Symbolic model checking without BDDs. In *Tools and Algorithms for the Construction and Analysis of Systems*, W. Rance Cleaveland (Ed.). Springer Berlin, 193–207. DOI: https://doi.org/10.1007/3-540-49059-0_14
[16] Marek Chalupa, Krishnendu Chatterjee, Andreas Pavlogiannis, Nishant Sinha, and Kapil Vaidya. 2017. Data-centric dynamic partial order reduction. *Proc. ACM Program. Lang.* 2, POPL (Dec. 2017). DOI: https://doi.org/10.1145/3158119
[17] Edmund Clarke, Armin Biere, Richard Raimi, and Yunshan Zhu. 2001. Bounded model checking using satisfiability solving. *Form. Methods Syst. Des.* 19, 1 (July 2001), 7–34. DOI: https://doi.org/10.1023/A:1011276507260
[18] Lucas Cordeiro and Bernd Fischer. 2011. Verifying multi-threaded software using SMT-based context-bounded model checking. In *Proceedings of the 33rd International Conference on Software Engineering (ICSE'11)*. Association for Computing Machinery, New York, NY, 331–340. DOI: https://doi.org/10.1145/1985793.1985839
[19] Ron Cytron, Jeanne Ferrante, Barry K. Rosen, Mark N. Wegman, and F. Kenneth Zadeck. 1991. Efficiently computing static single assignment form and the control dependence graph. *ACM Trans. Program. Lang. Syst.* 13 (1991), 451–490. DOI: https://doi.org/10.1145/115372.115320
[20] Martin Davis, George Logemann, and Donald Loveland. 1962. A machine program for theorem-proving. *Commun. ACM* 5, 7 (July 1962), 394–397. DOI: https://doi.org/10.1145/368273.368557
[21] Leonardo de Moura and Nikolaj Bjørner. 2008. Z3: An efficient SMT solver. In *Tools and Algorithms for the Construction and Analysis of Systems*, C. R. Ramakrishnan and Jakob Rehof (Eds.). Springer Berlin, 337–340. DOI: https://doi.org/10.1007/978-3-540-78800-3_24
[22] Leonardo De Moura and Nikolaj Bjørner. 2011. Satisfiability modulo theories: Introduction and applications. *Commun. ACM* 54, 9 (Sept. 2011), 69–77. DOI: https://doi.org/10.1145/1995376.1995394
[23] Thomas Dinsdale-Young, Mike Dodds, Philippa Gardner, Matthew J. Parkinson, and Viktor Vafeiadis. 2010. Concurrent abstract predicates. In *ECOOP 2010—Object-Oriented Programming*, Theo D'Hondt (Ed.). Springer Berlin, 504–528. DOI: https://doi.org/10.1007/978-3-642-14107-2_24
[24] Cormac Flanagan and Patrice Godefroid. 2005. Dynamic partial-order reduction for model checking software. In *Proceedings of the 32nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'05)*. Association for Computing Machinery, New York, NY, 110–121. DOI: https://doi.org/10.1145/1040305.1040315
[25] Malay K. Ganai and Aarti Gupta. 2008. Efficient modeling of concurrent systems in BMC. In *Model Checking Software*, Klaus Havelund, Rupak Majumdar, and Jens Palsberg (Eds.). Springer Berlin, 114–133. DOI: https://doi.org/10.1007/978-3-540-85114-1_10
[26] Harald Ganzinger, George Hagen, Robert Nieuwenhuis, Albert Oliveras, and Cesare Tinelli. 2004. DPLL(T): Fast decision procedures. In *Computer Aided Verification*, Rajeev Alur and Doron A. Peled (Eds.). Springer Berlin, 175–188.

[27] Natalia Gavrilenko, Hernán Ponce-de León, Florian Furbach, Keijo Heljanko, and Roland Meyer. 2019. BMC for weak memory models: Relation analysis for compact SMT encodings. In *Computer Aided Verification*, Isil Dillig and Serdar Tasiran (Eds.). Springer International Publishing, Cham, 355–365. DOI: https://doi.org/10.1007/978-3-030-25540-4_19

[28] Cunjing Ge, Feifei Ma, Jeff Huang, and Jian Zhang. 2016. SMT solving for the theory of ordering constraints. In *Languages and Compilers for Parallel Computing*, Xipeng Shen, Frank Mueller, and James Tuck (Eds.). Springer International Publishing, Cham, 287–302. DOI: https://doi.org/10.1007/978-3-319-29778-1_18

[29] Patrice Godefroid. 1997. Model checking for programming languages using VeriSoft. In *Proceedings of the 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'97)*. Association for Computing Machinery, New York, NY, 174–186. DOI: https://doi.org/10.1145/263699.263717

[30] Ashutosh Gupta, Corneliu Popeea, and Andrey Rybalchenko. 2011. Predicate abstraction and refinement for verifying multi-threaded programs. In *Proceedings of the 38th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'11)*. Association for Computing Machinery, New York, NY, 331–344. DOI: https://doi.org/10.1145/1926385.1926424

[31] Bernhard Haeupler, Telikepalli Kavitha, Rogers Mathew, Siddhartha Sen, and Robert E. Tarjan. 2012. Incremental cycle detection, topological ordering, and strong component maintenance. *ACM Trans. Algor.* 8, 1 (Jan. 2012). DOI: https://doi.org/10.1145/2071379.2071382

[32] Fei He, Zhihang Sun, and Hongyu Fan. 2021. Satisfiability modulo ordering consistency theory for multi-threaded program verification. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation (PLDI'21)*. Association for Computing Machinery, New York, NY, 1264–1279. DOI: https://doi.org/10.1145/3453483.3454108

[33] Alex Horn and Jade Alglave. 2014. Concurrent Kleene Algebra of Partial Strings. arXiv:1407.0385 [cs.LO].

[34] Alex Horn and Daniel Kroening. 2015. On Partial Order Semantics for SAT/SMT-based Symbolic Encodings of Weak Memory Concurrency. arXiv:1504.00037 [cs.LO].

[35] Jeff Huang. 2015. Stateless model checking concurrent programs with maximal causality reduction. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'15)*. Association for Computing Machinery, New York, NY, 165–174. DOI: https://doi.org/10.1145/2737924.2737975

[36] Omar Inverso, Truc L. Nguyen, Bernd Fischer, Salvatore La Torre, and Gennaro Parlato. 2015. Lazy-CSeq: A context-bounded model checking tool for multi-threaded C-Programs. In *Proceedings of the 30th IEEE/ACM International Conference on Automated Software Engineering (ASE'15)*. IEEE Press, 807–812. DOI: https://doi.org/10.1109/ASE.2015.108

[37] Omar Inverso, Ermenegildo Tomasco, Bernd Fischer, Salvatore La Torre, and Gennaro Parlato. 2014. Bounded model checking of multi-threaded C programs via lazy sequentialization. In *Computer Aided Verification*, Armin Biere and Roderick Bloem (Eds.). Springer International Publishing, Cham, 585–602. DOI: https://doi.org/10.1007/978-3-319-08867-9_39

[38] Irit Katriel and Hans L. Bodlaender. 2006. Online topological ordering. *ACM Trans. Algor.* 2, 3 (July 2006), 364–379. DOI: https://doi.org/10.1145/1159892.1159896

[39] Michalis Kokologiannakis, Ori Lahav, Konstantinos Sagonas, and Viktor Vafeiadis. 2017. Effective stateless model checking for C/C++ concurrency. *Proc. ACM Program. Lang.* 2, POPL (Dec. 2017). DOI: https://doi.org/10.1145/3158105

[40] Michalis Kokologiannakis, Azalea Raad, and Viktor Vafeiadis. 2019. Model checking for weakly consistent libraries. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'19)*. Association for Computing Machinery, New York, NY, 96–110. DOI: https://doi.org/10.1145/3314221.3314609

[41] Daniel Kroening and Michael Tautschnig. 2014. CBMC—C bounded model checker. In *Tools and Algorithms for the Construction and Analysis of Systems*, Erika Ábrahám and Klaus Havelund (Eds.). Springer Berlin, 389–391. DOI: https://doi.org/10.1007/978-3-642-54862-8_26

[42] Leslie Lamport. 1979. How to make a multiprocessor computer that correctly executes multiprocess programs. *IEEE Comput. Archit. Lett.* 28, 9 (1979), 690–691. DOI: http://doi.org/10.1109/TC.1979.1675439

[43] Alberto Marchetti-Spaccamela, Umberto Nanni, and Hans Rohnert. 1996. Maintaining a topological order under edge insertions. *Inform. Process. Lett.* 59, 1 (1996), 53–58. DOI: https://doi.org/10.1016/0020-0190(96)00075-0

[44] Antoni Mazurkiewicz. 1987. Trace theory. In *Petri Nets: Applications and Relationships to Other Models of Concurrency*, W. Brauer, W. Reisig, and G. Rozenberg (Eds.). Springer Berlin, 278–324.

[45] Matthew W. Moskewicz, Conor F. Madigan, Ying Zhao, Lintao Zhang, and Sharad Malik. 2001. Chaff: Engineering an efficient SAT solver. In *Proceedings of the 38th Annual Design Automation Conference (DAC'01)*. Association for Computing Machinery, New York, NY, 530–535. DOI: https://doi.org/10.1145/378239.379017

[46] Robert Nieuwenhuis, Albert Oliveras, and Cesare Tinelli. 2006. Solving SAT and SAT modulo theories: From an abstract Davis-Putnam-Logemann-Loveland procedure to DPLL(T). *J. ACM* 53, 6 (Nov. 2006), 937–977. DOI: https://doi.org/10.1145/1217856.1217859

[47] Scott Owens, Susmit Sarkar, and Peter Sewell. 2009. A better X86 memory model: X86-TSO. In *Proceedings of the 22nd International Conference on Theorem Proving in Higher Order Logics (TPHOLs'09)*. Springer-Verlag, Berlin, 391–407. DOI: https://doi.org/10.1007/978-3-642-03359-9_27

[48] David J. Pearce and Paul H. J. Kelly. 2007. A dynamic topological sort algorithm for directed acyclic graphs. *ACM J. Exp. Algor.* 11 (Feb. 2007), 1.7–es. DOI: https://doi.org/10.1145/1187436.1210590
[49] Alexander Schrijver. 1986. *Theory of Linear and Integer Programming*. John Wiley & Sons, Inc..
[50] Dennis Shasha and Marc Snir. 1988. Efficient and correct execution of parallel programs that share memory. *ACM Trans. Program. Lang. Syst.* 10, 2 (Apr. 1988), 282–312. DOI: https://doi.org/10.1145/42190.42277
[51] Nishant Sinha and Chao Wang. 2011. On interference abstractions. In *Proceedings of the 38th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'11)*. Association for Computing Machinery, New York, NY, 423–434. DOI: https://doi.org/10.1145/1926385.1926433
[52] Bjarne Steensgaard. 1996. Points-to analysis in almost linear time. In *Proceedings of the 23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'96)*. Association for Computing Machinery, New York, NY, 32–41. DOI: https://doi.org/10.1145/237721.237727
[53] R. Tarjan. 1971. Depth-first search and linear graph algorithms. In *Proceedings of the 12th Annual Symposium on Switching and Automata Theory (SWAT'71)*. 114–121. DOI: https://doi.org/10.1109/SWAT.1971.10
[54] Ermenegildo Tomasco, Truc L. Nguyen, Omar Inverso, Bernd Fischer, Salvatore La Torre, and Gennaro Parlato. 2016. Lazy sequentialization for TSO and PSO via shared memory abstractions. In *Proceedings of the Conference on Formal Methods in Computer-Aided Design (FMCAD'16)*. 193–200. DOI: https://doi.org/10.1109/FMCAD.2016.7886679
[55] Chao Wang, HoonSang Jin, Gary D. Hachtel, and Fabio Somenzi. 2004. Refining the SAT decision ordering for bounded model checking. In *Proceedings of the 41st Annual Design Automation Conference (DAC'04)*. Association for Computing Machinery, New York, NY, 535–538. DOI: https://doi.org/10.1145/996566.996713
[56] Chao Wang, Sudipta Kundu, Malay Ganai, and Aarti Gupta. 2009. Symbolic predictive analysis for concurrent programs. In *FM 2009: Formal Methods*, Ana Cavalcanti and Dennis R. Dams (Eds.). Springer Berlin, 256–272. DOI: https://doi.org/10.1007/978-3-642-05089-3_17
[57] Chao Wang, Zijiang Yang, Vineet Kahlon, and Aarti Gupta. 2008. Peephole partial order reduction. In *Tools and Algorithms for the Construction and Analysis of Systems*, C. R. Ramakrishnan and Jakob Rehof (Eds.). Springer Berlin, 382–396. DOI: https://doi.org/10.1007/978-3-540-78800-3_29
[58] CORPORATE SPARC International, Inc. 1994. The SPARC Architecture Manual (Version 9). Prentice-Hall, Inc. https://doi.org/10.5555/174556.
[59] Liangze Yin, Wei Dong, Wanwei Liu, Yunchou Li, and Ji Wang. 2018. YOGAR-CBMC: CBMC with scheduling constraint based abstraction refinement. In *Tools and Algorithms for the Construction and Analysis of Systems*, Dirk Beyer and Marieke Huisman (Eds.). Springer International Publishing, Cham, 422–426. DOI: https://doi.org/10.1007/978-3-319-89963-3_25
[60] Liangze Yin, Wei Dong, Wanwei Liu, and Ji Wang. 2018. Scheduling constraint based abstraction refinement for weak memory models. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering (ASE'18)*. Association for Computing Machinery, New York, NY, 645–655. DOI: https://doi.org/10.1145/3238147.3238223
[61] L. Yin, W. Dong, W. Liu, and J. Wang. 2020. On scheduling constraint abstraction for multi-threaded program verification. *IEEE Trans. Softw. Eng.* 46, 5 (May 2020), 549–565. DOI: https://doi.org/10.1109/TSE.2018.2864122
[62] Naling Zhang, Markus Kusano, and Chao Wang. 2015. Dynamic partial order reduction for relaxed memory models. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'15)*. Association for Computing Machinery, New York, NY, 250–259. DOI: https://doi.org/10.1145/2737924.2737956

Received 24 October 2021; revised 17 October 2022; accepted 2 December 2022