Consistency-Preserving Propagation for SMT Solving of Concurrent Program Verification

ZHIHANG SUN, Tsinghua University, China
HONGYU FAN, Tsinghua University, China
FEI HE*†, Tsinghua University, China

The happens-before orders have been widely adopted to model thread interleaving behaviors of concurrent programs. A dedicated ordering theory solver, usually composed of theory propagation, consistency checking, and conflict clause generation, plays a central role in concurrent program verification. We propose a novel preventive reasoning approach that automatically preserves the ordering consistency and makes consistency checking and conflict clause generation omissible. We implement our approach in a prototype tool and conduct experiments on credible benchmarks; results reveal a significant improvement over existing state-of-the-art concurrent program verifiers.

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

Additional Key Words and Phrases: Program verification, concurrent programs, memory model, satisfiability modulo theories

ACM Reference Format:
Zhihang Sun, Hongyu Fan, and Fei He. 2022. Consistency-Preserving Propagation for SMT Solving of Concurrent Program Verification. *Proc. ACM Program. Lang.* 6, OOPSLA2, Article 158 (October 2022), 28 pages. https://doi.org/10.1145/3563321

1 INTRODUCTION

Concurrent programs have been widely adopted to improve computing efficiency. They enable parallel computing by allowing multiple threads to run simultaneously. However, interleaving between threads causes numerous possible execution paths, placing a tremendous burden on verification. Therefore, it is highly desired to develop efficient techniques to alleviate the path explosion problem and improve the efficiency of concurrent program verification.

A common technique in concurrent program verification is to use partial orders (called *happens-before orders*) to represent the happens-before relation over shared memory access events [Alglave et al. 2013, 2012]. This technique classifies happens-before orders into several categories to express different semantics. For example, a *read-from order* indicates that a read event copies the value a write event wrote down; a *write-serialization order* indicates that a write event covers the value

*Corresponding author.
†Fei He is also with Key Laboratory for Information System Security, MoE of China, and Beijing National Research Center for Information Science and Technology, China.

Authors' addresses: Zhihang Sun, School of Software, Tsinghua University, Beijing, China, sunzh20@mails.tsinghua.edu.cn; Hongyu Fan, School of Software, Tsinghua University, Beijing, China, fhy18@mails.tsinghua.edu.cn; Fei He, School of Software, Tsinghua University, Beijing, China.

2475-1421/2022/10-ART158
https://doi.org/10.1145/3563321

another write event wrote down. Happens-before axioms are developed to specify the rules that all happens-before orders must obey.

Hardware memory models impose additional restrictions on happens-before orders. *Sequential consistency* (*SC*) [Lamport 1979] is the strictest and most commonly-assumed memory model, which requires a memory access to completely finish before the next access is issued. *Weak memory models* allow reordering of memory accesses to some extent; therefore, they allow more execution paths than SC. This paper focuses on SC and two widely adopted weak memory models: *total store order* (*TSO*) [Owens et al. 2009] and *partial store order* (*PSO*) [Weaver and Gremond 1994].

Lots of techniques have been proposed for solving constraints on these happens-before orders under the SAT/SMT framework. Earlier approaches (e.g., [Alglave et al. 2014, 2013, 2012; Sinha and Wang 2011]) suggest attaching an integer-valued timestamp to each event and representing the happens-before relation using the less-than relation over integer-valued timestamps. However, these approaches are somewhat over-reacting – they evaluate the exact value of each timestamp, while only their orders are necessary for solving. Moreover, these approaches need to apply happens-before axioms exhaustively, encoding all of their instances as constraints, irrespective of whether they actually take effect during verification, yielding a very large encoding formula.

He et al.'s Zord [He et al. 2021] overcomes these drawbacks by developing a dedicated theory solver for order constraints. Zord embodies the *from-read axiom* (Axiom 3 in Section 4.1) in its theory propagation module. Whenever the order constraint set is updated, Zord uses this axiom to derive necessary from-read orders. Using this approach, Zord avoids explicit and exhaustive encoding of *from-read* order constraints.

This paper proposes a new ordering theory solver (see Section 4). Compared to Zord [He et al. 2021], our solver also embodies the *write-serialization axiom* (Axiom 2 in Section 4.1). With this new solver, not only from-read orders, but also write-serialization orders can be automatically derived. Thanks to its reasoning ability, we no longer need to encode from-read and write-serialization constraints in the SMT formula. The size of the encoding formula is thus significantly reduced.

A typical theory solver needs to implement three procedures [Barrett and Tinelli 2018]: *theory propagation*, *consistency checking*, and *conflict clause generation*. This paper proposes a novel *preventive reasoning* approach (see Section 5): an assignment (unassigned yet) is called *fragile* if this assignment will directly lead to theory inconsistency; we find these fragile assignments and make the opposite assignments to prevent fragile assignments from occurring. An interesting finding is that by enforcing the opposite of these fragile assignments in propagation, ordering consistency is automatically achieved (Section 5.4). Therefore, our solver no longer needs to perform consistency checking and conflict clause generation; the workloads are thus significantly reduced.

We implement our approach in a prototype tool called Deagle[1] (`SV-COMP 2022 ConcurrencySafety` winner) and conduct experiments on 763 benchmarks collected from the `SV-COMP 2022 ConcurrencySafety` category. We compare our tool with state-of-the-art partial-order-based tools: CBMC [Alglave et al. 2013] and Zord [He et al. 2021], as well as the latest `SV-COMP ConcurrencySafety` winners: Lazy-CSeq [Inverso et al. 2014] and Yogar-CBMC [Yin et al. 2018b,a; Yin et al. 2020]. Experimental results show the effectiveness and efficiency of our approach: Deagle solves 34, 35, 42, and 12 more cases than CBMC, Zord, Lazy-CSeq, and Yogar-CBMC, respectively; on both-solved cases, Deagle runs 8.36x, 4.23x, 25.61x, and 1.47x faster than CBMC, Zord, Lazy-CSeq, and Yogar-CBMC, respectively. We also evaluate the effect of our consistency-preserving propagation (the propagation method that integrates preventive reasoning). Results show that consistency-preserving propagation achieves 1.30x speedup, with memory usage remaining almost unchanged. We also compare Deagle against Yogar-CBMC under TSO and PSO to evaluate our

[1]https://github.com/thufv/Deagle

performance under these weak memory models. Results show that Deagle solves 18 more cases than Yogar-CBMC under both TSO and PSO; on both-solved cases, Deagle achieves 1.85x and 1.82x speedups compared to Yogar-CBMC under TSO and PSO, respectively.

In summary, our main technical contributions include:

- A novel preventive reasoning approach that preserves the ordering consistency and makes the regularly required consistency checking and conflict clause generation omissible.
- A new ordering theory solver that has fewer encoding requirements and is more powerful in reasoning.
- An implementation of the approach and experimental results on credible `SV-COMP` benchmarks, demonstrating a significant improvement over state-of-the-art tools.

The rest of the paper is arranged as follows. Section 2 introduces basic notions and backgrounds; Section 3 demonstrates our symbolic encoding of concurrent programs; Section 4 proposes a basic ordering theory solver; Section 5 proposes consistency-preserving propagation. Implementation, evaluation, and experimental analysis are detailed in Section 6. Finally, we discuss related work in Section 7 and conclude the paper in Section 8.

2 PRELIMINARIES

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 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 a model M exists so that $M \models \Phi$.

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 a $\mathcal{T}$-model M exists so that $M \models \Phi$.

2.1 Satisfiability Modulo Theories

The *satisfiability modulo theories* (*SMT*) problem [Barrett and Tinelli 2018; de Moura and Bjørner 2008; De Moura and Bjørner 2011] determines the satisfiability of formulas in some combination of first-order background theories.

DPLL(T) is the standard algorithm for solving SMT problems. A typical DPLL(T) framework consists of a core solver (SAT solver) and several integrated theory solvers. Let ϕ be the input SMT formula, and $\mathcal{B}(\phi)$ its *Boolean abstraction* obtained by replacing each atom in ϕ with a fresh Boolean variable. Apparently, $\mathcal{B}(\phi)$ is a propositional logic formula and can be solved by the core solver. Moreover, $\mathcal{B}(\phi)$ is an over-approximation of ϕ with respect to satisfiability: if $\mathcal{B}(\phi)$ is unsatisfiable, so is ϕ; reversely, if $\mathcal{B}(\phi)$ is satisfiable, ϕ may not be. Therefore, after generating a satisfiable model M of $\mathcal{B}(\phi)$, we need to check whether M also satisfies ϕ by evaluating its consistency under the theory semantics.

Algorithm 1 shows how a DPLL(T) framework works. A model can be regarded as a set of assignments to all variables in $\mathcal{B}(\phi)$. Each assignment (assigning variable v to *true* or *false*) is represented as a literal (v or $\neg v$). DPLL(T) is essentially a DFS algorithm, which starts with an empty model and ends with a complete satisfiable model (or confirms that no satisfiable model exists). In each iteration, DPLL(T) decides a literal (Line 2) and performs unit propagation based on this decision (Line 4). Since some literals are *Boolean abstracted* from atoms belonging to some background theories, the assigned literals (either by decision or by propagation) need to be passed

Algorithm 1: DPLL(T) Algorithm

```
1  Procedure DPLL(T)
       Data: A partial model M
2      while l_d = decide() do
3          M ← l_d; theory_propagate(l_d);
4          for l_p in unit_propagation() do
5              M ← l_p; theory_propagate(l_p);
6              if inconsistent() then
7                  if conflict_resolution() then
8                      backtrack();
9                      break;
10                 else
11                     return unsat;
12                 end
13         end
14     end
15     return sat;
16 end
```

to their corresponding theory solvers for theory propagation and consistency checking (Line 3 and Line 5). If the current assignment is consistent with both SAT and theory semantics, another iteration starts to repeat the above process; if no more decisions can be made to expand the current assignment (Line 15), i.e., the current assignment is complete, DPLL(T) returns *sat*. In case of inconsistency, the corresponding theory solver generates one or several conflict clauses to prevent the core solver from reentering similar inconsistency in the future, and DPLL(T) backtracks by erasing several latest decisions (Line 7 - Line 8). In cases where no more decisions can be erased (Line 11), DPLL(T) terminates and returns *unsat*.

There are several schemes of interactions between the core solver and theory solvers. Algorithm 1 shows an *eager* scheme: each time the current model M is extended with a new assignment, no matter whether M is complete after extending, the core solver queries theory solvers for its consistency. There is also an alternative *lazy* scheme, which queries theory solvers only after a complete model M of $\mathcal{B}(\phi)$ is generated.

2.2 Modeling of Concurrent Programs

We follow Alglave et al.'s framework [Alglave et al. 2013] that models concurrent behaviors using *happens-before orders*. A *memory event* (*event* for short) e represents a memory access, specified by its address $addr(e)$, type $type(e)$ (either `read` or `write`), and enabling condition grd_e (an event is *enabled* if its enabling condition is *true*). Specifically, we write r (resp. w) for a `read` (resp. `write`) event. Denote $\mathbb{E}$ the set of all events in the program.

The *program order* $\prec_{po}$ is a total order on events from the same thread. Intuitively, $e_1 \prec_{po} e_2$ if they are issued by the same thread and e_1 occurs earlier than e_2. However, only a portion of program orders are encoded w.r.t. the chosen memory model. The subrelation $\prec_{po\text{-}loc}$ ($\subseteq \prec_{po}$) is the program order restricted to the same memory address. Under a certain memory model, some pairs of events in the program order are relaxed. We call the program order after relaxations the *preserved program order* $\prec_{ppo}$ ($\subseteq \prec_{po}$). This paper focuses on three memory models: *sequential consistency* (*SC*) [Lamport 1979], *total store order* (*TSO*) [Owens et al. 2009], and *partial store order* (*PSO*) [Weaver

(a) the original program (b) the CSSA form

Fig. 1. An example program

and Gremond 1994]. Among these models, SC relaxes nothing; TSO relaxes write-to-read program orders; PSO relaxes write-to-read and write-to-write program orders. Finally, only $\prec_{po\text{-}loc}$ and $\prec_{ppo}$ are encoded instead of the whole $\prec_{po}$.

The *read-from* order $\prec_{rf}$ links write event w to read event r, so that r obtains its value from w. A basic fact about the read-from order is that a read obtains its value from the *latest* write to the same address. Let r, w and w' be three events accessing the same address. If $w \prec_{rf} r$, w' must not happen between w and r (otherwise, w' is the *latest* write before r, and r should read from w' instead of w). Therefore, by $w \prec_{rf} r$ and w' preceding r, we can conclude that w' also precedes w (written $w' \prec_{ws} w$, called a *write-serialization* order); by $w \prec_{rf} r$ and w preceding w', we can conclude that r also precedes w' (written $r \prec_{fr} w'$, called a *from-read* order).

Let $\prec$ be the transitive closure of $\prec_{po\text{-}loc} \cup \prec_{ppo} \cup \prec_{rf} \cup \prec_{fr} \cup \prec_{ws}$, called the *happens-before* order. Intuitively, $e_1 \prec e_2$ indicates that the event e_1 must happen before e_2 in any execution of the program.

Lemma 1 (Consistent Execution [Alglave et al. 2012]). *An execution is* consistent *iff there is no event e so that $e \prec e$, i.e., there exists a linearization of events on this execution.*

An execution is *correct* if it satisfies the correctness property. A program is *correct* if it has no execution that is both consistent and incorrect.

3 SYMBOLIC ENCODING

In this section, we use a two-threaded program (in Figure 1a) to demonstrate our approach to encoding concurrent programs into SMT formulas. The encoding framework follows those in [Alglave et al. 2012; He et al. 2021], and the differences will be discussed at the end of this section.

3.1 Assignment Encoding

The concurrent static single assignment (CSSA) form [Wang et al. 2009] of the example program is depicted in Figure 1b, where each occurrence of each variable is replaced with a new copy of this variable (called an SSA variable). With the CSSA form, we can easily encode value assignments of the example program as:

$$\rho_{va} := (x_1 = 1) \wedge (y_1 = 2) \wedge (m = x_2 + y_2) \wedge (y_3 = 1) \wedge (grd_{x_3} \rightarrow x_3 = 2) \wedge (n = x_4 - y_5) \wedge (grd_{x_3} \leftrightarrow y_4 = 1) \quad (1)$$

Basically, each value assignment should be guarded by a conditional formula grd, representing its *enabling condition*. In this simple example, except for grd_{x_3}, enabling conditions of other assignments are equivalent to *true* and omitted.

The program's correctness property is that $m = 2$ and $n = 0$ should not hold simultaneously after both threads join. Its negation, i.e., the *error condition* of the program, is encoded as:

$$\rho_{err} := m = 2 \wedge n = 0 \quad (2)$$

3.2 Ordering Encoding

Each SSA variable represents a memory access to that variable. Given an SSA variable x_i, we use w_{x_i} (resp. r_{x_i}) to represent the corresponding event if x_i's type is `write` (resp. `read`).

Among the various happens-before orders of events (see Section 2.2), our approach only encodes the program order per location, the preserved program order (collectively called *program-related orders*) and the read-from order. Other orders (e.g., write-serialization and from-read) are automatically derived during the SMT solving in an on-demand fashion (see Section 4.1).

Considering the left thread of the example program, its program order per location is

$$\rho_{po\text{-}loc} \coloneqq (w_{x_1} \prec r_{x_2}) \wedge (w_{y_1} \prec r_{y_2})$$

its preserved program orders under SC, TSO and PSO are

$$\begin{aligned}
\rho_{ppo\text{-}sc} \coloneqq\ & (w_{x_1} \prec w_{y_1}) \wedge (w_{x_1} \prec r_{x_2}) \wedge (w_{x_1} \prec r_{y_2}) \wedge \\
& (w_{y_1} \prec r_{x_2}) \wedge (w_{y_1} \prec r_{y_2}) \wedge (r_{x_2} \prec r_{y_2}) \\
\rho_{ppo\text{-}tso} \coloneqq\ & (w_{x_1} \prec w_{y_1}) \wedge (r_{x_2} \prec r_{y_2}) \\
\rho_{ppo\text{-}pso} \coloneqq\ & (r_{x_2} \prec r_{y_2})
\end{aligned}$$

We abuse the ordering symbol $\prec$, making it also represent a predicate. A predicate $e_1 \prec e_2$ is called an *order constraint*. In this way, $\rho_{po\text{-}loc}$ and ρ_{ppo} defined above are conjunctions of order constraints.

We introduce a Boolean variable $rf^x_{i,j}$, called a *read-from variable* , to represent whether x_j reads from x_i. Therefore, if $rf^x_{i,j}$ is *true*, events w_{x_i} and r_{x_j} must both be enabled, and w_{x_i} happens before r_{x_j}. We have

$$rf^x_{i,j} \rightarrow grd_{x_i} \wedge grd_{x_j} \wedge x_i = x_j \tag{3}$$

$$rf^x_{i,j} \leftrightarrow w_{x_i} \prec_{rf} r_{x_j} \tag{4}$$

The former is called an *RF-Val* constraint and the latter is called an *RF-Ord* constraint [He et al. 2021]. Moreover, assuming $w_{x_{i_1}}, w_{x_{i_2}}, \cdots, w_{x_{i_k}}$ are possible writes that r_{x_j} may read from, then r_{x_j} must read from one of them, i.e.,

$$grd_{x_j} \rightarrow rf^x_{i_1,j} \vee rf^x_{i_2,j} \vee \cdots \vee rf^x_{i_k,j} \tag{5}$$

called an *RF-Some* constraint.

Let $\rho_{rf\text{-}val}$, $\rho_{rf\text{-}ord}$, and $\rho_{rf\text{-}some}$ be the conjunctions of all *RF-Val*, *RF-Ord*, and *RF-Some* constraints of the program, respectively.

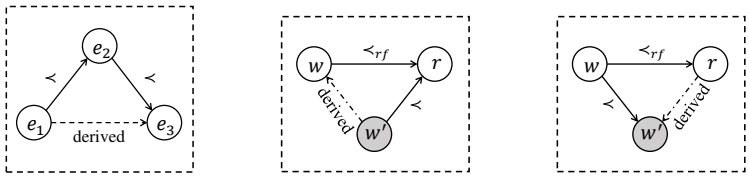
Taking r_{x_2} in Figure 1b as an example, which may read from w_{x_1} or w_{x_3}, we have

$$\begin{aligned}
& rf^x_{1,2} \rightarrow x_1 = x_2 \\
& rf^x_{1,2} \leftrightarrow w_{x_1} \prec_{rf} r_{x_2} \\
& rf^x_{3,2} \rightarrow grd_{x_3} \wedge x_3 = x_2 \\
& rf^x_{3,2} \leftrightarrow w_{x_3} \prec_{rf} r_{x_2} \\
& rf^x_{1,2} \vee rf^x_{3,2}
\end{aligned}$$

3.3 The Whole Formula

The whole encoding formula is

$$\Psi \coloneqq \psi_{ssa} \wedge \psi_{ord} \tag{6}$$



(a) Transitivity (b) Write-serialization (c) From-read

Fig. 2. Theory axioms

where ψ_{ssa} and ψ_{ord} represent the SSA encoding and ordering encoding, respectively:

$$\psi_{ssa} \coloneqq \rho_{va} \wedge \rho_{err} \wedge \rho_{rf\text{-}val} \wedge \rho_{rf\text{-}some} \tag{7}$$

$$\psi_{ord} \coloneqq \rho_{po\text{-}loc} \wedge \rho_{ppo} \wedge \rho_{rf\text{-}ord} \tag{8}$$

ψ_{ssa} is unrelated to the ordering, so that can be handled by classical SMT solvers; ψ_{ord} consists of Boolean variables (read-from variables and guard variables) and order constraints ($\prec$ and $\prec_{rf}$), called *ordering formulas*. We intend to devise a specific theory solver for handling ordering formulas.

3.4 Comparison with Existing Encoding Techniques

Alglave et al.'s modeling framework [Alglave et al. 2012] lays a foundation for modeling concurrent programs via partial orders. In Alglave et al.'s framework, all happens-before orders, including program orders, read-from orders, write-serialization orders, and from-read orders, need to be explicitly encoded into formulas. Zord [He et al. 2021] improves this framework by omitting from-read orders during encoding. In our approach, not only from-read orders but also write-serialization orders are omitted – they are left for the theory solver to deduce automatically. Therefore, our approach produces a significantly smaller formula than the existing approaches.

4 THE BASIC SOLVER

This section discusses our dedicated theory solver for order constraints.

4.1 Theory Axioms

Recall that our symbolic encoding formula contains only program and read-from orders. Some axioms are thus necessary for deducing all other orders in $\prec$. First, by transitivity of $\prec$, we have

Axiom 1 (Transitivity Derivation). *For any events e_1, e_2, and e_3,*

$$(e_1 \prec e_2) \wedge (e_2 \prec e_3) \rightarrow (e_1 \prec e_3).$$

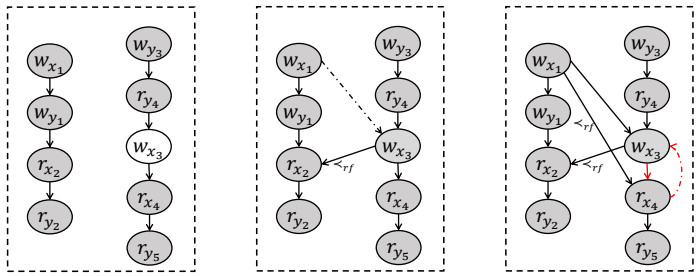
By the definition of the read-from orders, if $w \prec_{rf} r$, for any other *enabled* write w' to the same address, either w' precedes w, or w' follows r. Recall that in Section 2.2, a Boolean variable $grd_{w'}$ (called *guard variable*) is used to represent the enablement of w'. Then we have the following two axioms:

Axiom 2 (Write-serialization Derivation). *For any write events w, w' and read event r with $addr(w) = addr(w') = addr(r)$,*

$$(w \prec_{rf} r) \wedge (w' \prec r) \wedge grd_{w'} \rightarrow (w' \prec w).$$

Axiom 3 (From-read Derivation). *For any write events w, w' and read event r with $addr(w) = addr(w') = addr(r)$,*

$$(w \prec_{rf} r) \wedge (w \prec w') \wedge grd_{w'} \rightarrow (r \prec w').$$



(a) The initial state (b) After decision $rf^x_{3,2}$ (c) After decision $rf^x_{1,4}$

Fig. 3. Event graphs of the example

The enablement of w' is a necessary condition for Axiom 2 and Axiom 3 because otherwise, the event w' does not happen and discussing the happens-before ordering of w' is meaningless. On the other hand, $w \prec_{rf} r$ implies w and r being both enabled (see Equation (3)). Therefore, there is no need to consider enablements of w and r in the above axioms.

Figure 2 demonstrates the derivations in the above three axioms, where *dark* vertices represent enabled write events and *dashed* lines represent derived orders.

4.2 Event Graph

Our solver maintains a graph structure $(\mathbb{E}, \prec)$, called *event graph*, where $\mathbb{E}$ is the event set and $\prec$ is the happens-before order set. Moreover, each event e in $\mathbb{E}$ is associated with a guard variable grd_e, indicating the *enabling condition* of e. A guard variable grd_e is called *trivial* if the enabling condition it represents is *true*. Considering the example program in Figure 1b, the only non-trivial guard variable is grd_{x_3}, equivalent to $y_4 = 1$.

Let V_{ord} be the set of read-from and non-trivial guard variables, and α be the set containing the current assignments of variables in V_{ord}. At the beginning of SMT solving, all variables in V_{ord} are *unassigned*, so that only the transitive closure of program-related orders presents in the graph. Later, along with variable assignments, more and more orders are derived by applying theory axioms and added to the graph.

Definition 1. *An event graph is* stable *with respect to assignment set α if no more orders can be derived by applying theory axioms on α.*

Figure 3 shows three stable event graphs of the example program under SC on different stages of the SMT solving (orders derived by transitivity are omitted for clarity), where *dark* vertices represent enabled events and *light* vertices represent events whose enablements are undetermined yet.

4.3 Solving Procedure

Our basic solver consists of three components: *theory propagation*, *consistency checking*, and *conflict clause generation*. An overview of the solving procedure is illustrated in Figure 4.

4.3.1 Theory Propagation. On any assignment of literal l, the theory solver attempts to derive as many as possible orders by iteratively applying theory axioms. This procedure is called *theory propagation*.

Algorithm 2 shows the theory propagation algorithm. Note that assigned literal l is considered only if it assigns *true* to a variable in V_{ord}. Let $var(l)$ be the variable of l. No matter $var(l)$ is a

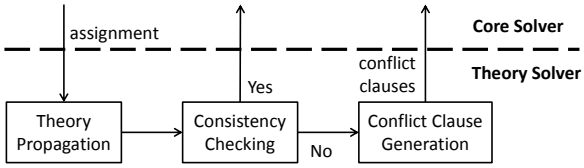


Fig. 4. Theory solver

read-from variable or a guard variable, the algorithm checks all possibilities for applying the write-serialization and from-read derivations (Line 6 to Line 8, and Line 12 to Line 14); for a read-from variable assignment, the algorithm also adds the corresponding order in $\prec$ and $\prec_{rf}$ (Line 4). During propagation, as soon as an order is added to $\prec$ (Lines 4, 19 and 22), *transitivity*() is invoked to derive orders by the transitivity axiom.

Theorem 1. *When Algorithm 2 terminates, the event graph* $(\mathbb{E}, \prec)$ *is stable with respect to the current assignment set* α.

Proof. Let α_k and $\prec_k$ be respectively the assignment set and the happens-before order set after the k-th assignment to V_{ord} variables. We prove the stability of $\prec_k$ with respect to α_k by induction on k:

(1) Initially, $\alpha_0 = \emptyset$, no WS or FR derivations can be performed. $\prec_0$ is the transitive closure of program-related orders, which is obviously stable with respect to α_0.
(2) Assume $\prec_i$ is stable with respect to α_i. Let l be the $(i+1)$-th assignment. According to Algorithm 2, all possible deductions triggered by l are fully considered. The resulting order set $\prec_{i+1}$ is thus stable with respect to $\alpha_i \cup \{l\}$, i.e., α_{i+1}.

□

4.3.2 Consistency Checking. After theory propagation, all happens-before orders under the current partial assignment present in the graph. The theory solver proceeds to check whether these happens-before orders are consistent. According to Lemma 1, we only need to enumerate all events in the event graph and check if any event contains a self-loop. Apparently, this check can be finished in $O(n)$ time, where n is the number of events.

4.3.3 Conflict Clause Generation. In case of inconsistency, the theory solver needs to find the reasons for the inconsistency. To this end, the theory solver records a *derivation reason* (*reason* for short) for each order in the graph. More specifically, each time we add an order to the graph, we also summarize its derivation reason as follows:

- The *reason* for a program-related order is *true*, for this order always presents in the graph;
- The *reason* for a read-from order is the corresponding read-from variable;
- if an order is derived by Axiom 1, its *reason* (following the notations in Axiom 1) is

$$reason(e_1 \prec e_2) \wedge reason(e_2 \prec e_3);$$

- if an order is derived by Axiom 2, its *reason* (following the notations in Axiom 2) is

$$reason(w \prec_{rf} r) \wedge reason(w' \prec r) \wedge grd_{w'};$$

- if an order is derived by Axiom 3, its *reason* (following the notation in Axiom 3) is

$$reason(w \prec_{rf} r) \wedge reason(w \prec w') \wedge grd_{w'}.$$

Algorithm 2: Theory Propagation

Data: the order sets $\prec$, $\prec_{rf}$

Procedure *theory_propagation*(l)
  Input: a positive literal l
  if $var(l)$ *is a read-from variable* **then**
    let w and r be the write and read events linked by $var(l)$;
    $\prec_{rf} \leftarrow \prec_{rf} \cup \{(w, r)\}$, $\prec \leftarrow \prec \cup \{(w, r)\}$;
    transitivity(w, r);
    foreach w' *s.t.* $addr(w) = addr(w') = addr(r)$ *and* $grd_{w'}$
      derive(w, w', r);
    end
  end
  if $var(l)$ *is a guard variable* **then**
    let w' be the write event guarded by $var(l)$;
    foreach w *and* r, *s.t.* $addr(w) = addr(w') = addr(r)$ *and* $w \prec_{rf} r$
      derive(w, w', r);
    end
  end
end
Procedure *derive*(w, w', r)
  Input: events w, w' and r, s.t. $addr(w) = addr(w') = addr(r)$, $(w, r) \in \prec_{rf}$ and $grd_{w'}$
  if $w' \prec r$ **then** /* WS derivation */
    $\prec \leftarrow \prec \cup \{(w', w)\}$;
    transitivity(w', w);
  else if $w \prec w'$ **then** /* FR derivation */
    $\prec \leftarrow \prec \cup \{(r, w')\}$;
    transitivity(r, w');
  end
end
Procedure *transitivity*(e, e')
  Input: a pair of events (e, e') s.t. $(e, e') \in \prec$
  foreach $(e', e'') \in \prec$ *s.t.* $(e, e'') \notin \prec$
    $\prec \leftarrow \prec \cup \{(e, e'')\}$;
    transitivity(e, e'');
  end
  foreach $(e'', e) \in \prec$ *s.t.* $(e'', e) \notin \prec$
    $\prec \leftarrow \prec \cup \{(e'', e)\}$;
    transitivity(e'', e);
  end
end

Let *cyc* be the set of self-loops detected in consistency checking. For each self-loop in *cyc*, we return the negation of its derivation reason as a conflict clause. The core solver *learns* the reported conflict clauses to prevent inconsistency with the same reasons from occurring in the future.

Finally, we prove that our proposed theory solver is correct w.r.t. our encoding:

Theorem 2. *When the theory solver returns a satisfiable model, this model represents a valid execution of the program under the given memory model; when the theory solver returns* unsatisfiable*, there is no valid execution of the program under the given memory model.*

Proof. Since Zord [He et al. 2021] employs a similar method which has been proved correct, we only need to prove that for each satisfiable model in our theory solver, a corresponding satisfiable model also exists in Zord, and vice versa.

Note that Zord requires enabled write events to the same address to be totally ordered: for both-enabled *write-write* pair (w_{x_i}, w_{x_j}), either $w_{x_i} \prec w_{x_j}$ or $w_{x_j} \prec w_{x_i}$. Compared to Zord, we avoid such complicated requirement. Considering a satisfiable model in our solver, if there exists an unordered pair (w_{x_i}, w_{x_j}), we can add an arbitrary order between them (note that if adding $w_{x_i} \prec w_{x_j}$ leads to inconsistency, $w_{x_j} \prec w_{x_i}$ must have been derived earlier). Therefore, for each satisfiabled model in our solver, we can find the corresponding satisfiable models in Zord.

On the other hand, a satisfiable model in Zord can be directly transformed into a satisfiable model of our solver by removing unnecessary WS orders (retaining only orders that can be derived from the WS axiom).

Above proof shows the equi-satisfiability between our solver and Zord, which guarantees our correctness based on Zord's correctness.

□

4.4 Example

We take the program in Figure 1 under SC as an example to demonstrate how the basic solver works. Figure 3a shows the program's initial event graph, where only program-related orders present. For readability, orders derived by transitivity (e.g. $w_{x_1} \prec r_{x_2}$, $w_{x_1} \prec r_{y_2}$, etc) are not drawn in the figure.

- Assume $rf^x_{3,2}$ is decided to *true* first: the theory solver adds the corresponding read-from order $w_{x_3} \prec_{rf} r_{x_2}$, and performs theory propagation. In addition to orders derived by transitivity (not drawn in the figure), one WS order $w_{x_1} \prec w_{x_3}$ (the dashed line in Figure 3b) is also derived from
$$(w_{x_3} \prec_{rf} r_{x_2}) \wedge (w_{x_1} \prec r_{x_2}) \wedge grd_{x_1} \rightarrow (w_{x_1} \prec w_{x_3}).$$
Now, the event graph becomes stable and is shown in Figure 3b.
- Assume $rf^x_{1,4}$ is decided to *true* second: the theory solver first adds $w_{x_1} \prec_{rf} r_{x_4}$ and performs theory propagation. Except for orders derived by transitivity, one FR order $r_{x_4} \prec w_{x_3}$ (the red dashed line in Figure 3c) is derived from
$$(w_{x_1} \prec_{rf} r_{x_4}) \wedge (w_{x_1} \prec w_{x_3}) \wedge grd_{x_3} \rightarrow (r_{x_4} \prec w_{x_3}).$$

Finally, from the event graph in Figure 3c, a self-loop is derived using the transitivity derivation

$$(w_{x_3} \prec r_{x_4}) \wedge (r_{x_4} \prec w_{x_3}) \rightarrow (w_{x_3} \prec w_{x_3}),$$

and its derivation reason is:

$$\begin{aligned} reason(w_{x_3} \prec w_{x_3}) &= reason(w_{x_3} \prec r_{x_4}) \wedge reason(r_{x_4} \prec w_{x_3}) \\ &= true \wedge (reason(w_{x_1} \prec_{rf} r_{x_4}) \wedge reason(w_{x_1} \prec w_{x_3}) \wedge grd_{x_3}) \\ &= true \wedge (rf^x_{1,4} \wedge (reason(w_{x_3} \prec_{rf} r_{x_2}) \wedge reason(w_{x_1} \prec r_{x_2}) \wedge grd_{x_1}) \wedge grd_{x_3}) \\ &= rf^x_{1,4} \wedge rf^x_{3,2} \wedge grd_{x_1} \wedge grd_{x_3}. \end{aligned}$$

The corresponding conflict clause is thus

$$\neg rf^x_{1,4} \vee \neg rf^x_{3,2} \vee \neg grd_{x_1} \vee \neg grd_{x_3}.$$

The above example shows that the basic theory solver *passively* preserves consistency (backtracks whenever inconsistency occurs) through consistency checking and conflict clause generation. In most cases, many time-consuming decisions and propagations are erased during backtracking. To avoid this drawback, we aim to enhance the theory solver with the ability to *actively* preserve consistency during theory propagation, instead of using consistency checking and conflict clause generation.

4.5 Comparison with Other Approaches

The first ordering theory for concurrent program verification was proposed in Zord [He et al. 2021]. The theory in Zord embodies only the from-read axiom. In comparison, our theory additionally embodies the write-serialization axiom. In this way, we omit explicit encoding of write-serialization constraints. Meanwhile, we also embody the transitivity axiom to construct the transitive closure of orders. As a result, we obtain a *complete* order set, with which consistency checking is reduced to trivial self-loop detection. In comparison, Zord relies on a cycle detection algorithm to perform consistency checking.

5 CONSISTENCY-PRESERVING PROPAGATION

This section enhances the above basic solver using *consistency-preserving propagation*. With this propagation, the ordering consistency is always preserved, so that the consistency checking and conflict clause generation can be safely omitted.

5.1 Overview

The main idea of consistency-preserving propagation is employing preventive reasoning to identify and disable the following *fragile* assignments. Recall that a positive literal v (resp. negative literal $\neg v$) denotes assigning variable v to *true* (resp. *false*).

Definition 2. *An assignment l is said* fragile *if its corresponding variable $var(l)$ is currently unassigned, and extending the current assignment set α with l leads to $\mathcal{T}$*-inconsistency, *i.e.,*

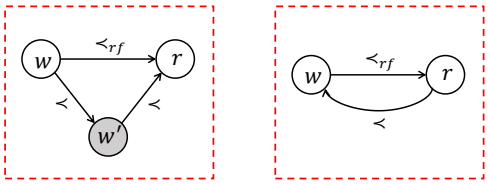
$$\alpha \cup \{l\} \models_{\mathcal{T}} \bot$$

In other words, we must prevent the fragile assignments from coming into truth. Therefore, once the theory solver detects a fragile assignment l, it makes the opposite assignment $\neg l$ and add it into α. In this way, $var(l)$ becomes assigned, and the fragile assignment l can never happen. This is called *preventive reasoning*.

For example, the literal $rf^x_{1,4}$ is fragile for the event graph in Figure 3b (where $\alpha = \{rf^x_{3,2}\}$). If preventive reasoning is enabled, we can immediately enforce the value of $rf^x_{1,4}$ to *false*. Then the decision in Figure 3c will not happen (since $rf^x_{1,4}$ is already assigned), and so does the subsequent inconsistency. This example shows the main insight of preventive reasoning: it prunes the decision space to avoid decisions that lead to inconsistency. More formally, we prove that if we perform preventive reasoning on each V_{ord} variable assignment, no self-loops will occur in the event graph during solving. In other words, consistency checking and conflict clause generation can be safely skipped, which simplifies the structure of the theory solver and increases solving efficiency by reducing the number of conflicts.

5.2 Preventive Reasoning

Let α be the current assignment set and $\prec_\alpha$ the order set stable with respect to α. Let l be a new assignment. If l conflicts with α, a self-loop will be formed during the theory propagation after several applications of theory axioms. Let us focus on how the self-loop is formed. First of all, we



(a) Pattern I (b) Pattern II

Fig. 5. Patterns of inconsistency

know that: 1) the last applied axiom that directly forms self-loop must be a transitivity derivation, because other two axioms, i.e., WS and FR, can only derive orders between two different events; 2) if two orders form a self-loop by transitivity, they must be reversed orders. Then we have the following lemmas.

Lemma 2. *Let α be the current assignment set, $\prec_\alpha$ the order set stable with respect to α, and l a new assignment. If l conflicts with α, then a self-loop is formed with two derivations.*

Proof. For simplicity, we denote in the following o an order and $\bar{o}$ its reversed order.

After each assignment, we *repeatedly* conduct derivations: upon deriving an order, we continue applying axioms on this order to derive more orders. For example, from l we derive o_1, from o_1 we derive $o_2, \cdots$, from o_{n-1} we derive o_n, which can be recorded as the *derivation chain* (*chain* for short) of o_n. Every derived order can be associated with such a chain. If $\bar{o}_n$ already exists, we say the chain of o_n leads to a self-loop. By contradiction we prove that if any chain $l, o_1, \cdots, o_n$ leads to a self-loop, n must be 1:

Let $l, o_1, \cdots, o_n$ be the shortest chain (assuming $n > 1$) that leads to a self-loop (so that $\bar{o}_n$ already exists in $\prec_\alpha$). We focus on the derivation from o_{n-1} to o_n. We can prove that whatever axiom is applied by this derivation (transitivity, WS, or FR), given that the consequence's reverse ($\bar{o}_n$) exists, we can derive the antecedent's reverse ($\bar{o}_{n-1}$). Taking WS derivation as an example (where $o_{n-1} = w' \prec r$ and $o_n = w' \prec w$):

$$o_{n-1} \wedge w \prec_{rf} r \wedge grd_{w'} \longrightarrow o_n,$$

We find that FR derivation can be employed on $\bar{o}_n$ to derive $\bar{o}_{n-1}$:

$$\bar{o}_n \wedge w \prec_{rf} r \wedge grd_{w'} \longrightarrow \bar{o}_{n-1}.$$

Since $\prec_\alpha$ is stable, $\bar{o}_{n-1}$ should also exist in $\prec_\alpha$.

As a result, we find a shorter chain $l, o_1, \cdots, o_{n-1}$ that leads to the same self-loop (the reader can refer to Section 4.3.3 to find that both self-loops share the same *derivation reason*, so we can treat them as indistinguishable), which is a contradiction.

In conclusion, as long as l conflicts with α, any self-loop caused by l is obtained in the following manner: l derives some order o_1 while $\bar{o}_1$ exists, so that causes this self-loop by transitivity. □

Lemma 3. *All self-loops formed in the theory propagation can be classified into the following two patterns:*

- $(w \prec w') \wedge (w' \prec r) \wedge (w \prec_{rf} r) \wedge grd_{w'} \Rightarrow \bot$;
- $(r \prec w) \wedge (w \prec_{rf} r) \Rightarrow \bot$.

Proof. According to Lemma 2, each self-loop is formed with two derivations: the first derives an order, and the second (must be transitivity) derives a self-loop from this order and its reversed order.

Now we discuss what the first derivation is. If l is the assignment to an RF variable (corresponding to $w \prec_{rf} r$), there are three cases:

- By the definition of RF: $w \prec_{rf} r \rightarrow w \prec r$;
- WS derivation: $w \prec_{rf} r \wedge grd_{w'} \wedge w' \prec r \rightarrow w' \prec w$ for some w';
- FR derivation: $w \prec_{rf} r \wedge grd_{w'} \wedge w \prec w' \rightarrow r \prec w'$ for some w'.

(Note that in the latter two cases, $w \prec_{rf} r \rightarrow w \prec r$ is unrelated to the self-loop's derivation.) The former case can be classified into the second pattern and the latter two cases into the first pattern.

On the other hand, if l is the assignment to a guard variable, only the WS and FR derivations are possible, both classified into the first pattern.

□

As depicted in Figure 5, the first inconsistency pattern describes a situation where r reads from w while another write event w' happens between them. From the first pattern, after applying the WS or FR derivation here comes to the inconsistency; from the second pattern, the inconsistency is triggered after applying the transitivity derivation.

Negations of these patterns are called *preventive clauses*, which should always be satisfied to preserve consistency:

- For write events w, w' and read event r with $addr(w) = addr(w') = addr(r)$,
$$\neg(w \prec w') \vee \neg(w' \prec r) \vee \neg(w \prec_{rf} r) \vee \neg grd_{w'};$$
- For write event w and read event r with $addr(w) = addr(r)$,
$$\neg(r \prec w) \vee \neg(w \prec_{rf} r).$$

Preventive reasoning performs propagations on those preventive clauses, similar to the idea of unit propagation: for each preventive clause, if only one of its literals is unassigned while all others are assigned *false*, to satisfy this clause, the final unassigned literal must be assigned *true*. For example, considering the event graph in Figure 3b, the preventive clause

$$\neg(w_{x_1} \prec w_{x_3}) \vee \neg(w_{x_3} \prec r_{x_4}) \vee \neg(w_{x_1} \prec_{rf} r_{x_4}) \vee \neg grd_{x_3}$$

is unit with $\neg(w_{x_1} \prec_{rf} r_{x_4})$ being the only unassigned literal. Then, by preventive reasoning, $\neg(w_{x_1} \prec_{rf} r_{x_4})$, i.e., $\neg rf^{x}_{1,4}$, is enforced.

5.3 Algorithm

Algorithm 3 details the consistency-preserving propagation algorithm with preventive reasoning integrated. On assigning positive literal l where $var(l) \in V_{ord}$, we still first employ theory axioms (Line 2, see Algorithm 2) to derive a stable order set. Let Δ be the newly derived order set. Preventive reasoning only considers orders between events with the same address, so we filter them into Δ_{per_addr} (Line 3). Then we identify preventive clauses related to these orders (Line 4 - Line 17) and make propagations on preventive clauses (Line 19 - Line 29).

A data structure Σ records all preventive clauses discovered and not satisfied so far. Recall that preventive reasoning is only responsible for unit propagation of read-from and guard variables. Therefore, a preventive clause should be unit-propagated by preventive reasoning only if all general $\prec$ orders already present:

- When write-write order $w \prec w'$ or write-read order $w' \prec r$ is added, the former two cases (Line 6 - Line 9 and Line 10 - Line 13) check if any happens-before chain $w \prec w' \prec r$ is constructed. For each constructed chain, we add $\neg(w \prec_{rf} r) \vee \neg grd_{w'}$ as a preventive clause.
- When read-write order $r \prec w$ is added, the last case (Line 14 - Line 15) adds $\neg(w \prec_{rf} r)$ as a preventive clause, which will be directly propagated.

Algorithm 3: Consisteny Preserving Propagation

```
   Data: the order set ≺, the preventive clause set Σ
 1 Procedure consistency_preserving_propagation(l)
       Input: a literal l
 2     Δ ← theory_propagation(l);
 3     Δ_per_addr ← filter_per_addr(Δ);
 4     foreach ord ∈ Δ_per_addr
 5         switch type of ord
 6             case write-read order w' ≺ r
 7                 foreach write w so that w ≺ w' ≺ r
 8                     Σ ← Σ ∪ {¬(w ≺_rf r) ∨ ¬grd_w'};
 9                 end
10             case write-write order w ≺ w'
11                 foreach write r so that w ≺ w' ≺ r
12                     Σ ← Σ ∪ {¬(w ≺_rf r) ∨ ¬grd_w'};
13                 end
14             case read-write order r ≺ w
15                 Σ ← Σ ∪ {¬(w ≺_rf r)};
16         preventive_propagation();
17     end
18 end
19 Procedure preventive_propagation()
20     foreach unit clause cl ∈ Σ
21         switch the unassigned literal in cl
22             case negated order ¬(w ≺_rf r)
23                 Let v_rf be RF variable of w ≺_rf r;
24                 v_rf ← false;
25             case negated guard variable ¬grd_w'
26                 grd_w' ← false;
27         Σ ← Σ/{cl};
28     end
29 end
```

Note that preventive clauses added to Σ are simplified: general happens-before orders $\prec$ that do not correspond to Boolean variables are omitted; all remaining literals are Boolean or correspond to Boolean variables. Therefore, we perform *preventive_propagation*() on clauses in Σ as common unit propagation: if the unassigned literal in a unit clause is a negated read-from order, we assign the corresponding read-from variable to *false* (Line 24); if the unassigned literal is a negated guard variable, we assign the guard variable to *false* (Line 26).

5.4 Properties

We prove that our proposed consistency-preserving propagation (Algorithm 3) surely preserves ordering consistency during solving. To prove this property, we need the following lemmas.

For preventive clause cl, we denote $\#$*unassigned-lits*(cl) the number of *unassigned* literals in cl. Let Σ_k be the set of preventive clauses after the k-th invocation of *consistency_preserving_propagation*().

Lemma 4. *For any preventive clause cl to be added to Σ, #unassigned-lits$(cl) \geq 1$.*

Lemma 5. *If a preventive clause cl is kept in Σ on exit of the algorithm, #unassigned-lits$(cl) \geq 2$.*

Lemma 6. *For any preventive clause $cl \in \Sigma_k$, after the $(k+1)$-th invocation of the algorithm, #unassigned-lits(cl) decreases 0 or 1.*

Now we prove the *consistency-preserving property* of Algorithm 3 in our theory solver:

Theorem 3 (Consistency-preserving property). *While incrementally assigning read-from and guard variables, if consistency-preserving propagation is conducted on each assignment as Algorithm 3 shows, the event graph is always consistent (self-loop free).*

Proof. According to Lemma 3, this theorem is reduced to that no preventive clauses can be unsatisfiable during solving. Let Σ_k be the preventive clause set after the k-th invocation of *consistency_preserving_propagation*(). We prove that no preventive clause in Σ_k can be unsatisfiable by induction on k:

(1) Initially, no read-from variables are assigned to *true*, so that no preventive clause can become unsatisfiable.
(2) Assuming the statement holds for $k = i$, now we prove the statement also holds for $k = i+1$. By Lemma 5, Σ_i contains no unit clauses. During the $(i+1)$-th invocation, Lemma 6 guarantees that existing clauses in Σ_i cannot be unsatisfiable, and Lemma 4 guarantees that new clauses added to Σ_{i+1} also cannot be unsatisfiable. Therefore, no preventive clause in Σ_{k+1} is unsatisfiable.

Therefore, after any number of invocations, no preventive clauses can be satisfiable, so the event graph contains no self-loops. □

The following theorem shows the soundness and completeness of the consistency-preserving propagation.

Theorem 4 (Correctness). *Let α be the current assignment set. If Algorithm 3 assigns literal $\neg l$, then l must be ordering-fragile; For any ordering-fragile assignment l, Algorithm 3 is able to find it and assign its negation $\neg l$.*

Proof. The former part is obvious, since derivations (Line 6 - Line 15) and unit propagations (Line 24, Line 26) of preventive clauses are based on theory axioms.

Consider the latter part. According to Lemma 3, the inconsistency that l will cause must correspond to some preventive clause. Before assignment of l, this preventive clause is unit with only $\neg l$ unassigned, so that Algorithm 3 must have propagated $\neg l$ in an earlier invocation. □

5.5 Example

After enhancing the theory solver with consistency-preserving propagation, the program shown in Figure 1 can be solved even without any decision.

Based on the Boolean theory, the core solver finds out that x_2 and y_2 must both be assigned to 1 to guarantee $m = 2$ (for reaching error condition $m = 2 \wedge n = 0$). Therefore, x_2 reads from x_1 ($rf^x_{1,2}$) and y_2 reads from y_3 ($rf^y_{3,2}$), respectively. Assuming $rf^y_{3,2}$ ($w_{y_3} \prec_{rf} r_{y_2}$) is assigned first, we obtain $w_{y_1} \prec w_{y_3}$ through WS derivation.

Order $w_{y_1} \prec w_{y_3}$ composes chain $w_{y_1} \prec w_{y_3} \prec r_{y_4}$ and $w_{y_1} \prec w_{y_3} \prec r_{y_5}$, so that $\neg(w_{y_1} \prec_{rf} r_{y_4}) \vee \neg grd_{y_3}$ and $\neg(w_{y_1} \prec_{rf} r_{y_5}) \vee \neg grd_{y_3}$ are added as preventive clauses. Given that grd_{y_3} is trivial, we derive $\neg(w_{y_1} \prec_{rf} r_{y_4})$ and $\neg(w_{y_1} \prec_{rf} r_{y_5})$. That is, neither r_{y_4} nor r_{y_5} reads from w_{y_1}, then their only choice is to read from w_{y_3}. Since $y_4 = y_3 = 1$, grd_{x_3} holds.

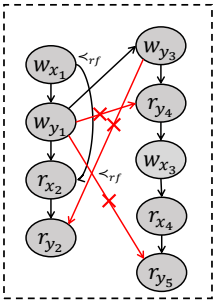


Fig. 6. The event graph after preventive reasoning

On the other hand, from $w_{y_1} \prec w_{y_3}$, we also propagate $w_{x_1} \prec w_{x_3}$ by transitivity, which composes chain $w_{x_1} \prec w_{x_3} \prec r_{x_4}$. Therefore, we add preventive clause $\neg(w_{x_1} \prec_{rf} r_{x_4}) \vee \neg grd_{x_3}$ and propagate $\neg(w_{x_1} \prec_{rf} r_{x_4})$. Finally, we confirm that x_4 reads from x_3. Updates on the event graph are shown in Figure 6, where red arrows with crosses represent *must-not-read-from orders* $\neg(w \prec_{rf} r)$ derived above.

The above propagation procedure has determined all values of *read* events, which yield $n = x_4 - y_5 = x_3 - y_3 = 1$, contradicting $n = 0$. Therefore, no counterexample satisfying $m = 2 \wedge n = 0$ exists; correctness property $\neg(m = 2 \wedge n = 0)$ holds.

All theory axioms (Axiom 1 - Axiom 3) can only derive $\prec$ orders, but cannot derive negated read-from orders as consistency-preserving propagation does. Therefore, without preventive reasoning, DPLL(T) must decide which events r_{x_4}, r_{y_4}, and r_{y_5} read from, and backtrack after inconsistency. Therefore, consistency-preserving propagation can derive more literals under the same partial assignment, which helps the SMT solver finish the solving procedure faster.

6 EVALUATION

This section reports the experimental results of our consistency-preserving propagation and comparison with state-of-the-art concurrent verification tools.

6.1 Implementation and Setup

We implement our approach in a tool called Deagle on top of CBMC and MiniSAT. CBMC is a powerful program verifier that uses partial orders to model concurrent programs, and MiniSAT is a well-designed and widely-adopted SAT solver. We modify CBMC by disabling the generation of WS and FR constraints, and use it as the front-end for generating encoding formulas. Meanwhile, MiniSAT is employed as the back-end for solving the generated formulas. We develop an ordering theory plug-in for MiniSAT, enabling MiniSAT to run DPLL(T) specified for our ordering theory solver.

We collect all 763 benchmarks from the `SV-COMP 2022`[2] `ConcurrencySafety` category. It contains 15 sub-categories which can be classified into:

- *Classical cases* (492): cases that exist up to `SV-COMP 2020` and the `Nidhugg benchmark suite`, including *pthread* (44), *pthread-atomic* (11), *pthread-ext* (48), *pthread-wmm* (283), *pthread-lit* (11), *ldv-races* (12), *ldv-linux-3.14-races* (7), *pthread-complex* (5), *pthread-driver-races* (21), *pthread-C-DAC* (4), *pthread-divine* (16), *pthread-nondet* (6), and *pthread-deagle* (24). Most concurrent verification tools work well on these classical tasks.

[2]https://sv-comp.sosy-lab.org/2022/

Table 1. Results on consistency-preserving propagation

Category	Strategy	Solved	CPU Time (s)	Memory (GB)	Decisions	Propagations
ALL	Deagle	562	2714	50.18	40.6M	30.9G
	Deagle$^-$	562	3532	52.01	96.6M	31.8G
Classical	Deagle	461	835	45.53	33.5M	1.83G
	Deagle$^-$	461	1033	47.10	87.3M	2.89G
Latest	Deagle	101	1879	4.65	7.07M	29.0G
	Deagle$^-$	101	2499	4.91	9.32M	28.9G

- *Latest cases* (271): cases added in `SV-COMP 2021` and `SV-COMP 2022`, including *goblint-regression* (103) and *weaver* (168). These tasks contain large loops or complex library functions, which are not suitable for most concurrent verification tools to solve.

Many cases in the benchmark set contain loops. All tools involved in our experiments employ loop unwinding to transform the original program into a loop-free program. For a program with loops whose iteration numbers can be predetermined by static analysis, its *unwinding limit* is set to this predetermined number; otherwise, this limit is set to a constant number. In either case, all tools use the same unwinding limit for the same benchmark. The constant number for the unwinding limit is set to 2 in our experiments. The reported result *false* indicates a *falsified* benchmark, while the result *true* only shows the *bounded correctness* of this benchmark. Among *true* cases, we report *proved* if the program is fully expanded (no execution path exceeds the unwinding limit), and report *borderline* otherwise. *Proved*, *falsified*, and *borderline* results are all considered *solved* in the following analysis.

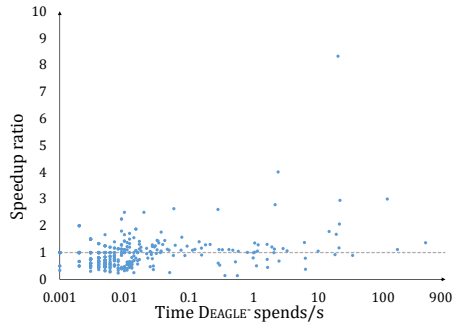
The first experiment evaluates the effectiveness of our consistency-preserving propagation. The second experiment compares Deagle to state-of-the-art tools that also employ partial order-based modeling and SAT/SMT-based solving, including:

- CBMC: a well-known tool that implemented the approach in [Alglave et al. 2013] by modeling concurrent programs using partial orders and solving the encoding formula by SAT/SMT solvers. CBMC is used as the base by Zord and Deagle.
- Zord: a recently proposed approach [He et al. 2021] that implemented a dedicated theory solver for order constraints of concurrent program verification.

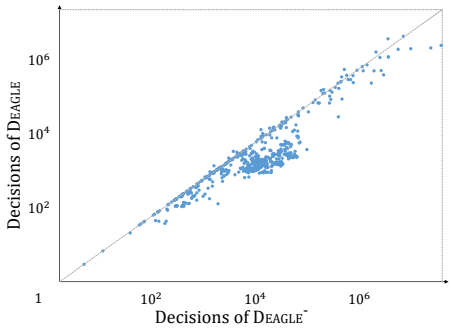
The third experiment compares Deagle to other state-of-the-art tools that achieved outstanding results in `SV-COMP ConcurrencySafety` category:

- Lazy-CSeq: a concurrent program verifier based on lazy sequentialization [Inverso et al. 2014], the `SV-COMP ConcurrencySafety` category winner in 2020 and 2021.
- Yogar-CBMC: a tool based on the abstraction-refinement method [Yin et al. 2018b], the `SV-COMP ConcurrencySafety` category winner in 2017, 2018, and 2019. Also, Yogar-CBMC has been extended to weak memory models TSO and PSO [Yin et al. 2017]. Therefore, we also make comparisons with Yogar-CBMC under TSO and PSO to evaluate our performance on these weak memory models.

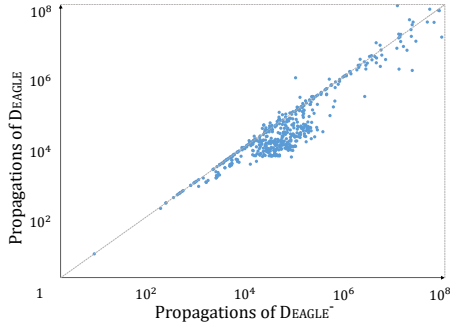
All experiments are conducted on a computer with an Intel(R) Core(TM) i5-10400 CPU and 16 GB memory. The operating system is Archlinux-5.11.8. Following the experimental settings of `SV-COMP`, each verification task's time limit is set to 900 s and memory limit is set to 15 GB.



(a) DEAGLE vs. DEAGLE⁻ on speedup ratio



(b) DEAGLE vs. DEAGLE⁻ on decision number



(c) DEAGLE vs. DEAGLE⁻ on propagation number

Fig. 7. Comparing DEAGLE with DEAGLE⁻ on classical cases

6.2 Evaluation of Consistency-Preserving Propagation

DEAGLE implements the enhanced solving procedure with the consistency-preserving propagation. Denote DEAGLE⁻ as the configuration that implements only the basic solving procedure (Section 4). We compare DEAGLE to DEAGLE⁻ to evaluate the effectiveness and efficiency of the consistency-preserving propagation.

Given that the front-ends (for generating SMT encoding formulas) of DEAGLE and DEAGLE⁻ are the same, this experiment compares the CPU time and peak memory of their back-ends (for solving the SMT formulas) *only*. Experimental results are summarized in Table 1, where results on classical cases and latest cases are also separately listed. Among all 763 benchmarks, both strategies solve the same 562 cases, including 461 classical cases (out of 492) and 101 latest cases (out of 271). Compared to DEAGLE⁻, DEAGLE achieves 1.30x speedup with the similar peak memory usage. Note that latest cases are mostly unsolvable, and even those solvable by coincidence are not representative w.r.t. their scales. Therefore, the following detailed evaluations between DEAGLE and DEAGLE⁻ are performed on classical cases.

We evaluate the performance of DEAGLE and DEAGLE⁻ on each case to confirm whether consistency-preserving propagation is efficient on the majority of the benchmarks. The speedup ratio DEAGLE achieves over DEAGLE⁻ on each case is summarized in Figure 7a, whose *X-axis* shows the runtime of DEAGLE⁻ and *Y-axis* shows the speedup ratio (runtime of DEAGLE⁻/ runtime of DEAGLE). Points above the horizontal line $y = 1$ represent cases on which consistency-preserving propagation shows efficiency. Since the runtimes of small cases are too unstable to exactly record, we only consider

Table 2. Overall results with state-of-the-art tools under SC

Tool	Solved (classical + latest)	Both-solved		
		Num.	CPU Time (s) (-/Deagle/Deagle$^-$)	Memory (GB) (-/Deagle/Deagle$^-$)
Deagle	562 (461 + 101)	-	-	-
Deagle$^-$	562 (461 + 101)	-	-	-
CBMC	528 (437 + 91)	525	6681/**799**/**987**	48.68/**35.35**/**36.58**
Zord	527 (435 + 92)	525	3568/**844**/**1063**	44.96/**36.18**/**37.70**
Lazy-CSeq	520 (424 + 96)	487	13701/**535**/ -	63.20/**21.84**/-
Yogar-CBMC	550 (450 + 100)	550	3723/**2532**/ -	45.88/**44.83**/-

the speedup ratio of larger cases. Deagle runs faster on 65.5% of all large (>100ms) cases, and the average speedup ratio (over all large cases) is 1.58x. Based on the above results, we conclude that consistency-preserving propagation speeds up the SMT solving procedure in most benchmarks.

An obvious advantage of consistency-preserving propagation is that more intermediate results are derived by applying preventive reasoning (Section 5.1); in contrast, obtaining these results in the basic solver usually costs many decisions and backtracks. As Section 5.5 shows, the example program can be solved without any decision using consistency-preserving propagation. We discuss if consistency-preserving propagation's effect on reducing decisions and shortening the solving procedure is universal. The rightmost columns *Decisions* and *Propagations* in Table 1 sum up the numbers of decisions and propagations made by each tool. Figure 7b and Figure 7c detail the number of decisions and propagations these tools make on each classical case. Totally, consistency-preserving propagation reduces decisions by 61.6% and reduces propagations by 36.7%. In summary, consistency-preserving propagation successes in reducing decisions and shortening the solving procedure as is expected.

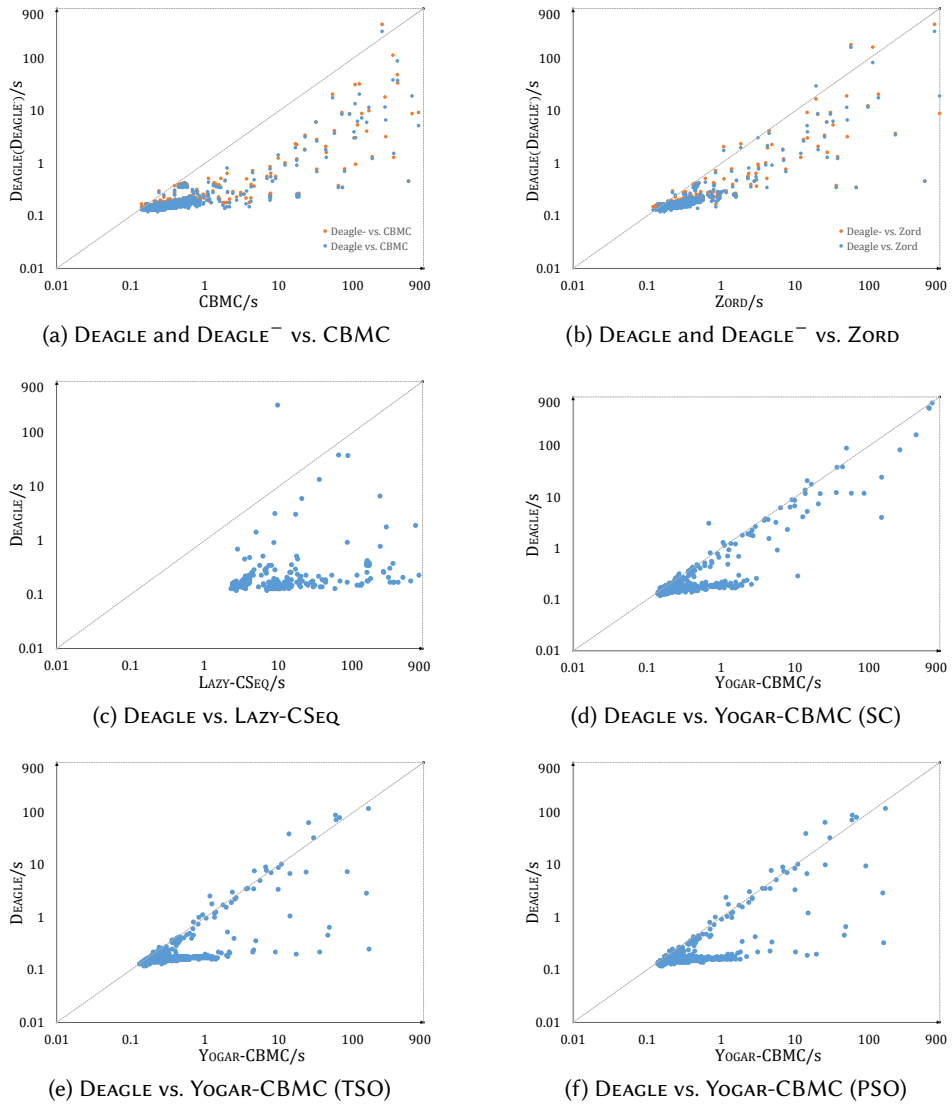
Note that the amount of speedup on classical cases (1.24x) does not precisely match the amounts of decision reduction (61.6%) and propagation reduction(36.7%). We want to discuss the overhead of our method. The maximal number of preventive clauses that Deagle needs to maintain is $O(n^3)$, where n is the number of events. But in actual, Deagle adds a preventive clause $\neg(w \prec_{rf} r) \vee \neg grd_{w'}$ only when a $w \prec w' \prec r$ chain is constructed. Therefore, the number and size of preventive clauses are reduced. The experimental results indicate that employing consistency-preserving propagation is worthwhile in most cases.

6.3 Comparison with Partial-Order-Based Tools

The second experiment compares Deagle and Deagle$^-$ with other partial-order-based concurrent program verifiers, namely CBMC and Zord. CBMC is an early and well-known bounded model checker which verifies concurrent programs using the partial-order-based approach introduced in [Alglave et al. 2012]. CBMC serves as a framework for partial-order-based approaches, based on which some later tools, including Zord and Deagle, are developed.

The main technical differences between these tools are listed below:

- Based on CBMC, Zord proposes axiomatic *FR derivation*, so that Zord avoids exhaustive FR encoding compared to CBMC.
- Compared to Zord, Deagle$^-$ proposes axiomatic *WS derivation* and implements the theory solver into MiniSAT instead of Z3.
- Compared to Deagle$^-$, Deagle proposes *consistency-preserving propagation*.



(a) DEAGLE and DEAGLE⁻ vs. CBMC

(b) DEAGLE and DEAGLE⁻ vs. ZORD

(c) DEAGLE vs. LAZY-CSEQ

(d) DEAGLE vs. YOGAR-CBMC (SC)

(e) DEAGLE vs. YOGAR-CBMC (TSO)

(f) DEAGLE vs. YOGAR-CBMC (PSO)

Fig. 8. Comparing DEAGLE (DEAGLE⁻) with individual tools

Results. Experimental results are summarized in Table 2. Column *Solved* shows the number of cases each tool successfully verifies within the time limit (we also respectively count classical and latest cases); column *Num* shows the number of both-solved cases in comparison with DEAGLE (the same as DEAGLE⁻); columns *Time* and *Memory* sum up the CPU time and memory usage of the selected tool/DEAGLE/DEAGLE⁻ on both-solved cases. Different from the first experiment (Section 6.2), the following experiments (Section 6.3 and Section 6.4) compare the end-to-end performance of tools (since these tools employ different front-ends), so the reported time and memory usages in Table 2 include those spent by both back-ends and front-ends.

As Table 2 shows, Deagle (also Deagle$^-$) solves 34 more cases than CBMC and 35 more cases than Zord. Considering both-solved cases, Deagle runs 8.36x faster and consumes 27.4% less memory than CBMC, and 4.23x faster and 19.5% less memory than Zord. Deagle$^-$ also runs 6.77x faster and consumes 24.9% less memory than CBMC, and 3.36x faster and consumes 16.1% less memory than Zord. In conclusion, both our proposed theory axioms and the consistency-preserving propagation enable us to solve more cases and significantly accelerate the solving efficiency compared to the previous partial-order-based approaches.

After each assignment of ordering variables, Deagle's consistency-preserving propagation applies axiomatic deduction to derive more orders and performs preventive reasoning to enforce values of other ordering variables. Those deductions and enforcements help the constraint solver reduce redundant search space and improve the time efficiency.

Figure 8a and Figure 8b compare Deagle (blue points) and Deagle$^-$ (orange dots) with CBMC and Zord, respectively, on the CPU time of each both-solved case. Note that the speedup ratios for some cases are so large (up to 100) that we cannot use the same diagram as Figure 7a. However, if we still calculate the speedup ratio of each case to follow the style of Figure 7a (X-axis as the baseline's time and Y-axis as the speedup ratio), some ratios are too large (up to 100x) to show in the figure. In these figures, the X and Y axes show the runtime of baseline and Deagle/Deagle$^-$, respectively. Points below (or above) the diagonal represent cases where Deagle/Deagle$^-$ is superior (or inferior) to the compared tool. There are clusters of points at the bottom left in both figures, which are trivial cases and can be quickly solved by both tools. Therefore, they distribute irregularly around the diagonal. As verification tasks go complex, most cases are below the diagonal, indicating that our tactic is superior to the baselines.

Deagle is inferior to the baselines in several complex cases (CPU time > 1s). Both baseline tools learn conflict clauses from ordering inconsistency and use the learned conflict clauses to prohibit the same inconsistency from occurring again. After each backtrack (or restart), a portion of the preventive clauses (corresponding to the backtracked decisions) are abandoned in Deagle, while the learned conflict clauses can all be kept in CBMC and Zord. These conflict clauses are still useful in SAT solving of the core solver. Therefore, on benchmarks where backtracks/restarts are frequent, Deagle is at a disadvantage.

Note that both Figure 8a and Figure 8b use a log scale for their x- and y-axes. The data points of Deagle and Deagle$^-$ look very similar in these two figures. However, as reported in Section 6.2, Deagle is 1.30x faster than Deagle$^-$. Even though, the speedup of Deagle over Deagle$^-$ is not as significant as that of Deagle$^-$ over Zord. There are two reasons for this. Firstly, Zord is built on Z3, while Deagle$^-$ (and also Deagle) are built on MiniSAT. This engine change contributes partially to the final performance improvement. Secondly, preventive reasoning suggests a new framework of theory solving. This new framework is very different from the classical ones in that the consistency checking and conflict clause generation can be safely omitted. Therefore, Deagle and Deagle$^-$ employ actually two different theory solving frameworks when implemented. Our implementation of Deagle is still experimental, and optimizations can be conducted to further improve its efficiency. For example, we may borrow optimizations in SAT solvers (including better-designed data structures for dynamically maintaining clauses, the two watched literals technique, etc.) to facilitate preventive reasoning in the future.

6.4 Comparison with Other Verifiers

The third experiment compares Deagle with Lazy-CSeq and Yogar-CBMC, the champion tools in the `SV-COMP ConcurrencySafety` category from 2017 to 2021. Experimental results are also summarized in Table 2; detailed comparisons on each case are depicted in Figure 8c, Figure 8d, Figure 8e, and Figure 8f.

Comparison with Lazy-CSeq. Table 2 summarizes that Deagle solves 42 more cases than Lazy-CSeq. Considering both-solved cases, Deagle runs 25.61x faster and consumes 65.4% less memory than Lazy-CSeq. Figure 8c demonstrates that Deagle outperforms Lazy-CSeq in all but 1 complex case.

Lazy-CSeq uses a different idea from the partial-order-based approaches. It transforms a concurrent program into a single-threaded one, which executes events from all threads in a *round-robin* scheme. The transformed program can be solved by any single-threaded verification tool (e.g., CBMC). However, the size of the transformed program is usually very large. Compared to Lazy-CSeq, our tool sees the essence of concurrent behaviors, i.e., their happens-before orderings, and yields better performance.

Compared to other tools that only set an *unwinding limit* as the parameter, Lazy-CSeq also sets a *round-robin limit* (the maximal number of thread switches). This parameter greatly affects Lazy-CSeq's performance: a larger round-robin limit yields a larger transformed program; however, a smaller round-robin limit risks missing counterexamples. In this experiment, we follow the default strategy of Lazy-CSeq in `SV-COMP`: Lazy-CSeq attempts three settings of the round-robin limit: 2, 4, and 20. For each case, Lazy-CSeq tries these settings from small to large, until a counterexample is generated. If all settings cannot find a counterexample, Lazy-CSeq proves the *bounded correctness* of the program within 20 thread switches.

The bounded switching strategy is a limitation of Lazy-CSeq, but it can give Lazy-CSeq a superior status on rare occasions. There are two cases on which Lazy-CSeq outperforms Deagle. Both of them are *trivial in the round-robin scheme*: their property can be violated within one or two thread switches. Therefore, Lazy-CSeq can find a counterexample with the smallest setting of the round-robin limit. Deagle is more *complete* than Lazy-CSeq, in that we need not limit the number of thread switches.

Comparison with Yogar-CBMC under SC. As Table 2 summarizes, Deagle solves 12 more cases than Yogar-CBMC. Considering both-solved cases, Deagle runs 1.47x faster and consumes 2.3% less memory than Yogar-CBMC. According to Figure 8d, Deagle outperforms Yogar-CBMC in all but 3 complex cases.

Though also derived from CBMC, Yogar-CBMC features a *CEGAR* (*counterexample-guided abstract refinement*) method, which applies a *lazy* scheme to invoke the ordering theory. It does not check ordering consistency until a complete model is generated. When a self-loop is detected, it learns conflict clauses and completely restarts the solving procedure. In this method, a self-loop, which could have been detected or avoided, is postponed until the model becomes complete. In contrast, our Deagle is constantly adjusting the decision space to avoid self-loops.

Comparison with Yogar-CBMC under Weak Memory Models. Since Yogar-CBMC has also been adapted to TSO and PSO, we evaluate Deagle against Yogar-CBMC under these memory models. Experimental results are summarized in Table 3 and detailed CPU time comparisons on each case are depicted in Figure 8e (TSO) and Figure 8f (PSO), respectively.

Table 3 summarizes that Deagle solves 18 more cases than Yogar-CBMC under both TSO and PSO. Considering both-solved cases, Deagle runs 1.85x and 1.82x faster under TSO and PSO, respectively, while memory usage remains almost the same. In conclusion, compared to Yogar-CBMC, under memory model TSO and PSO, we achieve an even better efficiency improvement than the improvement under SC (1.47x).

Note that both Deagle and Yogar-CBMC spend less time under TSO/PSO than SC. Especially, Deagle solves 6 more cases (568 under TSO/PSO; 562 under SC). This is because weak memory models allow more executions of programs, making some cases *sat* though originally *unsat* under

Table 3. Results compared with Yogar-CBMC under weak memory models

Model	Tool	Yogar-CBMC/Deagle			
		Solved (classical + latest)	Both-Solved	CPU Time (s)	Memory (GB)
TSO	Deagle	568 (462 + 106)	-	-	-
	Yogar-CBMC	550 (449 + 101)	550	1319/714	44.01/43.08
PSO	Deagle	568 (462 + 106)	-	-	-
	Yogar-CBMC	550 (449 + 101)	550	1310/720	44.01/43.09

SC. On these cases, verification tools (under SC) need to prove that no models are satisfiable by pruning all possibilities; but (under weak memory models) find a satisfiable model halfway.

6.5 Threats to Validity

The main threats to validity are whether our implementation and evaluation are credible.

Firstly, Deagle is built on top of CBMC and MiniSAT. At the front-end, our implementation is simpler than CBMC since both write-serialization and from-read orders are omitted; moreover, our implementation is straightforward and well inherited. At the back-end, we implement the consistency-preserving theory solver as a plug-in for MiniSAT. Our implementation is loosely-coupled with MiniSAT. Additionally, the above two aspects also show that the performance improvements are mainly due to our approach.

Secondly, all benchmarks are collected from the `ConcurrencySafety` category of `SV-COMP 2022`. These benchmarks are representative, comprehensive, credible, and have been widely accepted by the program verification community. Additionally, most of these benchmarks contain loops. Bounded model checking [Clarke et al. 2001] is a popular technique for handling programs with loops – by unwinding loops to a predefined limit, the program can be transformed into a loop-free version. The competitor tools in our experiments are all based on bounded model checking, but they possess different strategies for determining the unwinding limit. With different unwinding limits, the transformed loop-free programs are different. For a fair comparison, we design a unified unwinding strategy to ensure that all tools run on the same loop-free programs. According to the detailed experimental results and analysis, we are confident about the effectiveness of Deagle.

7 RELATED WORK

Concurrent program verification is complicated due to nondeterminism caused by thread interleaving. There are many studies on improving the ability and efficiency of concurrent program verification, including bounded model checking [Alglave et al. 2013, 2012; Cordeiro and Fischer 2011; Kroening and Tautschnig 2014; Ponce-de León et al. 2020], abstraction refinement [Günther et al. 2016; Gupta et al. 2015; Jhala et al. 2018; Yin et al. 2020], stateless model checking [Abdulla et al. 2019, 2017; Godefroid 1997; Godefroid et al. 1996; Kokologiannakis et al. 2017; Kokologiannakis and Vafeiadis 2021; Oberhauser et al. 2021], sequentialization [Inverso et al. 2014; Tomasco et al. 2016], etc.

The popular partial-order-based framework [Alglave et al. 2013, 2012] encodes the happens-before relation over memory access events using integer-valued clocks, and order constraints over events are represented as differences among clock variables. then encodes these order constraints into a formula and employs a constraint solver to check their total ordering. Fan et al. [Fan et al. 2022] proposes an idea to utilize the domain knowledge of multi-threaded program in constraint solving. They assign ordring variables higher priority during SMT solving and achieve higher efficiency.

However, encoding the deductions of all possible order constraints into the formula is expensive and unnecessary since lots of order constraints cannot exist simultaneously; only a small portion of deductions take effect while the solver always considers all. Our symbolic encoding is simpler. We only encode program-related orders and read-from order constraints into the verification condition formula. As a complement, we propose several axiomatic rules to deduce necessary order constraints during constraint solving.

Moreover, the happens-before relation is partially ordered, but representing it with a less-than relation on integer-valued clocks unnecessarily lifts it to the total order. To avoid this verbosity, many previous works [He et al. 2021; Yin et al. 2018b,a; Yin et al. 2020] reduce consistency checking of a concurrent program's execution to cycle detection on an *event order graph* (*EOG*). He et al. [He et al. 2021] propose a new ordering SMT theory and develop a dedicated theory solver (Zord based on Z3 [de Moura and Bjørner 2008]) for consistency checking with incremental cycle detection. They also employ *from-read* propagation and *unit-edge* propagation to achieve higher efficiency. Compared to those studies, we also maintain a graph (event graph) but employ preventive reasoning to prevent cycles from occurring in the graph. In this way, we bypass consistency checking and achieve higher efficiency.

Yin et al. [Yin et al. 2018b,a; Yin et al. 2020] propose *SCAR* (*scheduling constraints-based abstraction refinement*), an efficient SAT-based framework for verifying concurrent programs. They completely ignore the happens-before relation initially. Instead, in each iteration, after the SAT solver returns a counterexample, SCAR builds EOG from the counterexample, finds cycles in the EOG (if a cycle is found, this counterexample is impossible in practice), and adds conflict clauses to enhance the formula. They maintain the transitive closure of edges to accelerate cycle detection. A partial assignment that causes ordering inconsistency may occur during SAT solving. However, only after the SAT solver completes the assignment can SCAR discover the inconsistency. Moreover, the SAT solver runs from scratch in each refinement round, so assignments unrelated to the consistency may be performed repeatedly. In comparison, the online strategy our method employs avoids these drawbacks. Once a variable related to happens-before orders is assigned *true*, the consistency-preserving propagation conducts exhaustive axiomatic deduction and preventive reasoning to update the event graph while preserving consistency.

Loops (and recursive functions) are the main hurdle for program verification. *Bounded model checking* (*BMC*) sets an upper execution bound for loops and unwinds the input program to a loop-free bounded program. Using BMC, a verification tool can only find counterexamples within the upper bound or confirm that correctness properties hold within the upper bound. Though incomplete, BMC is practical in finding bugs within certain iterations of loops, therefore adopted by most verification tools. CBMC and tools derived from it (Zord, Deagle, Yogar-CBMC) all employ BMC when input programs contain loops or recursions. Based on the BMC and the *round-robin schedule* mechanic, which simulates the thread interleaving process, Inverso et al. [Inverso et al. 2014] propose *lazy sequentialization* to transform a multi-threaded program into a set of single-threaded programs with the same correctness property. Their method is implemented in Lazy-CSeq [Inverso et al. 2015], the `ConcurrencySafety` category winner of `SV-COMP 2020` and `SV-COMP 2021`. Unlike Lazy-CSeq, we focus on a lightweight and essential perspective to model concurrent programs, i.e., the happens-before relation. We model the interleaving semantic into order constraints on memory access events and develop a dedicated solver with preventive reasoning mechanics to solve these constraints. We also perform extensive experiments with Lazy-CSeq.

Stateless model checking (*SMC*) [Godefroid 1997] verifies each possible execution trace of a concurrent program iteratively. Since thread interleaving results in numerous traces, SMC utilizes *partial order reduction* (*POR*) techniques [Abdulla et al. 2014; Huang 2015] to classify traces into equivalent classes and visit each equivalent class once. Many kinds of equivalent classes have been developed,

including Mazurkiewicz equivalence [Mazurkiewicz 1986], observation equivalence [Chalupa et al. 2017], reads-value-from equivalence [Agarwal et al. 2021], etc. Since partial order reduction is also dynamically conducted on the event graph, combining preventive reasoning with partial order reduction could provide further pruning the search space and achieving higher efficiency.

8 CONCLUSION AND FUTURE WORK

In this paper, we propose a theory solver that preserves ordering consistency for partial-order-based concurrent program verification. The underlying idea is preventive reasoning, which detects and disables fragile assignments in advance to prevent them from causing ordering inconsistency. We prove that the theory propagation procedure with preventive reasoning can preserve ordering consistency of the theory solver. This means the regularly-required consistency checking and conflict clause generation can be safely skipped. We implement this approach in a prototype tool called Deagle and compare it with several state-of-the-art concurrent program verifiers on credible benchmarks. The experimental results indicate that Deagle is effective and efficient in most cases.

Event graph records the set of happens-before orders with respect to the current assignment mapping. It conveys important information about the ordering variables. So far, we only utilize this information in our theory solver. Actually, the conveyed information may also be utilized by the core solver. We are planning to optimize the core solver by utilizing this information.

ACKNOWLEDGMENTS

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

REFERENCES

Parosh Abdulla, Stavros Aronis, Bengt Jonsson, and Konstantinos Sagonas. 2014. Optimal Dynamic Partial Order Reduction. *SIGPLAN Not.* 49, 1 (Jan 2014), 373–384. https://doi.org/10.1145/2578855.2535845

Parosh Abdulla, Mohamed Faouzi Atig, Bengt Jonsson, Magnus Lång, Phong Ngo, and Konstantinos Sagonas. 2019. Optimal stateless model checking for reads-from equivalence under sequential consistency. *Proceedings of the ACM on Programming Languages* 3 (10 2019), 1–29. https://doi.org/10.1145/3360576

Parosh Aziz Abdulla, Stavros Aronis, Mohamed Faouzi Atig, Bengt Jonsson, Carl Leonardsson, and Konstantinos Sagonas. 2017. Stateless Model Checking for TSO and PSO. *Acta Inf.* 54, 8 (Dec 2017), 789–818. https://doi.org/10.1007/s00236-016-0275-0

Pratyush Agarwal, Krishnendu Chatterjee, Shreya Pathak, Andreas Pavlogiannis, and Viktor Toman. 2021. Stateless Model Checking Under a Reads-Value-From Equivalence. In *Computer Aided Verification: 33rd International Conference, CAV 2021, Virtual Event, July 20–23, 2021, Proceedings, Part I.* Springer-Verlag, Berlin, Heidelberg, 341–366. https://doi.org/10.1007/978-3-030-81685-8_16

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. https://doi.org/10.1007/978-3-319-08867-9_33

Jade Alglave, Daniel Kroening, and Michael Tautschnig. 2013. Partial Orders for Efficient Bounded Model Checking Of Concurrent Software. In *Proceedings of the 25th International Conference on Computer Aided Verification - Volume 8044* (Saint Petersburg, Russia) *(CAV 2013)*. Springer-Verlag, Berlin, Heidelberg, 141–157. https://doi.org/10.1007/978-3-642-39799-8_9

Jade Alglave, Luc Maranget, Susmit Sarkar, and Peter Sewell. 2012. Fences in Weak Memory Models (Extended Version). *Form. Methods Syst. Des.* 40, 2 (Apr 2012), 170–205. https://doi.org/10.1007/s10703-011-0135-z

Clark Barrett and Cesare Tinelli. 2018. *Satisfiability Modulo Theories.* Springer International Publishing, Cham, 305–343. https://doi.org/10.1007/978-3-319-10575-8_11

Marek Chalupa, Krishnendu Chatterjee, Andreas Pavlogiannis, Nishant Sinha, and Kapil Vaidya. 2017. Data-Centric Dynamic Partial Order Reduction. *Proc. ACM Program. Lang.* 2, POPL, Article 31 (Dec 2017), 30 pages. https://doi.org/10.1145/3158119

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. https://doi.org/10.1023/A:1011276507260

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* (Waikiki, Honolulu, HI, USA) *(ICSE '11)*. Association for Computing Machinery, New York, NY, USA, 331–340. https://doi.org/10.1145/1985793.1985839

Leonardo de Moura and Nikolaj Bjørner. 2008. Z3: an efficient SMT solver. *Tools and Algorithms for the Construction and Analysis of Systems* 4963, 337–340. https://doi.org/10.1007/978-3-540-78800-3_24

Leonardo De Moura and Nikolaj Bjørner. 2011. Satisfiability modulo Theories: Introduction and Applications. *Commun. ACM* 54, 9 (Sept. 2011), 69–77. https://doi.org/10.1145/1995376.1995394

Hongyu Fan, Weiting Liu, and Fei He. 2022. Interference Relation-Guided SMT Solving for Multi-Threaded Program Verification. In *Proceedings of the 27th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming* (Seoul, Republic of Korea) *(PPoPP '22)*. Association for Computing Machinery, New York, NY, USA, 163–176. https://doi.org/10.1145/3503221.3508424

Patrice Godefroid. 1997. Model Checking for Programming Languages Using VeriSoft. In *Proceedings of the 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (Paris, France) *(POPL '97)*. Association for Computing Machinery, New York, NY, USA, 174–186. https://doi.org/10.1145/263699.263717

Patrice Godefroid, Jan van Leeuwen, Juris Hartmanis, Gerhard Goos, and Pierre Wolper. 1996. *Partial-order methods for the verification of concurrent systems: an approach to the state-explosion problem*. Vol. 1032. Citeseer.

Henning Günther, Alfons Laarman, and Georg Weissenbacher. 2016. Vienna Verification Tool: IC3 for Parallel Software. In *Proceedings of the 22nd International Conference on Tools and Algorithms for the Construction and Analysis of Systems - Volume 9636*. Springer-Verlag, Berlin, Heidelberg, 954–957. https://doi.org/10.1007/978-3-662-49674-9_69

Ashutosh Gupta, Thomas A. Henzinger, Arjun Radhakrishna, Roopsha Samanta, and Thorsten Tarrach. 2015. Succinct Representation of Concurrent Trace Sets. In *Proceedings of the 42nd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2015, Mumbai, India, January 15-17, 2015*, Sriram K. Rajamani and David Walker (Eds.). ACM, 433–444. https://doi.org/10.1145/2676726.2677008

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* (Virtual, Canada) *(PLDI 2021)*. Association for Computing Machinery, New York, NY, USA, 1264–1279. https://doi.org/10.1145/3453483.3454108

Jeff Huang. 2015. Stateless Model Checking Concurrent Programs with Maximal Causality Reduction. *SIGPLAN Not.* 50, 6 (Jun 2015), 165–174. https://doi.org/10.1145/2813885.2737975

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* (Lincoln, Nebraska) *(ASE '15)*. IEEE Press, 807–812. https://doi.org/10.1109/ASE.2015.108

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 *Proceedings of the 16th International Conference on Computer Aided Verification - Volume 8559*. Springer-Verlag, Berlin, Heidelberg, 585–602. https://doi.org/10.1007/978-3-319-08867-9_39

Ranjit Jhala, Andreas Podelski, and Andrey Rybalchenko. 2018. *Predicate Abstraction for Program Verification*. Springer International Publishing, Cham, 447–491. https://doi.org/10.1007/978-3-319-10575-8_15

Michalis Kokologiannakis, Ori Lahav, Konstantinos Sagonas, and Viktor Vafeiadis. 2017. Effective Stateless Model Checking for C/C++ Concurrency. *Proc. ACM Program. Lang.* 2, POPL, Article 17 (Dec 2017), 32 pages. https://doi.org/10.1145/3158105

Michalis Kokologiannakis and Viktor Vafeiadis. 2021. *GenMC: A Model Checker for Weak Memory Models*. 427–440. https://doi.org/10.1007/978-3-030-81685-8_20

Daniel Kroening and Michael Tautschnig. 2014. CBMC–C bounded model checker. In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer, 389–391. https://doi.org/10.1007/978-3-642-54862-8_26

Leslie Lamport. 1979. How to make a multiprocessor computer that correctly executes multiprocess programs. *IEEE transactions on computers* 28, 09 (1979), 690–691. https://doi.org/10.1109/TC.1979.1675439

Antoni Mazurkiewicz. 1986. Trace theory. In *Advanced course on Petri nets*. Springer, 278–324. https://doi.org/10.1007/3-540-17906-2_30

Jonas Oberhauser, Rafael Chehab, Diogo Behrens, Ming Fu, Antonio Paolillo, Lilith Oberhauser, Koustubha Bhat, Yuzhong Wen, Haibo Chen, Jaeho Kim, and Viktor Vafeiadis. 2021. VSync: push-button verification and optimization for synchronization primitives on weak memory models. 530–545. https://doi.org/10.1145/3445814.3446748

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* (Munich, Germany) *(TPHOLs '09)*. Springer-Verlag, Berlin, Heidelberg, 391–407. https://doi.org/10.1007/978-3-642-03359-9_27

Hernán Ponce-de León, Florian Furbach, Keijo Heljanko, and Roland Meyer. 2020. Dartagnan: Bounded Model Checking for Weak Memory Models (Competition Contribution). In *Tools and Algorithms for the Construction and Analysis of Systems: 26th International Conference, TACAS 2020, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2020, Dublin, Ireland, April 25–30, 2020, Proceedings, Part II* (Dublin, Ireland). Springer-Verlag, Berlin, Heidelberg, 378–382. https://doi.org/10.1007/978-3-030-45237-7_24

Nishant Sinha and Chao Wang. 2011. On Interference Abstractions. In *Proceedings of the 38th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (Austin, Texas, USA) *(POPL '11)*. Association for Computing Machinery, New York, NY, USA, 423–434. https://doi.org/10.1145/1926385.1926433

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 *2016 Formal Methods in Computer-Aided Design (FMCAD)*. IEEE, 193–200. https://doi.org/10.1109/FMCAD.2016.7886679

Chao Wang, Sudipta Kundu, Malay Ganai, and Aarti Gupta. 2009. Symbolic Predictive Analysis for Concurrent Programs, Vol. 5850. 256–272. https://doi.org/10.1007/978-3-642-05089-3_17

D. Weaver and Tom Gremond. 1994. The SPARC architecture manual : version 9. Prentice-Hall.

Liangze Yin, Wei Dong, Wanwei Liu, Yunchou Li, and Ji Wang. 2018b. 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. https://doi.org/10.1007/978-3-319-89963-3_25

Liangze Yin, Wei Dong, Wanwei Liu, and Ji Wang. 2017. Scheduling Constraint Based Abstraction Refinement for Multi-Threaded Program Verification. *IEEE Transactions on Software Engineering* PP (08 2017). https://doi.org/10.1109/TSE.2018.2864122

Liangze Yin, Wei Dong, Wanwei Liu, and Ji Wang. 2018a. Scheduling Constraint Based Abstraction Refinement for Weak Memory Models. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering* (Montpellier, France) *(ASE 2018)*. Association for Computing Machinery, New York, NY, USA, 645–655. https://doi.org/10.1145/3238147.3238223

L. Yin, W. Dong, W. Liu, and J. Wang. 2020. On Scheduling Constraint Abstraction for Multi-Threaded Program Verification. *IEEE Transactions on Software Engineering* 46, 5 (may 2020), 549–565. https://doi.org/10.1109/TSE.2018.2864122