Satisfiability Modulo Ordering Consistency Theory for Multi-threaded Program Verification

Fei He[*†]
School of Software
Tsinghua University
Beijing, China
hefei@tsinghua.edu.cn

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

Abstract

Analyzing multi-threaded programs is hard due to the number of thread interleavings. Partial orders can be used for modeling and analyzing multi-threaded programs. However, there is no dedicated decision procedure for solving partial-order constraints. In this paper, we propose a novel *ordering consistency theory* for multi-threaded program verification under sequential consistency, and we elaborate its theory solver, which realizes incremental consistency checking, minimal conflict clause generation, and specialized theory propagation to improve the efficiency of SMT solving. We conducted extensive experiments on credible benchmarks; the results show significant promotion of our approach.

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

Keywords: Program verification, satisfiability modulo theory, memory consistency model, concurrency

ACM Reference Format:
Fei He, Zhihang Sun, and Hongyu Fan. 2021. Satisfiability Modulo Ordering Consistency Theory for Multi-threaded Program Verification. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation (PLDI '21), June 20–25, 2021, Virtual, Canada.* ACM, New York, NY, USA, 16 pages. https://doi.org/10.1145/3453483.3454108

[*]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.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than the author(s) must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.
PLDI '21, June 20–25, 2021, Virtual, Canada
© 2021 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ACM ISBN 978-1-4503-8391-2/21/06…$15.00
https://doi.org/10.1145/3453483.3454108

1 Introduction

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

A *memory consistency model* [6] restricts the execution order of shared-memory accesses from different threads. It determines what value(s) a read access can return. In this paper, we follow the well-known *sequential consistency (SC)* model [42], where an execution path is an interleaving of instructions from different threads.

A promising technique for verifying multi-threaded programs is *bounded model checking* [16, 19] using *partial orders* to represent the happens-before relation between shared-memory access events [10, 11]. In this way, one can specify the uncertain and complicated interleaving behaviors of multi-threaded programs.

A popular approach (e.g., in [9–11, 50, 56]) for solving partial order constraints is based on *integer difference logic*. Each event is associated with an integer-valued clock, and event orders are represented as differences among these clock variables. Then the partial order constraints can be solved by the decision procedure of integer difference logic. This approach determines a clock value for each event, which goes a little bit too far because we only care about the events' order, not their exact clock values.

Moreover, there is an important axiom (Axiom 2 in Section 4) in reasoning about multi-threaded programs, which defines the derivation rule for the so-called *from-read* orders. Existing approaches [9–11, 29, 50] encode all possible from-read constraints, irrespective of whether they are actually needed for verification. This method yields a large number of from-read constraints, which significantly increases the burden on the solver and degenerates its performance.

In this paper, we propose a new and novel *ordering consistency* $\mathcal{T}_{ord}$ theory (see Section 4) and elaborate its theory solver (see Section 5) for multi-threaded program verification. We no longer need to specify all possible from-read orders in the encoding formula. One direct benefit is the significant reduction in the size of the encoding formula. Another benefit is *on-demand* deduction of from-read orders.

With a specialized theory propagation procedure (see Section 5.4), a from-read order is derived only when the relevant variables get assigned. In this way, we avoid the generation of massive useless from-read constraints.

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

Last but not least, inspired by the idea of *unit clause propagation*, we propose a novel technique for theory propagation. We attempt to find the so-called *unit edges* with this technique and use these edges to enforce values of some unassigned variables (see Section 5.4). The decision iterations of DPLL(T) can therefore be greatly reduced, and the whole performance is significantly improved.

We have implemented the proposed approach in CBMC [41] and Z3 [23], and conducted experiments on 1061 credible benchmarks in the `ConcurrencySafety` category of `SV-COMP` 2019. We compare our approach with state-of-the-art concurrent verification tools, namely CBMC [10], Lazy-CSeq [36], CPA-Seq [14, 15], and Dartagnan [28]. Our approach solves 38, 119, and 897 more cases than CBMC, CPA-Seq, and Dartagnan, respectively, and 6 less cases than Lazy-CSeq. Counting on both-solved cases, our approach runs 2.33x, 90.04x, 139.47x and 7.20x faster, consumes 18.7%, 99.6%, 99.0% and 94.5% less memory, than CBMC, CPA-Seq, Dartagnan, and Lazy-CSeq, respectively.

We have also compared our approach with state-of-the-art stateless model checking tools, namely, Nidhugg/rfsc[4] and GenMC [39] on 9 benchmarks from the Nidhugg suite. Experimental results show that as the scale (measured by the number of traces) of the program increases, our approach is superior to these tools in most cases.

In summary, our main contributions are:

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

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

2 Preliminaries

2.1 Notions

In *first-order logic*, a *term* is a variable, a constant, or an n-ary function applied to n terms; an *atom* is $\bot$, $\top$, or an n-ary predicate applied to n terms; a *literal* is an *atom* or its negation. A *first-order formula* is built from literals using Boolean connectives and quantifiers. A *model* M consists of a non-empty object set $dom(M)$, called the *domain* of M, an assignment that maps each variable to an object in $dom(M)$, and an interpretation for each constant, function and predicate, respectively. A formula Φ is *satisfiable* if there exists a model M so that $M \models \Phi$; Φ is *valid* if for any model M, $M \models \Phi$.

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

2.2 Satisfiability Modulo Theory and DPLL(T)

The *satisfiability modulo theories (SMT)* problem [12, 23, 24] is a decision problem for formulas in some combination of first-order background theories. A *theory solver* is required for each background theory $\mathcal{T}$, called $\mathcal{T}$-solver, with which the $\mathcal{T}$-satisfiability of any conjunction of literals in $\mathcal{T}$ can be determined.

DPLL(T) is the standard framework for solving SMT instances. It extends the classical DPLL algorithm [22, 45] with dedicated theory solvers. Figure 1 shows a high-level overview of DPLL(T). Given an SMT formula Ψ, DPLL(T) first replaces each atom with a fresh Boolean variable. This process is called *Boolean abstraction* because the resulting formula, denoted by $\mathcal{B}(\Psi)$, is an over-approximation of the original formula Ψ with respect to satisfiability. The satisfiability of $\mathcal{B}(\Psi)$ can be determined by a SAT solver. If $\mathcal{B}(\Psi)$ is unsatisfiable, so is Ψ; but the reverse may not hold. If $\mathcal{B}(\Psi)$ is satisfiable and M is the satisfying model returned by the SAT solver, we need to go ahead to check whether M is consistent with the underlying first-order theories.

A theory solver can be integrated with DPLL(T) in an *online* or *offline* scheme. Let M be the current (partial) assignment to $\mathcal{B}(\Psi)$. In the online scheme, $\mathcal{T}$-solver checks $\mathcal{T}$-consistency of M as long as M changes (even when M

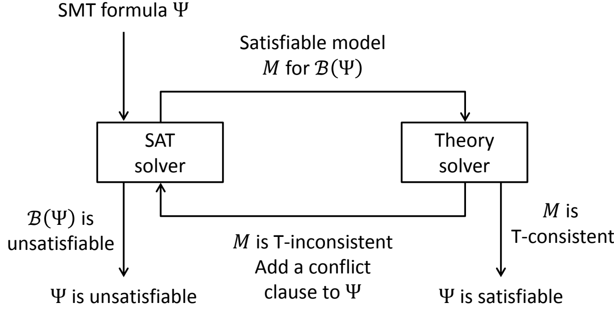


Figure 1. Flow of DPLL(T)

is a partial assignment); in the offline scheme, consistency checking is involved only when M is a satisfying model of $\mathcal{B}(\Psi)$. If M is $\mathcal{T}$-inconsistent, $\mathcal{T}$-solver attempts to generate a *conflict clause* and adds it to the clause set to prevent the solver from repeating the same path in the future. A typical theory solver also supports *theory propagation*, which deduces values of unassigned literals by theory axioms.

2.3 Concurrent Executions as Partial Orders

Our approach is built on the framework of Alglave et al. [10], which models concurrent executions using partial orders.

An *event* e is a *read* or *write* memory access with the following attributes:

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

Let $\mathbb{E}$ be the set of all events. There are some relations over events in $\mathbb{E}$. The *program order* relation $\prec_{po}$ is a total order of events from the same processor. The *write serialization* relation $\prec_{ws}$ is a total order of writes with the same address. The *read-from* relation $\prec_{rf}$ links a write event e_1 (with $type(e_1) = \mathtt{W}$) to a read event e_2 (with $type(e_2) = \mathtt{R}$), so that e_2 reads the value written by e_1.

Moreover, given a pair of write events e_1, e_2 (with $type(e_1) = type(e_2) = \mathtt{W}$) and a read event e_3 (with $type(e_3) = \mathtt{R}$) so that $e_1 \prec_{ws} e_2$ and $e_1 \prec_{rf} e_3$, we know that e_1 happens before e_2, and e_3 reads from e_1. To ensure that e_3 does not read from e_2, e_3 must happen before e_2. We call such relation the *from-read* relation $\prec_{fr}$.

An execution is *valid* if $\prec_{po} \cup \prec_{rf} \cup \prec_{ws} \cup \prec_{fr}$ does not form a cycle, i.e., there exists a *linearization* of events on this execution. An execution is *correct* if it satisfies the correctness condition. An incorrect execution is also called a *counterexample*. A program is *correct* iff it does not contain any valid counterexample.

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

(a) The original program

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

(b) The SSA form

Figure 2. A two-threaded program

3 Symbolic Encoding of Multi-threaded Programs

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

3.1 Symbolic Encoding

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

We first convert the original program to its *static single assignments (SSA)* form [21], shown in Figure 2b, where each occurrence (no matter write or read) of each shared variable is replaced with a fresh copy of this variable. A similar SSA transformation procedure is adopted in [10, 50, 56].

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

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

Value Assignment Encoding. Value assignments of variables in each thread can be encoded by directly interpreting the SSA statements. Considering thread t_1, the encoding $\rho_{va}^{t_1}$

of its value assignments is:

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

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

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

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

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

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

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

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

However, the execution corresponding to this model is invalid. Let us consider variable x. Recall that $(\!|x_1|\!)^w$ and $(\!|x_5|\!)^w$ are write accesses, $(\!|x_2|\!)^r$ and $(\!|x_4|\!)^r$ are read accesses. By $x_1 = 0, x_5 = 1$ and $x_4 = 0$, $(\!|x_4|\!)^r$ must read from $(\!|x_1|\!)^w$. Moreover, since $(\!|x_2|\!)^r$ happens between $(\!|x_1|\!)^w$ and $(\!|x_4|\!)^r$, $(\!|x_2|\!)^r$ should also read (the value of 0) from $(\!|x_1|\!)^w$, which contradicts the assignment $x_2 \mapsto 1$ in the model.

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

Program Order Constraints. We use ρ_{po} to encode the program order constraints. Program order in a thread represents the natural order of access events in this thread. For example, the program order $\rho_{po}^{t_1}$ of t_1 is:

$$(\!|x_2|\!) \prec_{po} (\!|m_3|\!) \prec_{po} (\!|x_3|\!) \prec_{po} (\!|m_4|\!) \prec_{po} (\!|x_4|\!) \prec_{po} (\!|y_2|\!)$$

Moreover, since t_1 and t_2 are child threads of `main`, all events in t_1 and t_2 should happen between (in *PO*) the invocations to `pthread_create` and `pthread_join`, respectively.

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

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

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

- *RF-Val constraint*: if $rf_{j,i}^x$ is *true*, both $(\!|x_i|\!)^r$ and $(\!|x_j|\!)^w$ are enabled, and their values are equal, i.e.,
$$rf_{j,i}^x \rightarrow guard((\!|x_i|\!)^r) \wedge guard((\!|x_j|\!)^w) \wedge (x_i = x_j)$$
- *RF-Ord constraint*: if $rf_{j,i}^x$ is *true*, the write event $(\!|x_j|\!)^w$ must happen before the read event $(\!|x_i|\!)^r$, i.e.,
$$rf_{j,i}^x \rightarrow (\!|x_j|\!)^w \prec_{rf} (\!|x_i|\!)^r$$
- *RF-Some constraint*: if the read event $(\!|x_i|\!)^r$ is enabled, it must obtain its value from a certain write event in $\pi((\!|x_i|\!)^r)$, i.e.,
$$guard((\!|x_i|\!)^r) \rightarrow \bigvee_{(\!|x_j|\!)^w \in \pi((\!|x_i|\!)^r)} rf_{j,i}^x$$

In the following, we use $\rho_{rf\text{-}val}$, $\rho_{rf\text{-}ord}$, and $\rho_{rf\text{-}some}$ to represent the conjunctions of all *RF-Val*, *RF-Ord*, and *RF-Some* constraints over all *RF* variables, respectively.

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

With respect to each *WS* variable, we have the following constraints:

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

In the following, we use $\rho_{ws\text{-}cond}$, $\rho_{ws\text{-}ord}$ and $\rho_{ws\text{-}some}$ to represent the conjunctions of all *WS-Cond*, *WS-Ord* and *WS-Some* constraints over all *WS* variables, respectively.

From-Read Constraints. Considering one read access $(\!|x_i|\!)^r$ and two write accesses $(\!|x_j|\!)^w$, $(\!|x_k|\!)^w$ to the same variable x, if $(\!|x_j|\!)^w$ happens before $(\!|x_k|\!)^w$ and $(\!|x_i|\!)^r$ reads from $(\!|x_j|\!)^w$, $(\!|x_i|\!)^r$ must happen before $(\!|x_k|\!)^w$; otherwise, $(\!|x_k|\!)^w$ is closer than $(\!|x_j|\!)^w$ to $(\!|x_i|\!)^r$, and $(\!|x_i|\!)^r$ should read from $(\!|x_k|\!)^w$ rather than $(\!|x_j|\!)^w$. Formally, this rule can be formulated as the following *from-read* (*FR*) constraint:

$$rf_{j,i}^x \wedge ws_{j,k}^x \rightarrow (\!|x_i|\!)^r \prec_{fr} (\!|x_k|\!)^w$$

Let ρ_{fr} denote the conjunction of all *FR* constraints.

Most existing techniques [10, 50, 52] for concurrent program verification include all *from-read* constraints in their

encoding formulas. This is a safe choice to ensure the correctness of the SMT encoding. However, it is not practical. For each *FR* constraint $rf^x_{j,i} \wedge ws^x_{j,k} \rightarrow (\!|x_i|\!)^r \prec_{fr} (\!|x_k|\!)^w$, only when both $rf^x_{j,i}$ and $ws^x_{j,k}$ are evaluated *true* can the ordering $(\!|x_i|\!)^r \prec_{fr} (\!|x_k|\!)^w$ be activated. At the beginning of SMT solving, all *RF* and *WS* variables are unassigned – none of these *FR* orders can be activated. For most of the time, only a small portion of these *FR* constraints takes effect. Maintaining such a large set of (unnecessary for most of the time) constraints is expensive for the SMT solver. As a result, the efficiency of SMT solving degenerates (see Section 6.3 for more details).

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

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

where

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

represents the data and control flow of the program, and

$$\Phi_{ord} = \rho_{po} \wedge \rho_{rf-ord} \wedge \rho_{ws-ord} \tag{3}$$

represents the order constraints of the program.

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

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

3.2 Comparison to Other Approaches

Our encoding is built on [10, 11, 50, 52, 56]. Compared to their encoding formulas, the most significant difference is that our encoding does not include *FR* constraints, which has already been discussed in the preceding section (also see Section 6.3 for experimental results).

Secondly, the way we model ordering constraints is also different. The existing techniques (e.g., [10, 29, 50]) use integer-valued clocks to model the time of occurrence for each event, and use differences between clock values to model the order among events. Then, they can rely on the integer difference logic to solve ordering constraints. However, note here we do not need to compute the exact occurrence time of each event, but only their orders. We thus intend to develop a dedicated solver for ordering consistency theory.

Thirdly, compared to the encoding in [10], the meaning of each *RF* or *WS* variable is slightly different. In our encoding, if an *RF* or a *WS* variable is assigned *true*, the two related events must both be enabled, while the encoding in [10] has no such requirement. As a result, the derivation of *FR* orders with our encoding need not consider guard conditions anymore. This change is quite important, since the guard conditions often involve arithmetic computation and data structures, which can hardly be handled by a dedicated theory solver for order constraints.

3.3 Correctness of the Encoding

Even though our encoding is slightly different from that in [10], we can prove its correctness.

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

Proof. This can be proved by showing the equisatisfiability of $\Psi \wedge \rho_{fr}$ and the encoding in [10]. The encoding formula of [10] has been proved correct (by Theorem 1 of [10]); we thus have the conclusion. ◻

4 Ordering Consistency Theory

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

4.1 Theory Definition

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

$$\Sigma_{ord} : \{e_1, e_2, \cdots, \prec_{po}, \prec_{ws}, \prec_{rf}, \prec_{fr}\},$$

where

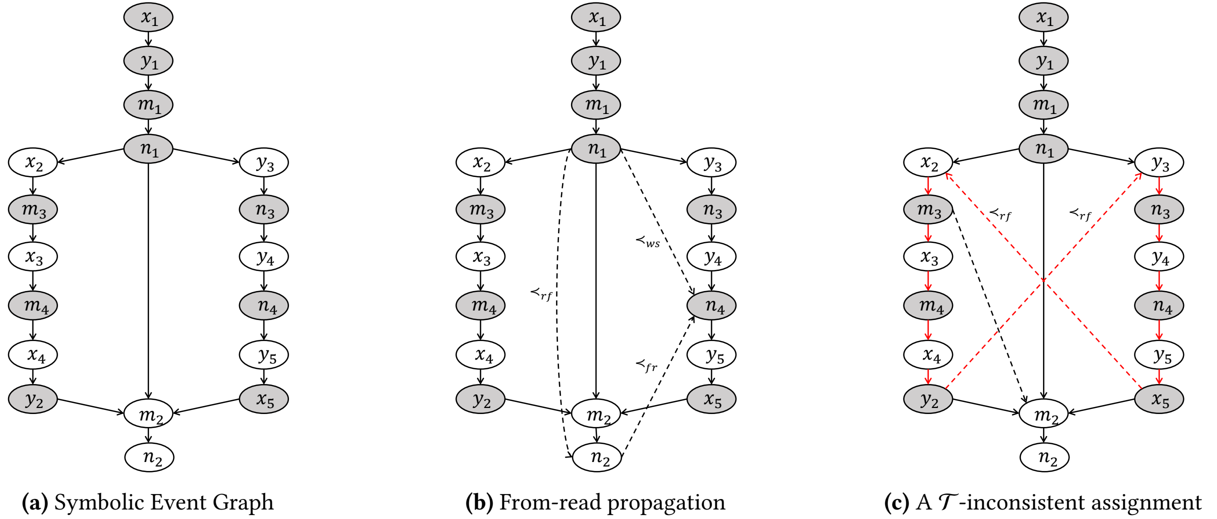
- $e_1, e_2, \cdots$ are constants, intended to represent the access events in $\mathbb{E}$.
- $\prec_{po}, \prec_{ws}, \prec_{rf}, \prec_{fr}$ are binary predicates, intended to represent the different orders among access events.

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

Now we discuss the axioms of $\mathcal{T}_{ord}$.

Axiom 1 (Partial Order)**.** *Predicates* $\prec_{po}, \prec_{ws}, \prec_{rf}, \prec_{fr}$ *in* Σ_{ord} *are* partial orders*, and*

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



(a) Symbolic Event Graph **(b)** From-read propagation **(c)** A $\mathcal{T}$-inconsistent assignment

Figure 3. Examples of theory propagation

- $\forall e_1, e_2.\ e_1 \prec_{fr} e_2 \longrightarrow type(e_1) = \mathtt{R} \land type(e_2) = \mathtt{W}$;

Axiom 2 (*FR* Derivation)**.** *For any two write events $e_1, e_2 \in \mathbb{E}$ and a read event $e_3 \in \mathbb{E}$ with $addr(e_1) = addr(e_2) = addr(e_3)$, $type(e_1) = type(e_2) = \mathtt{W}$ and $type(e_3) = \mathtt{R}$, we have:*

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

Each predicate symbol in Σ_{ord} defines a *binary relation* over $\mathbb{E}$. We use the same symbol for a predicate and the binary relation it defines.

Axiom 3 (Acyclicity)**.** *Let $\prec$ be the union $\prec_{po} \cup \prec_{ws} \cup \prec_{rf} \cup \prec_{fr}$, and $\prec^+$ the transitive closure of $\prec$, then*

$$\forall e. \neg(e \prec^+ e).$$

The above axioms define the intended semantics of $\prec_{po}$, $\prec_{ws}$, $\prec_{rf}$, $\prec_{fr}$, as we understand them in the preceding sections. Note that Axiom 3 should hold after any number of applications of Axiom 2, i.e., after deriving any number of $\prec_{fr}$ orders.

4.2 Event Graph

Let $\alpha : X_{rf} \cup X_{ws} \longrightarrow \{true, false, unassigned\}$ be the current assignment to ordering variables. Let $\prec$ be the set of orders derived from α. The event set $\mathbb{E}$ and the order set $\prec$ can be represented as a graph, called an *event graph*, where events are represented as nodes and orders as edges.

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

According to the axioms of $\mathcal{T}_{ord}$, on each edge addition, we need to check whether this new edge leads to a cycle. If this is the case, we say the current variable assignment is *invalid* – we then need to analyze the event graph to find the reason for the cycle. Otherwise, if there is no cycle, we go ahead to apply Axiom 2 to derive *FR* edges. Note that if any *FR* edge is derived, we need to check the consistency of the current variable assignment again.

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

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

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

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

Figure 3 shows three event graphs that may occur during SMT solving of the program in Figure 2. To differentiate event types, we use *grey* and *white* nodes to represent write and read events, respectively. Moreover, program orders are drawn as solid lines, while others are drawn as dashed lines. In the beginning, the event graph contains only *PO* edges, as shown in Figure 3a. After some assignments, more edges are added to the event graph. Figure 3b shows an updated

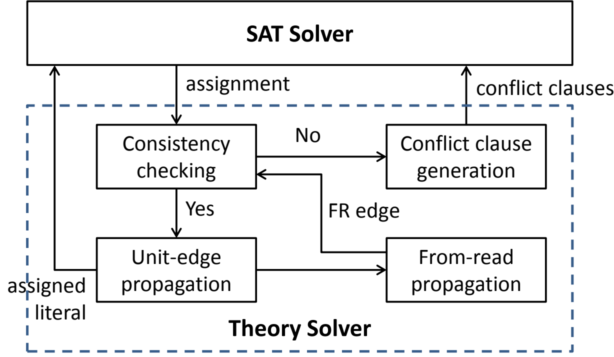


Figure 4. $\mathcal{T}_{ord}$-solver with DPLL(T)

event graph during SMT solving, in which $(\!|n_1|\!)^w \prec_{rf} (\!|n_2|\!)^r$ and $(\!|n_1|\!)^w \prec_{ws} (\!|n_4|\!)^w$ derive $(\!|n_2|\!)^r \prec_{fr} (\!|n_4|\!)^w$. Figure 3c shows another event graph, where its edges form a cycle, indicating a $\mathcal{T}_{ord}$-inconsistency.

Similar structures to our event graph were defined in [10, 50, 56]. Note that the events discussed in this paper can hold symbolic values, and thus our event graph is actually a "symbolic" event graph. In [10], a so-called *symbolic event structure* is defined, which, however, is used to depict program order only. Moreover, the *event order graph* defined in [56] represents a counterexample instead of a program. In [50], an *interference skeleton* is defined, which equals the *skeleton* of our event graph.

5 Theory Solver for $\mathcal{T}_{ord}$

This section presents our $\mathcal{T}_{ord}$-solver, with emphasis on algorithms for consistency checking, conflict clause generation, and theory propagation.

5.1 Overview

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

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

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

5.2 Consistency Checking

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

Due to complicated thread interactions, numerous ordering variables need to be assigned in DPLL(T), leading to massive and frequent consistency checking. Besides, *from-read propagation* can cause more edge insertions and more consistency checking. The previous work [9] suggests a fresh cycle detection on each consistency checking, which is inefficient. Note that before an edge addition, the event graph must be acyclic – otherwise, it must have been recognized in the previous consistency checking. It is thus more practical to perform cycle detection incrementally.

We employ an *incremental cycle detection (ICD)* algorithm [8, 13] to check $\mathcal{T}_{ord}$-consistency. Each node in the graph is labeled with a topological order [32], which is required to be consistent with edges in the graph. A topological order exists for a directed graph (including the event graph) iff this graph is acyclic. Once a new edge is inserted into the event graph, $\mathcal{T}_{ord}$-solver reuses the previous topological order and attempts to compute a new topological order incrementally. If a new topological order is computed, the event graph is acyclic, and the current assignment is $\mathcal{T}_{ord}$-consistent. Otherwise, a $\mathcal{T}_{ord}$-inconsistency is reported.

The employed ICD algorithm is based on a *two-way search* on the event graph and applies pseudo-topological order instead of topological order. Assuming an edge from e_i to e_j is to be added, if $ord(e_i) < ord(e_j)$, the algorithm directly adds the edge. Otherwise, ICD performs a backward search to traverse along incoming edges from e_i and records the traversed nodes in set $\mathbb{B}$. If e_j is visited during the backward search, a cycle is detected. Otherwise, the ICD algorithm continues to perform a forward search to traverse along outgoing edges from e_j and records the traversed nodes in set $\mathbb{F}$. If any node in $\mathbb{B}$ is traversed in the forward search, a cycle is detected. Note that the detected cycle is not necessarily the minimal one. We rely on the *conflict clause generation* algorithm to produce the "shortest" cycles.

Please refer to [13] for the detailed procedure, correctness, and complexity of the two-way-search ICD algorithm. For an event graph with n nodes and m edges, each call to the ICD algorithm takes $O(min\{n^{\frac{2}{3}}, m^{\frac{1}{2}}\})$ time.

5.3 Conflict Clause Generation

If a $\mathcal{T}_{ord}$-inconsistency occurs, we need to find the inconsistency reason and report it to the SAT solver.

To find the inconsistency reason, it is sufficient to consider *critical cycles* [49]. Formally, a cycle is *critical* if it is *simple* (i.e., no duplicate nodes) and has no *PO-chords* [49]. Recall that $\mathcal{T}_{ord}$-solver records a *derivation reason* for each edge, and the *derivation reason* of a path can be calculated accordingly

(see Section 4.2). The *derivation reason* of critical cycles can be returned as the inconsistency reason.

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

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

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

- *Step 1 (Subgraph construction).* We construct a subgraph of the event graph by removing all nodes other than $\mathbb{E}_{j-i}$, and deleting non-*PO* edges that have a *PO* chord (e.g., $(\!|n_1|\!)^w \prec_{rf} (\!|n_2|\!)^r$ in Figure 3b).
- *Step 2 (Iterative solving).* We traverse the subgraph in topological order. Let e_n be the current node to be visited. Denote $SP(e_n)$ the set of *shortest predecessors* of e_n such that the paths $e_j \prec^+ SP(e_n) \prec e_n$ have the shortest *width*. We lift $\wedge$ operator to sets, and compute $reasons(e_n)$ as

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

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

We show the correctness and complexity of our conflict clause generation algorithm by the following theorems. Their proofs are omitted due to the page limit.

Theorem 2. *The above conflict clause generation algorithm finds all conflict clauses with the shortest width.*

Theorem 3. *The time complexity of our conflict clause generation algorithm is $O(c \times m')$, where c is the number of computed conflict clauses and m' is the number of edges in the constructed subgraph.*

5.4 Theory Propagation

During theory propagation, $\mathcal{T}_{ord}$-solver deduces values of unassigned literals (by *unit-edge propagation*) and derives from-read orders (by *from-read propagation*).

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

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

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

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

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

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

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

5.5 Example

Let us consider how to verify the program example in Figure 2 using $\mathcal{T}_{ord}$-solver integrated with DPLL(T).

Given the encoding formula Ψ, DPLL(T) first applies unit-clause propagation and theory propagation to make as many as possible deductions. As a result, many *WS* variables are

assigned, e.g., $ws^x_{1,5}$, $ws^y_{1,2}$ are assigned *true*, and $ws^x_{5,1}$, $ws^y_{2,1}$, $ws^m_{3,1}$, $ws^m_{4,1}$, $ws^m_{4,3}$, $ws^n_{3,1}$, $ws^n_{4,1}$, $ws^n_{4,3}$ are assigned *false*.

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

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

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

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

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

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

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

6 Experimental Evaluation

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

6.1 Implementation and Setup

We implemented our techniques on top of CBMC [41] and Z3 [23]. CBMC is powerful and flexible bounded model checker for C/C++ programs and Z3 is a well-known and widely-adopted SMT solver. In our implementation, CBMC and Z3 act as the front and back ends, responsible for generating and solving SMT formulas respectively. We enhance CBMC by concerning our $\mathcal{T}_{ord}$-theory, and extend Z3 with our $\mathcal{T}_{ord}$-solver. All generated SMT formulas are in the SMT-LIB-v2.0 format. In the following, we call our enhanced implementation Zord [1].

All experiments were conducted on a computer with an Intel(R) Core(TM) i7-10710U CPU and 16 GB DDR4 memory. The operating system is ArchLinux-5.4.15. The time limit for each verification task is 900 seconds.

6.2 Experiment on SV-COMP Benchmarks

We collect benchmarks from the `ConcurrencySafety` category of `SV-COMP` 2019 [2]. This category is divided into 11 sub-categories, namely *ldv-races* (12), *pthread* (38), *atomic* (11), *C-DAC* (4), *complex* (5), *divine* (16), *driver-races* (21), *ext* (53), *lit* (11), *nondet* (6), and *wmm* (898), where the number adhered to each sub-category represents the number of programs it contains. There are 14 programs in *divine* sub-category that cannot be compiled by CBMC and are thus excluded from the benchmark set. In total, we get 1061 test cases.

We compare Zord with the following tools:

- CBMC: a tool that implements the partial-order-based verification algorithms [10], with Z3 as the underlying SMT solver.
- Lazy-CSeq: a tool that verifies concurrent programs using the lazy sequentialization schema [36].
- CPA-Seq: a configurable program verification platform [14, 15] with sequentially combined analysis strategies.
- Dartagnan: a bounded model checker [28] for concurrent program verification under various memory models.

Results. The experimental results are summarized in Table 1, where column *#Solved* shows the number of cases successfully solved by each tool, and the last three columns show data for both-solved cases.

In total, Zord solves 38 more cases than CBMC, 119 more cases than CPA-Seq, and 897 more cases than Dartagnan. The only exception is Lazy-CSeq, which solves 6 more cases than ours. Considering that Lazy-CSeq is a highly-optimized tool (winner of the `ConcurrencySafety` category of `SV-COMP` 2020), this result is acceptable. Considering the both-solved cases, Zord is 2.33x faster than CBMC, 90.04x faster than CPA-Seq, 139.47x faster than Dartagnan, and 7.20x faster than Lazy-CSeq. Meanwhile, Zord uses 18.7% less memory than CBMC, 99.6% less memory than CPA-Seq, 99.0% less memory than Dartagnan, and 94.5% less memory than Lazy-CSeq.

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

[1] https://thufv.github.io/research/zord.html
[2] https://sv-comp.sosy-lab.org/2019/

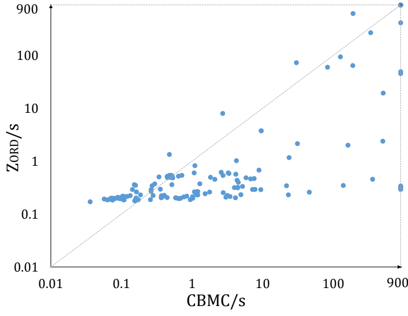


Figure 5. ZORD vs. CBMC

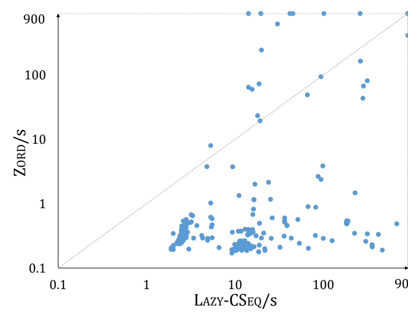


Figure 6. ZORD vs. LAZY-CSEQ

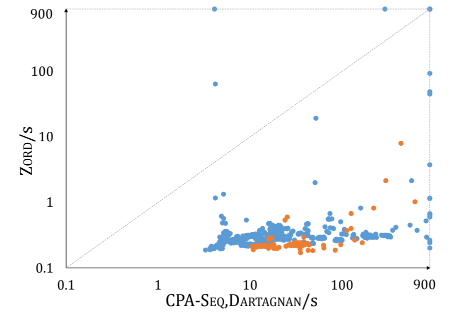


Figure 7. ZORD vs. CPA-SEQ (blue) and DARTAGNAN (orange)

concurrent programs. Table 2 lists the summary results with wmm sub-category excluded. In total, ZORD solves 38, 113, and 122 more cases than CBMC, CPA-SEQ, and DARTAGNAN, respectively, and 6 less cases than LAZY-CSEQ. Counting on the both-solved non-wmm cases, ZORD is 2.63x, 11.06x, 157.47x and 6.24x faster, and uses 18.5%, 97.5%, 99.6%, and 97.7% less memory, than CBMC, CPA-SEQ, DARTAGNAN, and LAZY-CSEQ, respectively.

Figure 5 compares ZORD with CBMC on the CPU time of each verification task. A point below (or above) the diagonal represents a case that ZORD is superior (inferior) to CBMC. Programs in wmm sub-category are all simple so that their accumulated CPU time by ZORD and CBMC are 269s and 338s, respectively; we only draw a single point in the figure to represent the whole wmm sub-category. There are a cluster of points at the bottom left of Figure 5, which indicates that these cases are solved extremely fast by both tools, and CBMC even solves slightly faster on some tasks. This is because either these tasks are trivial, or counterexamples occur at a low depth. When the cases become complex, our method starts to show its strength.

Figure 6 and Figure 7 compare ZORD with LAZY-CSEQ, and ZORD with CPA-SEQ (blue points) and DARTAGNAN (orange points) on each case, respectively. These results conform to those in Table 1. ZORD is remarkably superior to these three tools in most cases. Among all cases, ZORD outperforms DARTAGNAN. Only in 14 and 3 cases is ZORD inferior to LAZY-CSEQ and CPA-SEQ, respectively.

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

This experiment evaluates strategies of $\mathcal{T}_{ord}$-solver by testing their effects on the whole performance of program verification. Experiment settings are identical to 6.2.

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

Figure 8 shows time comparison between ZORD and ZORD$^-$ on each verification task. In most cases, ZORD is more efficient than ZORD$^-$. Among 1061 available cases, ZORD solves 1051 cases correctly, and ZORD$^-$ can solve 1049 cases. ZORD$^-$ has two more timeout cases whereas ZORD solves them within 95.4s and 248.3s. Since ZORD ignores *FR* constraints when generating the SMT formula, ZORD encodes a smaller formula than ZORD$^-$. Moreover, ZORD$^-$ uses unit clause propagation in the SAT level to assign *FR* constraints, whereas $\mathcal{T}_{ord}$-solver performs *from-read propagation* on the event graph to derive *FR* constraints with $O(1)$ complexity. So our application of the from-read axiom is more efficient. The reported *CPU_time* consumptions of ZORD$^-$ and ZORD

Table 1. Summary results on 1061 SV-COMP benchmarks

Tool	#Solved	Both-solved		
		Num.	CPU_time (s) (-/ZORD)	Memory (GB) (-/ZORD)
ZORD	1051	-	-	-
CBMC	1013	1013	2933/1257	7.61/6.19
CPA-SEQ	932	930	32866/365	1311.56/5.39
DARTAGNAN	154	154	6555/47	71.48/0.68
LAZY-CSEQ	1057	1050	14025/1949	114.15/6.33

Table 2. Summary results on 163 non-wmm SV-COMP benchmarks (i.e., with wmm sub-category excluded)

Tool	#Solved	Both-solved		
		Num.	CPU_time (s) (-/ZORD)	Memory (GB) (-/ZORD)
ZORD	153	-	-	-
CBMC	115	115	2594/988	1.84/1.50
CPA-SEQ	40	38	1106/100	29.56/0.75
DARTAGNAN	31	31	2992/19	54.19/0.21
LAZY-CSEQ	159	152	10491/1680	72.35/1.63

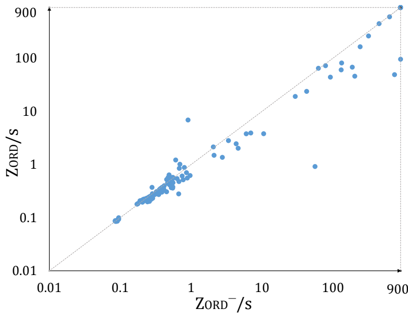


Figure 8. Zord vs. Zord$^-$

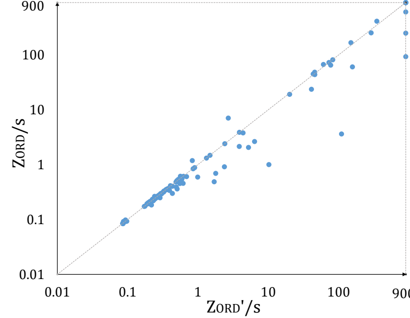


Figure 9. Zord vs. Zord$'$

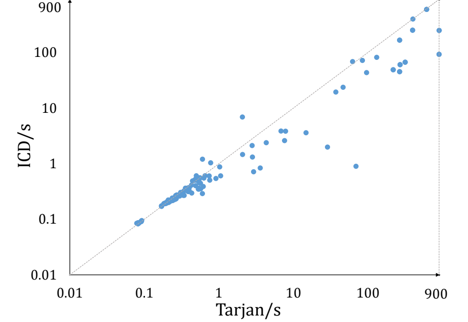


Figure 10. ICD vs. Tarjan's algorithm

are 13060.2s and 10745.6s, respectively. On 1049 both-solved cases, Zord$^-$ spends 2841.1s while Zord spends 2037.7s – these cases only take Zord 71.7% time as much as Zord$^-$. The experimental results indicate that it is more efficient to pass the derivation of *FR* constraints to $\mathcal{T}_{ord}$-solver than to encode those constraints into the SMT formula.

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

Figure 9 shows time comparison between Zord and Zord$'$ on each verification task. Among 1061 available cases, Zord$'$ verifies 1048 cases correctly, and Zord succeeds in 1051 cases – by applying *unit-edge propagation*, we solve 3 more cases within 93.0s, 606.3s, and 245.0s, respectively. On 1048 both-solved cases, Zord$'$ spends 1590.2s and Zord spends 1389.3s – the time reduction is 12.6%. Moreover, compared to Zord$'$, Zord reduces memory consumption, the number of decisions, propagations, and conflicts to 97.5%, 84.4%, 90.1%, and 79.0%, respectively. The solving procedure of Zord only involves 79.0% conflicts and 84.4% decisions as many as Zord$'$. Therefore, we confirm that *unit-edge propagation* helps DPLL(T) avoid possible cycles in advance and make fewer decisions to reduce the search space. In summary, *unit-edge propagation* is an obvious optimization for $\mathcal{T}_{ord}$-solver.

Effects of ICD Algorithm. We also implement Tarjan's non-incremental cycle detection algorithm in our $\mathcal{T}_{ord}$-solver and compare it with the ICD algorithm. Figure 10 shows the CPU_time of SMT solving with these two algorithms on each verification task. In small cases (<1s), the performance is similar whereas in most complicated cases, the ICD algorithm is more efficient than Tarjan's algorithm. Excluding both-timeout cases, the total CPU_time with Tarjan's algorithm is 4813s, and that with ICD is 2368s. SMT solving with the ICD algorithm is 2.03 times faster than that with Tarjan's algorithm. For an event graph with n nodes and m edges, the complexity of Tarjan's' algorithm is $O(m \times (n+m))$ and that of ICD is $O(m \times min\{n^{\frac{2}{3}}, m^{\frac{1}{2}}\})$. Generally, the event graph of a multi-threaded program is sparse ($m << n^2$). As programs become complex, incremental cycle detection starts to show its efficacy.

6.4 Comparison with Stateless Model Checking

Stateless model checking (SMC) [30] is another successful technique for multi-threaded program verification. It systematically explores all interleaving traces of the concurrent program. Meanwhile, various *partial order reduction* techniques [1, 17, 35, 57] are developed to alleviate explosion.

In this experiment, we compare Zord with two state-of-the-art SMC tools:

- Nidhugg/rfsc: an SMC tool that implements the reads-from equivalence exploration algorithm [4] under SC.
- GenMC: an SMC tool that implements an optimal algorithm to build execution graphs dynamically [40].

Originally, we wanted to perform this experiment on `SV-COMP` benchmarks. However, Nidhugg/rfsc and GenMC currently do not support many library functions used in `SV-COMP` benchmarks, e.g., `abort()`, `strcmp()`, `strcpy()`, and some of `__VERIFIER_*`. As a result, among the 1061 SV-COMP verification tasks, Nidhugg/rfsc and GenMC only solve 45 and 38 cases, respectively.

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

Finally, we get 9 examples: `CO-2+2W(`N`)`, `float_r(`N`)`, `airline(`N`)`, `fib_bench(`N`)`, `szymanski(`N`)`, `lamport(`N`)`, `cir_buf(`N`)`, `parker(`N`)` and `account(`N`)`, where N is a parameter.

[3]https://github.com/nidhugg/nidhugg/tree/master/benchmarks, commit 6af46b8

Table 3. Experiment results on Nidhugg benchmarks

Files	Rst	Traces	Nidhugg/rfsc	GenMC	CBMC	Zord	Files	Rst	Traces	Nidhugg/rfsc	GenMC	CBMC	Zord
`CO-2+2W(5)`	T	5	0.10	0.03	0.36	0.31	`lamport(2)`	T	1.7×10^4	2.46	0.87	463.37	4.18
`CO-2+2W(15)`	T	15	0.12	0.04	27.21	1.33	`lamport(6)`	T	1.3×10^6	282.10	65.42	TO	379.32
`CO-2+2W(25)`	T	25	0.10	0.04	307.77	6.55	`lamport(10)`	T	1.5×10^7	TO	834.33	TO	TO
`float_r(10)`	T	11	0.08	0.03	0.05	0.22	`cir_buf(5)`	T	252	0.19	0.07	TO	1.70
`float_r(50)`	T	51	0.11	0.04	TO	1.66	`cir_buf(9)`	T	4.9×10^4	25.90	12.07	TO	52.73
`float_r(100)`	T	101	0.30	0.11	TO	4.90	`cir_buf(13)`	T	-	TO	TO	TO	897.35
`airline(3)`	T	27	0.09	0.03	0.19	0.23	`parker(12)`	T	7.0×10^4	2.94	8.68	118.76	0.34
`airline(7)`	T	8.2×10^5	161.21	20.96	0.60	0.30	`parker(20)`	T	4.8×10^5	19.21	92.82	TO	0.34
`airline(9)`	T	-	TO	TO	1.44	0.50	`parker(28)`	T	1.7×10^6	52.60	500.56	TO	0.38
`fib_bench(4)`	T	3.4×10^4	2.74	0.82	TO	0.33	`account(5)`	F	3456	0.01	0.11	0.65	0.28
`fib_bench(5)`	T	5.3×10^5	37.53	14.15	TO	0.87	`account(15)`	F	-	0.01	TO	87.11	43.30
`fib_bench(6)`	T	8.2×10^6	537.98	286.76	TO	1.50	`account(25)`	F	-	0.13	TO	TO	TO
`szymanski(2)`	T	-	4.07	TO	16.90	1.66							
`szymanski(4)`	T	-	153.92	TO	271.81	6.14							
`szymanski(6)`	T	-	TO	TO	775.24	12.54							

Results. All benchmarks are verified under SC. The experimental results are listed in Table 3, where the *Rst* column reports the verification results, and the *Traces* column lists the number of traces explored by GenMC. If the result is *true*, the number reported in *Traces* is a measurement of the size of the trace space.

`CO-2+2W(`N`)` and `float_r(`N`)` are trivial examples: there is no branching statement; the `main` thread contains one read event; each child thread contains one visible atomic write. As a result, the number of traces in each program is equal (or nearly equal) to the number of child threads (i.e., N). Nidhugg/rfsc and GenMC can verify these programs in a few tenths of a second. This is understandable since the largest `float_r(100)` contains 101 traces only. Time consumptions of CBMC and Zord increase gradually on the increment of N, since the size of the encoding formula grows accordingly. However, Zord is obviously faster than CBMC. Our ordering theory and elaborated theory solver help Zord achieve higher efficiency.

`Airline(`N`)`, `fib_bench(`N`)`, and `szymanski(`N`)` contain branching statements, with N representing either the number of threads or the unrolling bound of loops. As N increases, Nidhugg/rfsc and GenMC slow down rapidly. Basically, the verification time of Nidhugg/rfsc and GenMC is proportional to the number of traces in the program. In contrast, Zord performs very well in all these programs.

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

In `parker(`N`)`, a `_parker()` procedure is invoked N times. As N increases, Nidhugg/rfsc and GenMC slow down rapidly, in line with the fast growth of traces. In contrast, Zord verifies each program in less than 0.4 seconds, and the time does not notably change as N increases. The main reasons are: 1) this example contains only one thread, and 2) Zord encodes the path condition of each memory event into the SMT formula. As a result, increasing N has no influence on the size of the SMT formula.

In program `account(`N`)`, Nidhugg/rfsc obviously outperforms others. Note that all variations of this example are buggy. It happens that Nidhugg/rfsc finds the safety violation by exploring one trace.

Summary. Firstly, as N increases, Zord outperforms CBMC significantly, demonstrating the efficiency of our $\mathcal{T}_{ord}$-theory solver. Secondly, SMC is suitable for programs with simple branching structures, while Zord is more suitable for complex programs. Thirdly, arrays and complex data structures in the encoding formula can slow down the SMT solving.

6.5 Threats to Validity

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

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

Implementation of $\mathcal{T}_{ord}$ in CBMC is simple and clear, and the implementation of $\mathcal{T}_{ord}$-solver is loosely coupled with the overall framework of Z3. Benchmarks are collected from the `ConcurrencySafety` category of `SV-COMP 2019` and Nidhugg. Many studies perform their experiments on these benchmarks to demonstrate the effectiveness of their method. These benchmarks are comprehensive, credible, and have

already been preprocessed for verification. Finally, we performed an extensive comparison with CBMC, Lazy-CSeq, CPA-Seq, Dartagnan, and two SMC techniques implemented in Nidhugg/rfsc and GenMC. The detailed results and analysis of these tools show that Zord is correct and competitive. We are thus confident in the effectiveness of Zord.

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

7 Related Work

There is much research on improving the availability and efficiency of multi-threaded program verification by constraint solving.

Multi-threaded program verification under SC has been comprehensively studied in recent years. Alglave et al. [10, 11] use a partial order relation on memory events to represent possible executions caused by thread interleaving. All partial order constraints are encoded into a formula. If no counterexample is found, the multi-threaded program is safe. Otherwise, we need to check that the counterexample is not spurious, i.e., that the corresponding execution does not form a cycle. The above technique inspires our method. We propose a new theory to denote order constraints of memory events. Then we elaborate on the theory solver of $\mathcal{T}_{ord}$ in Z3 [23]. It can benefit from optimizations of Z3 inherently and is fully compatible with DPLL(T) framework. By utilizing axioms of $\mathcal{T}_{ord}$, $\mathcal{T}_{ord}$-solver is responsible for verifying whether the current order constraints are valid, i.e., whether the current order is acyclic. In general, we develop a new partial order theory to extend the basic idea of [10, 11] with SMT-based constraints solving.

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

$\mathcal{T}_{ord}$-solver's decision procedure is closely related to incremental cycle detection (ICD), a mature algorithm with a long research history. Based on online topological ordering [41], researches on ICD begin with [43]. When an edge is inserted, it considers updating the topological order by reusing the previous order instead of calculating a new one. They propose a method that takes amortized $O(n)$ time for each edge insertion in a graph with n nodes, faster than the $O(n+m)$ offline algorithm (with m representing the number of edges). Later works [7, 8, 37, 47] and [13, 32] aim at proposing new cycle detection algorithms; some work better on sparse graphs (assuming $m = O(n)$) and some better on dense graphs (assuming $m = O(n^2)$).

To the best of our knowledge, the fastest algorithms are proposed by Bender et al. [13]. It proposes two $O(min\{m^{1/2}, n^{2/3}\}m)$ algorithms for sparse graphs and one $O(n^2 log(n))$ algorithm for dense graphs. Among them, we apply the two-way-search algorithm for sparse graphs using pseudo-topological order in our approach. This algorithm achieves $O(min\{m^{1/2}, n^{2/3}\})$ for each edge insertion by using pseudo-topological order and setting a well-designed limit to the order. However, it can only find a strongly connected component when a cycle occurs. Therefore, we extend the algorithm to searching for all critical cycles with the least non-PO edges.

Nieuwenhuis et al. [46] design an SMT implementation of difference logic, where literal $x - y \leq k$ is represented with an edge from x to y with weight k and $x - y < k$ represented with an edge with weight $k - \epsilon$ where $\epsilon = 1$ in integer difference logic and ϵ can be computed in linear time in real difference logic [48]. After adding an edge from x to y, a traverse of outgoing edges from y and a traverse of incoming edges from x are performed to check whether this edge addition forms a cycle. If a cycle on which weights of edges sum negative is found, an inconsistency occurs. Their difference logic solver's behavior after each edge addition is similar to us but far from incremental. It does not maintain a topological order, so that cannot make use of previous information. Thus, each traverse is exhaustive and takes $O(n+m)$ time in a graph with n nodes and m edges.

Ge et al.[29] propose another method to solve order constraints. They only define one sort of partial order relation, i.e., the "happens-before" relation, and unite other sorts of partial order relations to "happens-before". Then they check if there is a cycle among ordering constraints. In this manner, their theory cannot handle the intricate relation between different sorts of partial orders for concurrent program verification, which, in our understanding, is very important for the efficiency of SMT solving. In comparison, our method can model various order relations to formulate a multi-threaded program's possible executions. After each edge addition, their method calls Tarjan's algorithm to find cycles. We notice the high frequency of edge addition and deletion while solving an SMT formula with order constraints. Instead, we develop a new ordering consistency (OC) theory specifically for multi-threaded program verification and elaborate on its theory solver with incremental cycle detection to achieve higher efficiency. Using incremental cycle detection, m additions in a graph with n nodes take only $O(m \times min\{n^{\frac{2}{3}}, m^{\frac{1}{2}}\})$ time, better than Tarjan's algorithm that spends $O(m \times (n+m))$.

Multi-threaded program verification is complicated because of the uncertainty caused by thread interleaving. Too

many possible execution paths may result in the state explosion problem. The most efficient techniques to alleviate this problem include but are not limited to bounded model checking [16, 19, 27, 28, 51], partial order reduction [1, 26, 53], abstraction refinement [25, 31, 50], and stateless model checking [2, 3, 38].

Bounded model checking (BMC) sets an upper bound for loops or recursive functions to obtain a bounded program. BMC is efficient and powerful in bug-finding and therefore has been widely adopted by the majority of verification tools.

Moreover, to improve the availability of multi-threaded program verification, many researchers combine BMC with other techniques. Inverso et al. [36] propose a new approach named *Lazy Sequentialization*, which transforms a multi-threaded program into a sequential program under a specific unrolling bound and execution round. This method simulates thread interactions and is efficient in bug-finding. Yin et al. [54–56] develop a SAT-based verification framework called scheduling constraints-based abstraction refinement (SCAR) for verifying multi-threaded programs under several memory models. They ignore the order constraints first; instead, they add conflict clauses in the iterations of abstraction refinement to enhance the formula. They use the transitive closure to check the consistency of order relation, which achieves high efficiency. Cordeiro et al. [20] combine BMC with Satisfiable Modulo Theory (SMT). They encode all possible executions and utilize theory conflict to abstract thread interleaving. They implemented their tactics in ESCBMC, an SMT-based verification tool.

Stateless model checking (SMC) [30] checks correctness by enumerating all possible execution traces. Since thread interleaving results in numerous traces, traces inducing the same order between conflicting events are sorted into an equivalence class, namely, a Mazurkiewicz trace [44]. Mazurkiewicz traces can be further weakened to equivalence classes of traces with identical read-from relations [5]. Efficient traversal algorithms are developed to discover all Mazurkiewicz traces consistent with the memory model and prune inconsistent ones. Popular stateless model checkers, e.g., Nidhugg [2], Nidhugg/rfsc [5], GenMC [40], RCMC [38], DC-DPOR [18], are based on this technique.

8 Conclusion and Future Work

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

Our approach is designed for multi-threaded program verification under SC. We are planning to extend these techniques to weak memory models in the future.

Acknowledgments

The authors would like to thank the anonymous referees for their valuable comments and helpful suggestions. This work is supported by the National Natural Science Foundation of China under Grant No.: 62072267 and Grant No. 61672310, the National Key Research and Development Program of China under Grant No.: 2018YFB1308601, and the Guangdong Science and Technology Department under Grant No.: 2018B010107004.

References

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

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

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

[4] Parosh Aziz Abdulla, Mohamed Faouzi Atig, Bengt Jonsson, Magnus Lång, Tuan Phong Ngo, and Konstantinos Sagonas. 2019. Optimal Stateless Model Checking for Reads-from Equivalence under Sequential Consistency. *Proc. ACM Program. Lang.* 3, OOPSLA, Article 150 (Oct. 2019), 29 pages. https://doi.org/10.1145/3360576

[5] Parosh Aziz Abdulla, Mohamed Faouzi Atig, Bengt Jonsson, Magnus Lång, Tuan Phong Ngo, and Konstantinos Sagonas. 2019. Optimal Stateless Model Checking for Reads-from Equivalence under Sequential Consistency. *Proc. ACM Program. Lang.* 3, OOPSLA, Article 150 (Oct. 2019), 29 pages. https://doi.org/10.1145/3360576

[6] Sarita V Adve and Kourosh Gharachorloo. 1996. Shared memory consistency models: A tutorial. *computer* 29, 12 (1996), 66–76. https://doi.org/10.1109/2.546611

[7] Deepak Ajwani and Tobias Friedrich. 2007. Average-Case Analysis of Online Topological Ordering. In *Proceedings of the 18th International Conference on Algorithms and Computation* (Sendai, Japan) *(ISAAC'07)*. Springer-Verlag, Berlin, Heidelberg, 464–475. https://doi.org/10.1007/978-3-540-77120-3_41

[8] Deepak Ajwani, Tobias Friedrich, and Ulrich Meyer. 2008. An O(N2.75) Algorithm for Incremental Topological Ordering. *ACM Trans. Algorithms* 4, 4, Article 39 (Aug. 2008), 14 pages. https://doi.org/10.1145/1383369.1383370

[9] 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

[10] Jade Alglave, Daniel Kroening, and Michael Tautschnig. 2013. Partial Orders for Efficient Bounded Model Checking of Concurrent Software. In *Computer Aided Verification*, Natasha Sharygina and Helmut Veith (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 141–157. https://doi.org/10.1007/978-3-642-39799-8_9

[11] Jade Alglave, Luc Maranget, Susmit Sarkar, and Peter Sewell. 2010. Fences in Weak Memory Models. In *Computer Aided Verification*, Tayssir Touili, Byron Cook, and Paul Jackson (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 258–272. https://doi.org/10.1007/978-3-642-14295-6_25

[12] 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

[13] Michael A. Bender, Jeremy T. Fineman, Seth Gilbert, and Robert E. Tarjan. 2015. A New Approach to Incremental Cycle Detection and Related Problems. *ACM Trans. Algorithms* 12, 2, Article 14 (Dec. 2015), 22 pages. https://doi.org/10.1145/2756553

[14] Dirk Beyer, Thomas A. Henzinger, and Grégory Théoduloz. 2007. Configurable Software Verification: Concretizing the Convergence of Model Checking and Program Analysis. In *Computer Aided Verification*, Werner Damm and Holger Hermanns (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 504–518. https://doi.org/10.1007/978-3-540-73368-3_51

[15] Dirk Beyer and M. Erkan Keremoglu. 2011. CPAchecker: A Tool for Configurable Software Verification. In *Computer Aided Verification*, Ganesh Gopalakrishnan and Shaz Qadeer (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 184–190. https://doi.org/10.1007/978-3-642-22110-1_16

[16] Armin Biere, Alessandro Cimatti, Edmund Clarke, and Yunshan Zhu. 1999. Symbolic Model Checking without BDDs. In *Tools and Algorithms for the Construction and Analysis of Systems*, W. Rance Cleaveland (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 193–207. https://doi.org/10.1007/3-540-49059-0_14

[17] 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

[18] 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

[19] 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

[20] 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

[21] Ron Cytron, Jeanne Ferrante, Barry K. Rosen, Mark N. Wegman, and F. Kenneth Zadeck. 1991. Efficiently computing static single assignment form and the control dependence graph. *ACM TRANSACTIONS ON PROGRAMMING LANGUAGES AND SYSTEMS* 13 (1991), 451–490. https://doi.org/10.1145/115372.115320

[22] Martin Davis, George Logemann, and Donald Loveland. 1962. A Machine Program for Theorem-Proving. *Commun. ACM* 5, 7 (July 1962), 394–397. https://doi.org/10.1145/368273.368557

[23] Leonardo de Moura and Nikolaj Bjørner. 2008. Z3: An Efficient SMT Solver. In *Tools and Algorithms for the Construction and Analysis of Systems*, C. R. Ramakrishnan and Jakob Rehof (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 337–340. https://doi.org/10.1007/978-3-540-78800-3_24

[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

[25] Thomas Dinsdale-Young, Mike Dodds, Philippa Gardner, Matthew J. Parkinson, and Viktor Vafeiadis. 2010. Concurrent Abstract Predicates. In *ECOOP 2010 – Object-Oriented Programming*, Theo D'Hondt (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 504–528. https://doi.org/10.1007/978-3-642-14107-2_24

[26] Cormac Flanagan and Patrice Godefroid. 2005. Dynamic Partial-Order Reduction for Model Checking Software. In *Proceedings of the 32nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (Long Beach, California, USA) *(POPL '05)*. Association for Computing Machinery, New York, NY, USA, 110–121. https://doi.org/10.1145/1040305.1040315

[27] Malay K. Ganai and Aarti Gupta. 2008. Efficient Modeling of Concurrent Systems in BMC. In *Model Checking Software*, Klaus Havelund, Rupak Majumdar, and Jens Palsberg (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 114–133. https://doi.org/10.1007/978-3-540-85114-1_10

[28] Natalia Gavrilenko, Hernán Ponce-de León, Florian Furbach, Keijo Heljanko, and Roland Meyer. 2019. BMC for Weak Memory Models: Relation Analysis for Compact SMT Encodings. In *Computer Aided Verification*, Isil Dillig and Serdar Tasiran (Eds.). Springer International Publishing, Cham, 355–365. https://doi.org/10.1007/978-3-030-25540-4_19

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

[30] 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

[31] Ashutosh Gupta, Corneliu Popeea, and Andrey Rybalchenko. 2011. Predicate Abstraction and Refinement for Verifying Multi-Threaded Programs. In *Proceedings of the 38th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (Austin, Texas, USA) *(POPL '11)*. Association for Computing Machinery, New York, NY, USA, 331–344. https://doi.org/10.1145/1926385.1926424

[32] Bernhard Haeupler, Telikepalli Kavitha, Rogers Mathew, Siddhartha Sen, and Robert E. Tarjan. 2012. Incremental Cycle Detection, Topological Ordering, and Strong Component Maintenance. *ACM Trans. Algorithms* 8, 1, Article 3 (Jan. 2012), 33 pages. https://doi.org/10.1145/2071379.2071382

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

[34] Alex Horn and Daniel Kroening. 2015. On partial order semantics for SAT/SMT-based symbolic encodings of weak memory concurrency. arXiv:1504.00037 [cs.LO] https://arxiv.org/abs/1504.00037

[35] Jeff Huang. 2015. Stateless Model Checking Concurrent Programs with Maximal Causality Reduction. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Portland, OR, USA) *(PLDI '15)*. Association for Computing Machinery, New York, NY, USA, 165–174. https://doi.org/10.1145/2737924.2737975

[36] Omar Inverso, Ermenegildo Tomasco, Bernd Fischer, Salvatore La Torre, and Gennaro Parlato. 2014. Bounded Model Checking of Multi-threaded C Programs via Lazy Sequentialization. In *Computer Aided Verification*, Armin Biere and Roderick Bloem (Eds.). Springer International Publishing, Cham, 585–602. https://doi.org/10.1007/978-3-319-08867-9_39

[37] Irit Katriel and Hans L. Bodlaender. 2006. Online Topological Ordering. *ACM Trans. Algorithms* 2, 3 (July 2006), 364–379. https://doi.org/10.1145/1159892.1159896

[38] 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

[39] Michalis Kokologiannakis, Azalea Raad, and Viktor Vafeiadis. 2019. Model Checking for Weakly Consistent Libraries. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Phoenix, AZ, USA) *(PLDI 2019)*. Association for Computing Machinery, New York, NY, USA, 96–110. https://doi.org/10.1145/3314221.3314609

[40] Michalis Kokologiannakis, Azalea Raad, and Viktor Vafeiadis. 2019. Model Checking for Weakly Consistent Libraries. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Phoenix, AZ, USA) *(PLDI 2019)*. Association for Computing Machinery, New York, NY, USA, 96–110. https://doi.org/10.1145/3314221.3314609

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

[42] Leslie Lamport. 1979. How to make a multiprocessor computer that correctly executes multiprocess programs. *IEEE Computer Architecture Letters* 28, 09 (1979), 690–691. http://doi.org/10.1109/TC.1979.1675439

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

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

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

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

[47] David J. Pearce and Paul H. J. Kelly. 2007. A Dynamic Topological Sort Algorithm for Directed Acyclic Graphs. *ACM J. Exp. Algorithmics* 11 (Feb. 2007), 1.7–es. https://doi.org/10.1145/1187436.1210590

[48] Alexander Schrijver. 1986. *Theory of Linear and Integer Programming*. John Wiley & Sons, Inc., USA.

[49] Dennis Shasha and Marc Snir. 1988. Efficient and Correct Execution of Parallel Programs That Share Memory. *ACM Trans. Program. Lang. Syst.* 10, 2 (April 1988), 282–312. https://doi.org/10.1145/42190.42277

[50] 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

[51] Chao Wang, HoonSang Jin, Gary D. Hachtel, and Fabio Somenzi. 2004. Refining the SAT Decision Ordering for Bounded Model Checking. In *Proceedings of the 41st Annual Design Automation Conference* (San Diego, CA, USA) *(DAC '04)*. Association for Computing Machinery, New York, NY, USA, 535–538. https://doi.org/10.1145/996566.996713

[52] Chao Wang, Sudipta Kundu, Malay Ganai, and Aarti Gupta. 2009. Symbolic Predictive Analysis for Concurrent Programs. In *FM 2009: Formal Methods*, Ana Cavalcanti and Dennis R. Dams (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 256–272. https://doi.org/10.1007/978-3-642-05089-3_17

[53] Chao Wang, Zijiang Yang, Vineet Kahlon, and Aarti Gupta. 2008. Peephole Partial Order Reduction. In *Tools and Algorithms for the Construction and Analysis of Systems*, C. R. Ramakrishnan and Jakob Rehof (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 382–396. https://doi.org/10.1007/978-3-540-78800-3_29

[54] Liangze Yin, Wei Dong, Wanwei Liu, Yunchou Li, and Ji Wang. 2018. YOGAR-CBMC: CBMC with Scheduling Constraint Based Abstraction Refinement. In *Tools and Algorithms for the Construction and Analysis of Systems*, Dirk Beyer and Marieke Huisman (Eds.). Springer International Publishing, Cham, 422–426. https://doi.org/10.1007/978-3-319-89963-3_25

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

[56] 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

[57] Naling Zhang, Markus Kusano, and Chao Wang. 2015. Dynamic Partial Order Reduction for Relaxed Memory Models. In *Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Portland, OR, USA) *(PLDI '15)*. Association for Computing Machinery, New York, NY, USA, 250–259. https://doi.org/10.1145/2737924.2737956