Interference Relation-Guided SMT Solving for Multi-Threaded Program Verification

Hongyu Fan
School of Software, Tsinghua University
Key Laboratory for Information System Security, MoE
Beijing National Research Center for Information Science and Technology
Beijing, China
fhy18@mails.tsinghua.edu.cn

Weiting Liu
School of Software, Tsinghua University
Key Laboratory for Information System Security, MoE
Beijing National Research Center for Information Science and Technology
Beijing, China
lwt16@mails.tsinghua.edu.cn

Fei He*
School of Software, Tsinghua University
Key Laboratory for Information System Security, MoE
Beijing National Research Center for Information Science and Technology
Beijing, China
hefei@tsinghua.edu.cn

Abstract

Concurrent program verification is challenging due to a large number of thread interferences. A popular approach is to encode concurrent programs as SMT formulas and then rely on off-the-shelf SMT solvers to accomplish the verification. In most existing works, an SMT solver is simply treated as the backend. There is little research on improving SMT solving for concurrent program verification.

In this paper, we recognize the characteristics of interference relation in multi-threaded programs and propose a novel approach for utilizing the interference relation in the SMT solving of multi-threaded program verification under various memory models. We show that the backend SMT solver can benefit a lot from the domain knowledge of concurrent programs. We implemented our approach in a prototype tool called Zpre. We compared it with the state-of-the-art Z3 tool on credible benchmarks from the *ConcurrencySafety* category of *SV-COMP* 2019. Experimental results show promising improvements attributed to our approach.

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

Keywords: Concurrent programs, Program verification, Satisfiability modulo theory, Partial order, Weak memory models

*Corresponding author

PPoPP '22, April 2–6, 2022, Seoul, Republic of Korea
© 2022 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ACM ISBN 978-1-4503-9204-4/22/04…$15.00
https://doi.org/10.1145/3503221.3508424

1 Introduction

Concurrent program verification is challenging in practice [7, 44]. Thread interference is the main hurdle for verifying such systems. Consider such a situation that a variable is shared between two threads; it is hard to say which access shall happen before the other. Within the weak memory models, the ordering of instructions in each thread can further be violated. To prove the correctness of a concurrent program, one needs to consider all possible interferences between concurrent threads. The vast number of thread interferences makes the reasoning of concurrent programs highly intricate.

At the same time, *Satisfiability modulo theories* (SMT) [8, 19, 20] plays a vital role in program verification. An SMT-based program verifier consists of two components: a frontend that encodes the verification condition as an SMT formula; and a backend that solves the verification condition formula. There are many well-known studies on frontends. One of the most successful techniques is the use of partial orders [6, 7, 33, 35, 36] to model the interference relation of concurrent programs. Each memory access is associated with a unique timestamp, and the interference relations are represented as ordering constraints over these timestamps. Sinha and Chao [45] introduced interference abstraction to *over-* and *under-approximate* thread interference, which can analyze concurrent programs more efficiently. Cordeiro [16] proposed a method to check deadlock and data races of concurrent programs by explicitly exploring interferences and producing one symbolic execution per interference. From the angle of an SMT solver, all the above techniques were focused on the encoding of concurrent program verification and did less on the backend SMT solver.

A powerful SMT solver is certainly a crucial factor for concurrent program verification. However, most of the SMT solvers are designed for the general purpose of constraint solving. Domain-specific knowledge is neglected in these solvers. As a result, the SMT solver may explore redundant search space, which will be pruned if the domain knowledge is applied [9, 14, 26, 38]. In concurrent program verification, interference relations convey essential information about the interleavings of threads, and applying this knowledge in SMT solving may be quite useful for pruning the search space. When modeling a multi-threaded program, each interference relation is represented using a Boolean variable, called an *interference variable*.

Our basic idea is to develop a strategy to guide the SMT solving of concurrent program verification. Note that the DPLL(T) [24] procedure is essentially a depth-first search process; it needs to choose a variable and make an assignment at each search step. We intend to construct a partial order, called the *decision order*, which represents priorities of unassigned variables being selected in the search procedure of DPLL(T). To some extent, the decision order determines the search direction of SMT solving. We propose several heuristics for constructing this decision order. Firstly, we recognize the importance of interference variables and assign them higher priorities in the decision order. Secondly, we distinguish several types of interference variables and assign them with different priorities. Finally, we propose a lightweight technique to enforce the decision order in the SMT solver to improve the efficiency of SMT solving.

Another issue for concurrent program verification is the relaxed memory access in the weak memory model (WMM). *Sequential consistency* (SC) [36] forces memory access in each thread to follow the order of instructions, but WMM (e.g., TSO [43], PSO [47]) allows certain memory operation ordering to be relaxed, which brings intra-thread uncertainty and makes concurrent program verification even harder. Modeling concurrent programs in WMM is more complex than in SC. However, changing the memory model has less impact on the number of interference variables. This paper also illustrates that for concurrent program verification, our approach applies to WMMs and has a higher efficiency than in SC.

We implemented our approach on top of CBMC [33] and Z3 [19]. We performed extensive experiments on credible benchmarks collected from the *ConcurrencySafety* category of *SV-COMP* 2019 [1]. Counting on both-solved SMT instances, our proposed method achieved an average of 1.49x, 1.87x, and 1.89x speedups over Z3 in SC, TSO, and PSO, respectively.

The contributions of this paper are summarized as follows:

- We presented a novel approach for utilizing interference relation to guide SMT solving of multi-threaded program verification under various memory models.
- We devised efficient heuristics for constructing the decision order of multi-threaded programs.
- We implemented our approach on top of CBMC and Z3 and conducted extensive experiments to confirm its effectiveness and evaluate its efficiency. Experimental results show the promising performance of our approach.

The rest of this paper is organized as follows. Section 2 introduces necessary preliminaries. Section 3 uses a simple

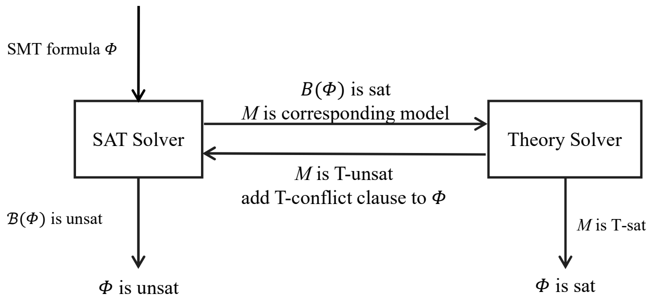


Figure 1. Flow of DPLL(T).

Algorithm 1: DPLL(T)

```
Input: An SMT formula $\Psi$
Output: sat or unsat
while true do
    while ! propagate_and_check() do
        if !has_decision() then return unsat ;
        else resolve_conflict() ;
    if !decide() then return sat ;
```

example to illustrate the importance of interference relation for concurrent program verification. Section 4 details our interference relation-guided SMT solving. Experimental results and analysis are presented in Section 5, followed by related works in Section 6 and conclusion in Section 7.

2 Preliminaries

2.1 SMT and DPLL(T)

Satisfiability modulo theories (SMT) is the satisfiability problem with a combination of first-order background theories. SMT is widely applied in software verification, hardware model checking, theorem proving, etc.

Figure 1 shows a high-level overview of the standard framework for SMT solving, called DPLL(T) [18, 24]. Given an SMT formula Φ, each atom of it is replaced by a new Boolean variable (called *Boolean abstraction*). The resulting formula, denoted as $\mathcal{B}(\Phi)$, is a Boolean formula. DPLL(T) employs a SAT solver to find a satisfiable model M for $\mathcal{B}(\Phi)$. Note that $\mathcal{B}(\Phi)$ is an *over-approximation* of Φ. If $\mathcal{B}(\Phi)$ is unsatisfiable, the original formula Φ must also be unsatisfiable. However, the opposite may not hold. If $\mathcal{B}(\Phi)$ is satisfiable, DPLL(T) calls theory solvers to check the T-satisfiability of M. If M is T-sat, Φ is also satisfiable. Otherwise, the theory solver adds conflict clauses to Φ and passes it to SAT solver. Then DPLL(T) performs the next iteration.

Algorithm 1 depicts the pseudocode of DPLL(T). The method *propagate_and_check()* employs unit and theory propagation to assign values to as many variables as possible. It returns *true* if the current model is T-sat, and *false* if the current model has theory conflicts. The method *resolve_conflict()*

learns conflict clauses to enhance constraints in formula Φ. The method *decide()* selects the next unassigned Boolean variable. We conclude that the current model satisfies Φ if there are no unassigned variables.

Example 2.1. For SMT formula $\phi \equiv (x = 3 \vee y \leq 4) \wedge x \geq 5$, where x and y are two variables. The SMT solver first abstracts ϕ as $\mathcal{B}(\phi) \equiv (b_1 \vee b_2) \wedge b_3$ where Boolean variables b_1, b_2, and b_3 denote $x = 3$, $y \leq 4$, and $x \geq 5$, respectively. If $\mathcal{B}(\phi)$ is *true*, then the clause $b_1 \vee b_2$ must be *true*. Firstly, suppose the SAT solver assigns b_1 to *true*, a satisfying model M, i.e., $(b_1 = 1, b_2 = 0, b_3 = 1)$ of $\mathcal{B}(\phi)$ is returned eventually. However, this model is spurious since $x = 3 \wedge y > 4 \wedge x \geq 5$ is unsat under integer theory. After resolving this conflict, the SAT solver assigns b_2 to *true* and returns another satisfying model M', i.e., $(b_1 = 0, b_2 = 1, b_3 = 1)$. Since $x \neq 3 \wedge y > 4 \wedge x \geq 5$ is also satisfiable, then ϕ is satisfiable.

Note that if the SAT solver assigns b_2 to *true* at first, then M' is found in the first iteration. It is evident that a more reasonable decision order of these unassigned Boolean variables may significantly improve the efficiency of SMT solving by reducing redundant search space. Heuristics such as VSIDS[38] and CFG-based method[14] are elaborated for selecting the next unassigned variable and deciding its value. In this paper, we utilize the knowledge of the interference relation and derive a reasonable decision order to guide the SMT solving of concurrent program verification.

2.2 Multi-Threaded Program and Memory Model

A concurrent program comprises multiple threads running in parallel. There are two kinds of variables in a concurrent program, i.e., *local variables* that are accessible by a specific thread only, and *shared variables* that are accessible by all threads. An *event* is either a read or a write access to a variable. An event is called *global* if it accesses a shared variable. In the following, we consider only global events. Denote $var(e)$, $guard(e)$ and $clk(e)$ as the accessed variable, guard condition and timestamp of event e, respectively. Two events e_1 and e_2 access the same variable if $var(e_1) = var(e_2)$. Considering branches, assumption statements, etc, $guard(e)$ encodes the guard condition of the occurrence of event e. As in [45], we use an integer-valued timestamp to specify the order of the event: an event e_1 happens before another event e_2 iff $clk(e_1) < clk(e_2)$.

A memory model determines the execution order of memory access events. *Sequential Consistency* (SC) [36] is the most simple and commonly assumed memory model. In SC, the execution order of events from the same thread must follow the order of instructions, i.e., reordering between neighboring access events is not allowed. On the contrary, a *Weak Memory Model* (WMM) allows certain memory access orders to be inconsistent with the instruction order. More specifically, this paper mainly considers two WMMs, i.e., *Total Store Order* (TSO) [43], widely employed in x86 and SPARC architectures, where permutation to a write event followed by a read event is allowed if two events refer to different memory addresses; and *Partial Store Order* (PSO) [47], which further relaxes the order between two write events that manipulate different memory addresses.

Initial x:=0 y:=0	
t_1	t_2
x:=y+1	y:=x+1
m:=y	n:=x
Assert(!(m==0&&n==0))	

Initial x_1:=0 y_1:=0	
t_1	t_2
x_2:=y_2+1	y_4:=x_3+1
m_1:=y_3	n_1:=x_4
Assert(!(m_2==0&&n_2==0))	

Figure 2. A three-threaded program and its SSA form.

3 Interference Relation for Concurrent Program Verification

In this section, we first use the simple example in Figure 2 (left) to introduce the symbolic encoding of multi-threaded programs. Then we show the importance of interference relation in verifying multi-threaded programs.

3.1 Symbolic Encoding

As usual, we adopt the *static single assignment* (SSA) [45] style during encoding. The SSA form of the example program is shown in Figure 2 (right). In this paper, we also use SSA to represent the corresponding access event.

With SSA form, the *verification condition* for the correctness of the program can be encoded as an SMT formula, i.e.,

$$\Phi = \Phi_{prog} \wedge \Phi_{err}, \tag{1}$$

where Φ_{prog} encodes the program and Φ_{err} encodes the error condition. The program is *correct* with respect to the given property iff the verification condition Φ is *unsatisfiable.*

Following the modeling approach in [6], a multi-threaded program can be encoded using SSA statements and partial orders, i.e.,

$$\Phi_{prog} = \Phi_{ssa} \wedge \Phi_{po} \wedge \Phi_{rf} \wedge \Phi_{rf\_some} \wedge \Phi_{ws} \wedge \Phi_{fr}, \tag{2}$$

where Φ_{ssa}, Φ_{po}, Φ_{rf}, Φ_{rf_some}, Φ_{ws} and Φ_{fr} will be explained in the following (see Figure 3 for detailed encodings of the example program).

SSA Statements. Φ_{ssa} encodes the program statements in SSA form.

Program Order. Φ_{po} encodes the *program order* for each thread. If two events e_1 and e_2 have the program order, then e_1 and e_2 must belong to the same thread and e_1 happens before e_2, i.e, $clk(e_1) < clk(e_2)$.

WMM relaxes some order restrictions of events from the same thread, so Φ_{po} may be different under WMM. In particular, TSO relaxes the program order between a write event w to a read event r if $var(w) \neq var(r)$. The program order

$$\Phi_{ssa}:\ x_1 = 0 \wedge y_1 = 0 \wedge x_2 = y_1 + 1 \wedge m_1 = y_3 \wedge y_4 = x_3 + 1 \wedge n_1 = x_4$$

$$\begin{array}{lrl} \Phi_{po}: & t_1 : & clk(y_2) < clk(x_2) \quad clk(x_2) < clk(y_3) \quad clk(y_3) < clk(m_1) \\ & t_2 : & clk(x_3) < clk(y_4) \quad clk(y_4) < clk(x_4) \quad clk(x_4) < clk(n_1) \\ & main : & clk(x_1) < clk(y_1) \quad clk(y_1) < clk(m_2) \quad clk(m_2) < clk(n_2) \end{array}$$

$$\begin{array}{lll} \Phi_{rf}: & rf^x_{1,3} \to x_3 = x_1 \wedge clk(x_1) < clk(x_3) & rf^x_{2,3} \to x_3 = x_2 \wedge clk(x_2) < clk(x_3) \\ & rf^x_{1,4} \to x_4 = x_1 \wedge clk(x_1) < clk(x_4) & rf^x_{2,4} \to x_4 = x_2 \wedge clk(x_2) < clk(x_4) \\ & rf^y_{1,2} \to y_2 = y_1 \wedge clk(y_1) < clk(y_2) & rf^y_{4,2} \to y_2 = y_4 \wedge clk(y_4) < clk(y_2) \\ & rf^y_{1,3} \to y_3 = y_1 \wedge clk(y_1) < clk(y_3) & rf^y_{4,3} \to y_3 = y_4 \wedge clk(y_4) < clk(y_3) \end{array}$$

$$\Phi_{rf\_some}:\ (rf^x_{1,3} \vee rf^x_{2,3}) \wedge (rf^x_{1,4} \vee rf^x_{2,4}) \wedge (rf^y_{1,2} \vee rf^y_{4,2}) \wedge (rf^y_{1,3} \vee rf^y_{4,3})$$

$$\begin{array}{lll} \Phi_{ws}: & ws^x_{1,2} \to clk(x_1) < clk(x_2) & \neg ws^x_{1,2} \to clk(x_2) < clk(x_1) \\ & ws^y_{1,4} \to clk(y_1) < clk(y_4) & \neg ws^y_{1,4} \to clk(y_4) < clk(y_1) \end{array}$$

$$\begin{array}{lll} \Phi_{fr}: & rf^x_{1,3} \wedge ws^x_{1,2} \to clk(x_3) < clk(x_2) & rf^x_{2,3} \wedge (\neg ws^x_{1,2}) \to clk(x_3) < clk(x_1) \\ & rf^x_{1,4} \wedge ws^x_{1,2} \to clk(x_4) < clk(x_2) & rf^x_{2,4} \wedge (\neg ws^x_{1,2}) \to clk(x_4) < clk(x_1) \\ & rf^y_{1,2} \wedge ws^y_{1,4} \to clk(y_2) < clk(y_4) & rf^y_{4,2} \wedge (\neg ws^y_{1,4}) \to clk(y_2) < clk(y_1) \\ & rf^y_{1,3} \wedge ws^y_{1,4} \to clk(y_3) < clk(y_4) & rf^y_{4,3} \wedge (\neg ws^y_{1,4}) \to clk(y_3) < clk(y_1) \end{array}$$

$$\Phi_{err}:\ m_2 == 0 \wedge n_2 == 0$$

Figure 3. Symbolic encoding of the example program

of t_1 under TSO is:

$$\begin{array}{ll} t_1 : clk(y_2) < clk(x_2) & clk(y_2) < clk(y_3) \\ \quad clk(y_3) < clk(m_1) & clk(x_2) < clk(m_1) \end{array}$$

PSO further relaxes the program order between a write event w to a read/write event e if $var(w) \neq var(e)$, so program order of t_1 under PSO is:

$$t_1 : clk(y_2) < clk(x_2) \quad clk(y_2) < clk(y_3) \quad clk(y_3) < clk(m_1)$$

Program order under TSO (PSO) of t_2 and *main* threads are similar to t_1, we don't show them for brevity. Note that program order between write event y_1 and read event m_2 in *main* thread is not relaxed since this order restriction is preserved by *thread_create* and *thread_join* functions.

Read-From Order. Φ_{rf} encodes the *read-from* relation. For a shared variable x, a *read-from* relation links a write event x_i to a read event x_j if $var(x_i) = var(x_j)$; and: (1) x_j loads the value stored by x_i; (2) x_i happens before x_j, called a *read-from* order, represented as $clk(x_i) < clk(x_j)$; (3) the guard condition of x_i must be *true*. A Boolean variable is introduced to represent a read-from relation:

$$rf^x_{i,j} \rightarrow (x_j = x_i \wedge clk(x_i) < clk(x_j) \wedge guard(x_i))$$

If the read event x_j happens, it must get its value from one write event that accesses the same variable. We use Φ_{rf_some} to encode such constraints:

$$guard(x_j) \rightarrow \bigvee_{var(x_i)=var(x_j)} rf^x_{i,j}$$

Write-Serialization Order. Φ_{ws} encodes a total order over the write events to the same memory address, called the *write-serialization* order. If x_i and x_k have this order, then $var(x_i) = var(x_k)$, and $clk(x_i) < clk(x_k)$. A Boolean variable is introduced to represent a write-serialization relation:

$$ws^x_{i,k} \rightarrow clk(x_i) < clk(x_k) \quad \neg ws^x_{i,k} \rightarrow clk(x_k) < clk(x_i)$$

From-Read Order. Φ_{fr} encodes the *from-read* order. If a read event x_j gets its value from a write event x_i, then for any other write x_k to the same address, it cannot happen between x_i and x_j, because otherwise x_k overwrites x_i, and x_j should get its value from x_k but not x_i. In other words:

$$rf^x_{i,j} \wedge ws^x_{i,k} \rightarrow clk(x_j) < clk(x_k)$$

Finally, the error condition Φ_{err} is the negation of the safety property. Figure 3 details the symbolic encoding of the example program. The verification condition of the example program is the conjunction of all above constraints.

3.2 Interference Relation

Two accesses *interfere* [45] if they access the same address and at least one of them is a write access. According to the access types (read or write), there are three kinds of interference relations, i.e., *read-from relation*, *write-serialization relation* and *from-read relation.*

Rethinking of the encoding formula (Equation (2)) of the multi-threaded program, it can be divided into three parts, i.e.,

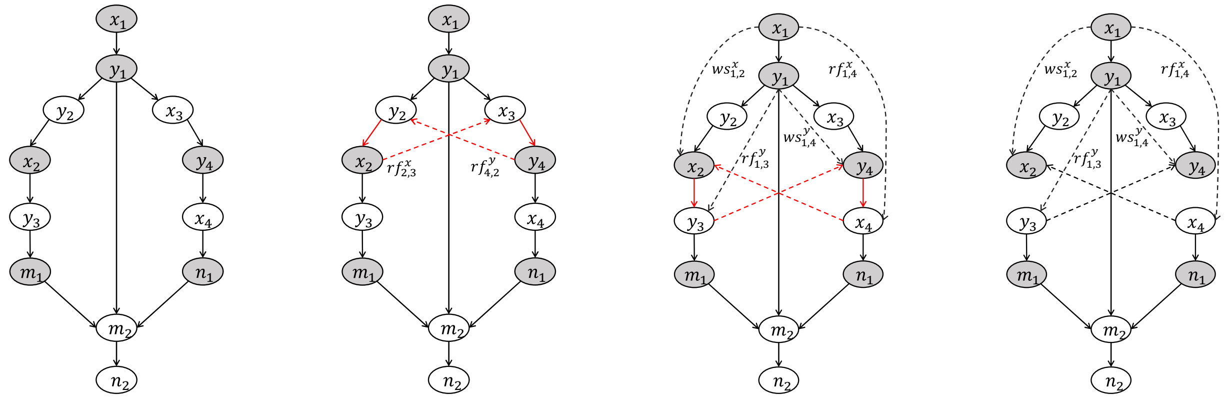
$$\Phi_{prog} = \Phi_{ssa} \wedge \Phi_{po} \wedge \Phi_{itf}$$

where $\Phi_{itf} = \Phi_{rf} \wedge \Phi_{rf\_some} \wedge \Phi_{ws} \wedge \Phi_{fr}$. The first conjunct Φ_{ssa} represents the data and control flow per thread; the second, Φ_{po}, depicts the program order per thread; and the third, Φ_{itf}, captures the interference between threads.

Regard Φ_{prog} as a first-order formula over a set of variables. After Boolean abstraction (see Section 2), there are four kinds of Boolean variables occurring in $\mathcal{B}(\Phi_{prog})$:

- *SSA variables*, i.e., the variables for representing the program statements, assignment statements, and guard conditions in the program, denoted as V_{ssa}.
- *ordering variables*, i.e., the variables for representing the ordering constraints between memory access events, denoted as V_{ord}.
- *read-from variables*, i.e., the variables for representing the read-from relations, denoted as V_{rf};
- *write-serialization variables*, i.e., the variables for representing write-serialization relations, denoted as V_{ws}.

For example, Boolean variables which represents the program statement $x_2 = y_1 + 1$ or assignment statement $x_3 = x_1$ (in Figure 3) are included in V_{ssa}; and a Boolean variable that



(a) program order only **(b)** an invalid execution **(c)** another invalid execution **(d)** the execution in **(c)** under TSO

Figure 4. Event order graph of the example program

denotes $clk(x_1) < clk(x_3)$ is included in V_{ord}. The read-from and write-serialization variables are also called *interference variables*. Let $V_{itf} = V_{rf} \cup V_{ws}$ be the set of interference variables, and $V = V_{ssa} \cup V_{ord} \cup V_{itf}$ be the set of all variables.

An *assignment* of V is a mapping from variables in V to values. The assignment is said *complete* if every variable in V is assigned a value, and *partial* if only part of variables in V are assigned. DPLL(T) is essentially a search procedure that starts with an empty assignment and tries to find a complete assignment that satisfies the verification condition.

3.3 Interference Relation is Important

A *concrete concurrent execution* of a multi-threaded program is a complete assignment of V. A *symbolic concurrent execution* is a *partial assignment* that assigns values to part of variables in V_{itf}. A concrete concurrent execution determines not only the order among memory accesses, but also their concrete values; whereas a symbolic concurrent execution specifies only the order among memory accesses. The above definitions are equivalent to those in [6, 44].

Symbolic concurrent execution can be represented as an *event order graph* (EOG), where each node represents an event, and each edge represents an order between two events. For readability purposes, we use the grey and white nodes in the graph to represent the write and read events, the solid and dashed edges to represent the program and interference order, respectively. Program order regulates the primary ordering constraints among the events in the program, which are determined at the beginning of SMT solving. For example, Figure 4a shows the EOG of the example program with program order under SC. Partial valuation to interference variables determines interference order, which gives more ordering constraints among events on the EOG and establishes a symbolic concurrent execution. It has been proven [44] that a symbolic concurrent execution is valid if there exists a *total order* among the events of the execution, i.e., the corresponding EOG contains no cycle. If the current partial assignment to V_{itf} produces a cycle on the EOG, the DPLL(T) needs to backtrack and try other assignments to V_{itf}.

Figure 4b shows an invalid symbolic concurrent execution because there is no total order among the events, i.e., a cycle ($y_2 \to x_2 \to x_3 \to y_4 \to y_2$) occurs. The cycle is caused by two interference variables $rf^x_{2,3}$ and $rf^y_{4,2}$, and the DPLL(T) needs to resolve this conflict. Figure 4c shows another invalid symbolic concurrent execution. According to the read-from order $clk(y_1) < clk(y_3)$ (by $rf^y_{1,3}$) and the write-serialization order $clk(y_1) < clk(y_4)$ (by $ws^y_{1,4}$), a from-read order $clk(y_3) < clk(y_4)$ is derived. Similarly, another from-read order $clk(x_4) < clk(x_2)$ is also derived. Apparently, the current assignment to V_{itf} is invalid since the EOG contains a cycle, i.e., $x_2 \to y_3 \to y_4 \to x_4 \to x_2$. The cycle is caused by interference variables $rf^x_{1,4}$, $rf^y_{1,3}$, $ws^x_{1,2}$ and $ws^y_{1,4}$, DPLL(T) also needs to backtrack and resolve this conflict.

Consider Figures 4b and 4c; the invalidity can be detected earlier only if the interference variables are assigned. Actually, for any invalid execution, the corresponding EOG must contain (at least) a cycle, and there must be some interference variables from which this cycle can be derived. Besides, most of the valid symbolic concurrent executions that violate safety properties are also caused by thread interference (see Figure 4d). Therefore, prioritizing interference variables in DPLL(T) can help the SMT solver quickly decide whether the current assignment leads to an invalid execution or not.

Moreover, consider the example program; if $rf^x_{1,4}$ is assigned *true*, both the ordering variable representing $clk(x_1) < clk(x_4)$ and the SSA variable representing the assignment statement $x_4 = x_1$ will further be propagated to *true*. We say interference variables *dominate* some ordering constraints variables in V_{ord} and some SSA variables in V_{ssa}. To some extent, the valuation of interference variables drives the search direction of the DPLL(T). Therefore, applying this knowledge and prioritizing interference variables can help the

DPLL(T) propagate more related Boolean variables instead of searching redundant search space.

In summary, interference variables characterize the interleaving semantics of the concurrent programs, which may be quite useful if they are specially treated in SMT solving.

3.4 Interference Relation is Neglected

It is widely accepted that reasonable heuristics for choosing the next decision variable might significantly improve the performance of SMT solving [24]. However, for concurrent program verification, the default DPLL(T) is unaware that interference variables are important; and it treats all variables in V equally.

Consider the *from-read* order constraint $rf^x_{2,3} \wedge (\neg ws^x_{1,2}) \rightarrow (clk(x_3) < clk(x_1))$ in Figure 3. During Boolean abstraction, it is first transformed into a clause, i.e, $(\neg rf^x_{2,3}) \vee ws^x_{1,2} \vee b_1$, where b_1 is a Boolean variable representing $clk(x_3) < clk(x_1)$. Although b_1 is dominated by $rf^x_{2,3}$ and $ws^x_{1,2}$, DPLL(T) may first assign b_1 to $true$, then continues to perform propagations and decisions until the conflict that $clk(x_3) < clk(x_1)$ betrays program order is found.

The default selection strategy (e.g., VSIDS [38, 42], random, unit clause[18]) for the next unassigned variable may cause the solver to explore redundant search space. Moreover, most program variables and all clock variables are non-Boolean variables. For example, a 32-bit integer-types variable can be represented by a bit-vector with a bit width of 32; if such a variable is assigned, the DPLL(T) has to keep making numerous decisions and propagations on each bit of the vector to derive a value of this variable. Additionally, handling non-Boolean variables often involves theory solvers, which are complicated and time-consuming.

Besides, if a symbolic concurrent execution is invalid, all its corresponding concrete concurrent executions are invalid. Note that a valuation to V_{itf} determines a symbolic concurrent execution. Instead of treating all variables in V equally, prioritizing interference variables can drive the search direction of the DPLL(T), helping the SMT solver backtrack earlier when the current assignment is already invalid. In this way, the redundant search space is pruned, and the SMT solving efficiency is improved. To this end, we propose an interference relation-guided SMT solving approach.

4 Interference Relation-Guided SMT Solving

In this section, we detail how to produce the decision order and how to utilize this decision order in SMT solving.

4.1 Decision Order Generation

An SMT-based program verifier is composed of two parts: a frontend that parses the program and encodes the verification condition as an SMT formula; and a backend (an SMT solver) that solves the verification condition formula.

During the frontend encoding verification condition formula, the interference variables are denoted in a special fashion. Each RF variable is named as $rf\_r_t\_r_i\_w_t\_w_i$ where r_t, w_t represent the IDs of threads to which the read and write events belong, respectively; r_i, w_i represent the intra-thread locations on which the read and write events occur, respectively. We also name WS variables with a similar recipe.

DPLL(T) is essentially a depth-first search process; at each search step, it needs to choose a variable and make an assignment. We intend to define a partial order $\preceq$ on V to prioritize variables selected for the assignment. Once the SMT formula is passed to the backend, the SMT solver recognizes the interference variables by their names and constructs a decision order on V.

Our first insight is that the interference variables are more important than other variables. We thus have:

HEURISTIC 1. *Interference variables are prior to other variables, i.e.,* $\forall v_1 \in V_{itf}, \forall v_2 \in V \setminus V_{itf}, v_1 \preceq v_2$.

By applying this native HEURISTIC 1, interference variables are prior to other variables in V when DPLL(T) selects the next unassigned variable. Moreover, we further utilize the knowledge of the concurrent program to arrange this decision order in detail.

There are two kinds of interference variables, i.e., the read-from (RF) variables, and the write-serialization (WS) variables. We observe that: (1) RF variables can dominate some valuations of SSA variables, while WS variables can not; (2) many WS orders are implied by program order; boolean variables for representing these WS orders can thus be directly propagated during DPLL(T). Therefore, we suggest giving the RF variables higher priorities than the WS variables, i.e.,

$$\forall v_1 \in V_{rf}, \forall v_2 \in V_{ws}, v_1 \preceq v_2$$

Additionally, A RF variable links a read event and a write event that access the same memory address. If these two events belong to the same thread, we call the RF variable *internal*; otherwise, it is *external.* Let V_{rfe} and V_{rfi} be the sets of external and internal RF variables, respectively. Once a read-from order $clk(w) < clk(r)$ is assigned, all events that happened before w should also happen before r. Meanwhile, all events that happen after r should also happen after w. If w and r are from the same thread, this read-from order is implied by the program order and has nothing to do with the interleaving semantics of concurrent programs. Instead, if w and r are from different threads, this read-from order captures the interference relation and reduces uncertainties caused by thread interleaving. The reason why concurrent programs are error-prone is thread interference. Therefore, we give higher priority to external RF variables, i.e.,

$$\forall v_1 \in V_{rfe}, \forall v_2 \in V_{rfi}, v_1 \preceq v_2$$

Moreover, thread interference causes that the value obtained by a read event may come from many possible write events. The more writes to a shared variable, the greater impact of the thread interferences on this variable. Let $\#write(v)$ be the number of write events that the read event r may read from. Intuitively, let r_1 and r_2 be the read events corresponding to the RF variables v_1 and v_2, respectively; if $\#write(v_1) > \#write(v_2)$, v_1 is assigned higher priority when DPLL(T) selects the next unassigned variable.

Algorithm *prior_to*(v_1, v_2) explains how we construct the decision order in detail. This algorithm takes two interference variables v_1 and v_2 as parameters and returns *true* if v_1 is prior to v_2, and *false* otherwise. It mainly handles the following 4 cases:

- RF variables are prior to WS variables. If $v_1 \in V_{rf}$ and $v_2 \in V_{ws}$, then this algorithm returns *true*.
- External RF variables are prior to internal RF variables. If $v_1 \in V_{rfe}$ and $v_2 \in V_{rfi}$, then *true* is returned.
- If v_1, v_2 are both RF variables, let n_1 be *#write*(v_1) and n_2 be *#write*(v_2); if $n_1 > n_2$, then v_1 is prior to v_2.
- Otherwise, the algorithm returns *false*.

Finally, it terminates with a new decision order generated.

4.2 Enhanced DPLL(T)

DPLL(T) selects the next unassigned variable in *decide()* procedure (see Algorithm 1). Therefore, we guided the SMT solving by enhancing the *decide()* procedure using the generated decision order. Figure 5 shows a high-level overview of our enhanced-DPLL(T). Assuming no conflicts are found after *propagation_and_check()*, if there are any unassigned interference variables in V_{itf}, the enhanced-DPLL(T) selects the *first* variable following the decision order $\preceq$ and assigns it with a random Boolean value. Otherwise, it selects and assigns the next variable using the default heuristics [38, 42] implemented in the SMT solver.

Our approach is a series of domain-specific heuristics for the recognition and priority decision of interference relations; these heuristics are devised for SMT solving of multi-threaded program verification. Meanwhile, the default heuristics (e.g., conflict-driven clause learning, VSIDS, unit-clause propagation) are also enabled in our approach. In the enhanced-DPLL(T), we first try to use our heuristics to select the next unassigned interference variable. If all our heuristics cannot apply, i.e., values of all interference variables are decided or propagated, we follow the default heuristics to select the next variable.

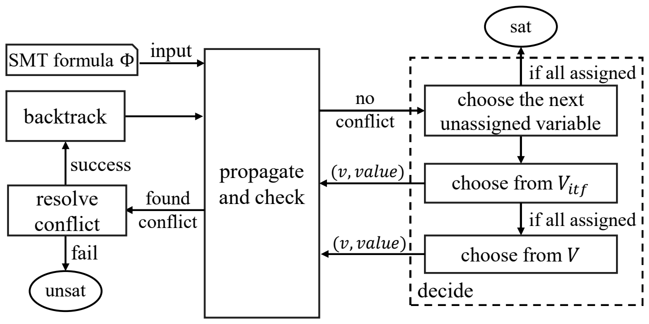


Figure 5. Enhanced DPLL(T)

5 Experimental Evaluation

This section details the experimental results and analysis of our interference relation-guided SMT solving tactic.

Implementation. We implemented our approach[1] on top of CBMC and Z3, where CBMC is a powerful and flexible bounded model checker for C\C++ programs, and Z3 is a well-known and widely-adopted SMT solver. In this paper, we modify CBMC to 1) extract information about interference variables; and 2) encode this information into the SMT formula. We use CBMC to generate SMT formulas under different memory models. All the generated SMT formulas are in the SMT-LIB-v2.6 format. The interference relation-guided SMT solving strategy is implemented in Z3. In the following, we call our implementation ZPRE and make Z3 with its default solving strategies (e.g., VSIDS, conflict-clause learning) the *baseline*.

Benchmarks. We collected benchmarks from the *ConcurrencySafety* category of *SV-COMP* 2019. These benchmarks have been widely accepted since they are comprehensive, credible, and have already been preprocessed for verification. Many studies perform their experiments on these benchmarks to demonstrate the effectiveness of their method.

The *ConcurrencySafety* category contains 12 subcategories and 1084 C programs, namely *ldv-linux* (9), *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 followed to each subcategory represents the number of programs it contains. Programs in *ldv-linux* and *complex* contains complicated data structures, CBMC fails to generate correct SMT files of these programs since Z3 reports parse errors and throws exceptions, so we exclude all 14 programs of these two subcategories, and we get 1070 programs in total.

Experimental Setup. Bounded model checking (BMC)[15] is efficient in finding bugs. A program can be converted to a loop-free one by replacing every loop with a nested (to a specific loop unrolling bound) if-statement. Let k^* be the minimal unrolling bound that violates the given property. If the current unrolling bound $k < k^*$, then the corresponding SMT formula is unsatisfiable, and the original program is correct under unrolling bound k. If $k \geq k^*$, then the SMT formula is satisfiable, and the original program violates the given property. In this paper, we set the loop unrolling bound from 1 to 6 and use CBMC to generate different SMT formulas for each multi-threaded program. We set a time limit of

[1]Artifact is available at: https://thufv.github.io/research/ppopp22-artifact

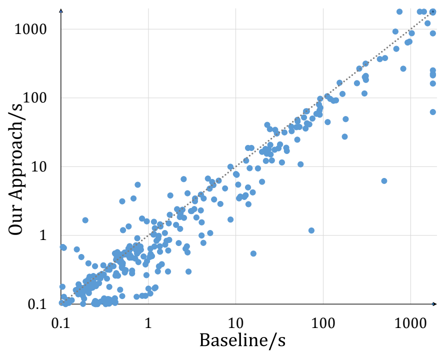


Figure 6. ZPRE vs. Z3 in SC

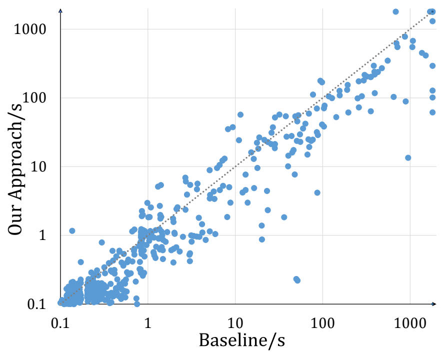


Figure 7. ZPRE vs. Z3 in TSO

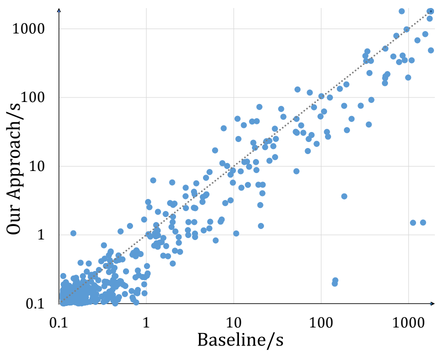


Figure 8. ZPRE vs. Z3 in PSO

5 minutes for generating the SMT formulas. Through our experience, if the generation cannot be finished in 5 minutes, the size of the generated SMT file is usually beyond 1GB, which is too hard to solve for the current SMT solvers. Moreover, if the original program contains no loops, the generated SMT files with different unrolling bounds are the same. After eliminating duplications, we finally obtained – 1673 SMT files in SC, 1651 SMT files in TSO, and 1643 SMT files in PSO. The file size ranges from several KB to hundreds of MB. In the following, an SMT instance is also called a verification task.

All the experiments are conducted on a computer with Intel(R) Core(TM) i7-8700 CPU @3.20 GHz and 32GB memory, and the operating system is ArchLinux-5.11.10. The time limit for solving each SMT instance is set to 1800 seconds.

5.1 Experimental Results

Overall Results. Figures 6, 7, and 8 display the SMT solving time (CPU time) of Z3 and ZPRE on all the verification tasks under SC, TSO, and PSO memory models, respectively. Each point in the panel corresponds to a verification task, with the X and Y coordinates representing the SMT solving time of the Z3 and ZPRE, respectively. Note that both x- and y-axis take logarithmic coordinates, and each point below/above the diagonal line represents a superior/inferior case of our approach against the Z3.

Considering the both-solved cases, Table 1 shows the overall results of Z3 and ZPRE under different memory models. The column *Sat* (*Unsat*) reports the accumulated CPU time and speedup of satisfiable (unsatisfiable) cases, and the column *All* reports these statistics on all cases. The unit of all time data is second; the speedup greater than 1.0x means that our tactic is faster than Z3 with respect to the selected memory model and the group of SMT instances.

Results under SC. In SC memory model, our tactic is superior to Z3 in most cases since most of the points in Figure 6 are below the diagonal. Considering 1589 both-solved cases, Z3 spends 14344.3s whereas ZPRE costs 9596.9s – our tactic is 1.49x times faster than the Z3 to resolve the same number of SMT instances under SC.

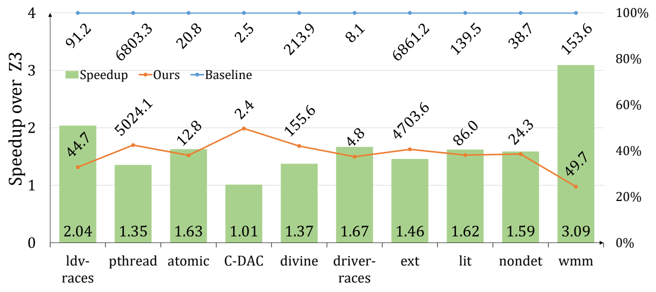


Figure 9. Time of subcategories in SC: Z3 vs. ZPRE

Table 1. Overall results: Z3 vs. ZPRE

MM	(Z3/ZPRE, #Speedup)		
	Sat	Unsat	All
SC	(8420/6312, 1.34x)	(5924/3194, 1.85x)	(14344/9507, 1.49x)
TSO	(7460/3224, 1.76x)	(12933/6698, 1.93x)	(20393/10922, 1.87x)
PSO	(15974/8244, 1.93x)	(7502/4348, 1.72x)	(23477/12393, 1.89x)

There are 7 points on the right boundary of Figure 6, indicating that there are 7 SMT instances that cannot be solved by Z3 (within the time limit of 1800 seconds) but can be solved by ZPRE correctly. Specifically, the exact solving time of ZPRE on these 7 SMT instances are 62.1s, 161.9s, 212,6s, 222.9s, 251,7s, 869.8s, and 1741.1s, respectively. If we cancel the time limit, the reported solving time for Z3 on these 7 SMT instances is 4015.7s, 3718.2s, 7713.1s, 2929.3s, 3179.5s, 7325.1s, and 14154.9s, respectively. Symmetrically, there are 3 points on the uppermost boundary of Figure 6, indicating 3 cases that ZPRE time out whereas Z3 does not. The accurate solving time of Z3 on these 3 cases is 1273.2s, 1416.7s, and 1746.1s, respectively. If we cancel the time limit, the used solving time for ZPRE on these 3 cases is 2583.7s, 2997.3s, 3435.6s, respectively.

Figure 9 shows the SMT solving time of each subcategory in SC, where the blue line represents Z3 and the orange line

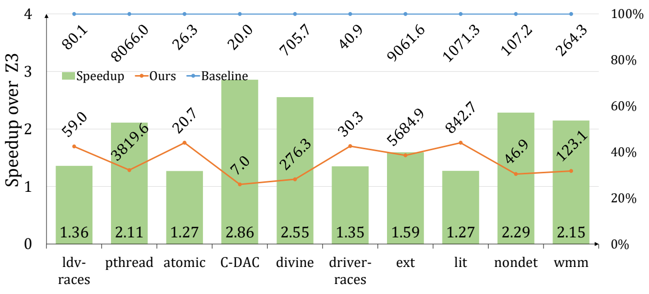


Figure 10. Time of subcategories in TSO: Z3 vs. ZPRE

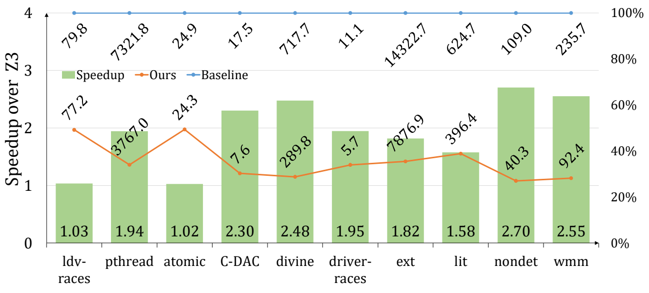


Figure 11. Time of subcategories in PSO: Z3 vs. ZPRE

represents ZPRE. We also draw histograms at the bottom of Figure 9 to indicate the speedup of our approach on each subcategory. In Figure 9, the speedup varies from 1.01x to 3.09x times. On some non-trivial and representative subcategories, e.g., *pthread*, *divine*, *ext*, *lit*, and *wmm*, our method is superior to Z3 with a speedup of 1.35x, 1.37x, 1.46x, 1.62x, and 3.09x, respectively.

Results under TSO and PSO. In TSO/PSO memory model, our tactic is still superior to Z3, witnessed by most points in Figure 7 and Figure 8 being below the diagonal.

Considering the 1592 both-solved cases under TSO, Z3 takes 20393.4s, whereas ZPRE only costs 10922.7s – our tactic is 1.87x times faster than the Z3 to resolve the same number of SMT files. On the right boundary of the Figure 7, there are 5 points timed out in Z3, but ZPRE can solve them. The accurate time of ZPRE is 293.2s, 101.4s, 1306.8s, 61.5s, and 127.8s. If we cancel the time limit for Z3 and perform SMT solving on these 5 tasks, the corresponding solving time is 2377.9s, 1997.5s, 5408.9s, 2030.3s, and 13692.0s. Our tactic is significantly faster than Z3 on these 5 cases in TSO. Symmetrically, 2 tasks are timed out in ZPRE but can be solved by Z3 with 683.4s and 1687.4s. Again, if we cancel the time limit, the reported solving time of ZPRE on these 2 cases is 2903.5s and 2991.2s. In these 2 cases, our approach are inferior to the default SMT solving strategy.

In PSO, the time consumptions of Z3 and ZPRE on the 1588 both-solved cases are 23477.4s and 12393.0s, respectively – ZPRE is 1.89x times faster than Z3. There is one task on the right boundary of Figure 8 that timed out in Z3, but can be solved by ZPRE with 485.3s. If we cancel the time limit, Z3 solved this task with 2753,7s. Systematically, there is one case timed out in ZPRE but can be solved by Z3 with 834.4s. Cancel the time limit; ZPRE spent 2119.3s to solve this task. In this case, our approach is not as efficient as the default SMT solving. However, the overall results in PSO indicate that ZPRE is obviously faster than Z3.

Figure 10 and Figure 11 show the SMT solving time on each subcategory under TSO and PSO, respectively. On some non-trivial and representative subcategories, e.g., *pthread*, *divine*, *ext*, *lit*, and *wmm*, our method is significantly superior to Z3 in TSO with a speedup (in Figure 10) of 2.11x, 2.55x, 1.59x, 1.27x, and 2.15x, respectively. The speedups of our approach under PSO on the same subcategories (in Figure 11) are 1.94x, 2,48x, 1.82x, 1.58x, and 2.55x, respectively. The detailed statistics of different subcategories indicates that our tactic is applicable to various concurrent programs and has a obvious improvements over Z3.

Table 2. The number of decisions, propagations, and conflicts of Z3 vs. ZPRE

MM	(Z3/ZPRE, #Ratio)		
	Decisions (10^7)	Propagations (10^9)	Conflicts (10^6)
SC	(6.09/4.24, 1.43x)	(1.84/1.17, 1.57x)	(4.51/1.94, 2.30x)
TSO	(7.22/5.19, 1.39x)	(4.91/2.88, 1.70x)	(6.12/5.12, 1.20x)
PSO	(10.61/8.87, 1.20x)	(2.67/1.44, 1.85x)	(2.81/2.00, 1.40x)

5.2 Result Analysis

There is a cluster of points in the bottom left of Figures 6, 7, and 8. These tasks are solved extremely fast by Z3 and ZPRE, and some even slightly faster without our tactic. SMT solving time on these tasks is short because they are trivial tasks, or the bug occurs at a low depth, so results by applying our strategy are dominated by the time taken to recognize and rearrange interference relations. Overhead of our tactic makes ZPRE inferior to Z3 on these tasks. However, as SMT files' size increases, it is clear that our method can bring on promising speedups.

Additionally, according to Figures 6, 7, and 8, there are several non-trivial cases on which our method is inferior to Z3, i.e., using only the default heuristics instead of combining with our tactic can finish the SMT solving earlier. This is understandable since our method is also heuristics-based, and there is no guarantee for our tactic to always make the best choice. Nevertheless, our method works for most of the SMT instances.

Table 2 shows the statistics of Z3 and ZPRE related to the search process of DPLL(T), including the numbers of decisions, propagations, and conflicts on both-solved cases. According to the second and third columns, ZPRE makes fewer guesses and propagations than Z3 during the search procedure. Moreover, from the last column in Table 2, ZPRE

Table 3. Summary of results: Z3 vs. ZPRE⁻ vs. ZPRE

MM	SMT Files	#Both-Solved			Z3		ZPRE⁻			ZPRE		
		Total	True	False	TO	CPU_Time(s)	TO	CPU_Time(s)	#Speedup	TO	CPU_Time(s)	#Speedup
SC	1674	1589	659	930	81	14344.3	78	11358.5	**1.26x**	77	9506.9	**1.49x**
TSO	1651	1592	632	960	56	20393.4	53	14726.5	**1.37x**	53	10922.7	**1.87x**
PSO	1643	1588	482	1106	53	23477.4	53	15557.2	**1.51x**	53	12393.0	**1.89x**

also meets fewer conflicts than Z3 during the search procedure. Therefore, applying our approach helps DPLL(T) make more reasonable choices and reduces the redundant search space, the solving efficiency is thus improved.

We also implemented our strategy with the naive HEURISTIC 1, called ZPRE⁻. In other words, we prioritize interference variables over other variables in V, but we don't rank interference variables in V_{itf}. We made a comparison between Z3, ZPRE⁻, and ZPRE. The experimental results are summarized in Table 3. Note that as the memory model changes from SC to TSO and PSO, all the *false* tasks in SC are still *false* in TSO and PSO, and some *true* tasks flip to *false*. From the experimental results, relaxing some ordering constraints in TSO and PSO causes more safety property violations, especially when allowing the reordering of two write events that access different memory addresses.

The experimental results in Table 3 show that our tactic is more efficient in TSO and PSO than in SC. This is reasonable. In SC, no ordering constraint is relaxed. Assuming events a, b, c, d are from the same thread, we have the program order $clk(a) < clk(b) < clk(c) < clk(d)$. However, weak memory models relax some ordering constraints of neighboring events. If program order between b and c is relaxed, whether there is program order between a and c, as well as b and d should be further analyzed. Moreover, the effect of *fences* also bring additional restrictions on neighboring memory events. As a result, in weak memory models, more program orders need to be explicitly encoded, making the size of ordering constraints greater than in SC. However, changing the memory model does not affect the number of interference variables. Since interference variables are more important than other variables, compared to the random decision of DPLL(T), our tactic can show better performance in weak memory models than in SC.

Summary. Considering the both-solved tasks, ZPRE⁻ speeds up the SMT solving than Z3 by 1.26x times in SC, 1.37x times in TSO, and 1.51x in PSO; and that number of ZPRE over Z3 is 1.49x in SC, 1.87x in TSO, and 1.89x in PSO, respectively. ZPRE is more efficient than ZPRE⁻. The overall result indicates that the further proposed methods in Section 4.1 can also accelerate the SMT solving for concurrent program verification. In conclusion, our interference relation-guided SMT solving tactic is effective and efficient.

Other Attempts. We also tried to combine our tactic with other strategies such as branching heuristics [14, 38]. Their method utilizes the control-flow information and prioritizes branch conditions during the SMT solving. However, benchmarks from the *ConcurrencySafety* category mainly focus on the multi-threading and atomicity, the number of branch-statements in these programs are small. Experimental results show that applying branching heuristics has no obvious improvement on this benchmark set.

5.3 Threats to Validity

The main threats to our method's validity are whether the performance improvements are due to our tactic and whether our implementation and experimental results are credible.

We force the decision order of DPLL(T) by applying our heuristics, so we compare ZPRE with the default solving strategy instantly. The improvements over Z3 must come from our tactic. Secondly, we detail the time comparison of each subcategory in three memory models to show ZPRE is effective towards different multi-threaded programs.

Moreover, when generating SMT formulas, we record thread information explicitly by naming the interference variables in a special fashion, and we do not modify the analysis process forcibly. Then we replace the default decision order of DPLL(T) (Section 4) with our heuristics in Z3. Implementation in CBMC and Z3 is simple and clear. Benchmarks are collected from *SV-COMP* 2019, one of the most representative and convincing open sources in program verification. We are thus confident in the effectiveness of our tactic.

6 Related Work

There are numerous researches on improving the performance of constraint solving in DPLL-based framework; and verification of concurrent programs under different memory models has been extensively studied. We discuss representative techniques in these two fields in recent years.

6.1 Heuristic of Decision Order

Many previous works focus on branching heuristics during constraint solving. VSIDS [38, 42] is a famous branching heuristic in CDCL [40] SAT solving. In VSIDS, each variable in each polarity has a counter, and the counter increases if a new conflict clause is inferred. Their method selects the unassigned literal with the highest counter as the next

Boolean variable. MOM [23] applies information from the backtrack search into the decision procedure. Marques-Silva [39] studies the practical impact of several branching heuristics used in SAT solving. Heuristic in DLIS [37, 41] selects the literal that appears most frequently in unresolved clauses as the next Boolean variable. However, all the aforementioned techniques focus on traditional satisfiable problems and only catch knowledge within the DPLL(T) framework.

Recently, Chen [14] applies the control-flow information to SMT solving. Their tactic decides branch condition in advance during DPLL(T) and prunes redundant search space significantly since branch condition implies whether a block of code can be executed. Yin [51] rearranges the decision order by assigning higher priority to the transition variables over other variables and applying structural information in SAT solving. In this paper, we utilize the interference relation of concurrent programs to guide SMT solving. There are some other related works on forcing decision order with heuristics. Gupta [27] proposes a BDD-based analysis procedure to generate learning clauses and apply them to SAT solving with several heuristics. One heuristic is that the Boolean variable in the backtrack point is likely to be useful, so they assign these Boolean variables that appeared in the backtrack point with higher priority. Another heuristic is that they keep track of Boolean variables at those levels to which the maximum number of backtracks have taken place. Chao [13] predicts and refines the SAT decision order for BMC by analyzing all previous unsatisfiable instances, then deciding these crucial variables in advance in the current instance.

In recent years, theory-based decision heuristics for DPLL(T) have been extensively studied. These approaches extract constraints under some specific theories and utilize these constraints to prune search space during constraint solving. Goldwasser [26] traverses the linear arrangement induced by the predicates in the formula instead of Boolean space in DPLL(T) on SMT benchmarks. Kuehlmann [34] uses circuit-based knowledge to guide SAT solving. They present a combination of Boolean reasoning techniques based on BDDs, structural transformations, and an SAT procedure for problems derived from circuit structures. Bruttomesso [10] advocates are utilizing structural information like equalities and arithmetic functions to reason bit-vector theory at a higher abstraction level rather than the traditional *bit-blasting*.

6.2 Concurrent Program Verification

Verification of concurrent programs is complicated due to the vast number of thread interferences and it suffers from the path explosion. The most efficient techniques to alleviate this problem include bounded model checking [6, 13, 30, 33], partial order reduction [2, 22, 31], abstraction refinement [11, 21, 45, 48, 49]. and stateless model checking [2, 5, 12, 29, 32].

Bounded model checking (BMC) limits the depth of loops or recursive functions to obtain the bounded program, which is expert in finding property violations. The vast majority of verification tools for verifying concurrent programs have employed BMC. Cordeiro [17] develops a lazy approach to abstract all possible interactions and calls an SMT solver to conduct constraint solving. Their approach reduces the state space by abstracting the interleaving from the conflict generated by the SMT solver. Given an unrolling bound k and an execution round r, [30] proposes a new technique named *Lazy-Sequentialization* to convert the origin program into a sequential program and simulate thread interaction in arbitrary order with an arbitrary number of statements. Our method also applies BMC to generate loop-free programs under different unrolling bounds. Our front end uses the ordering constraints between memory events to build possible interferences. Moreover, we utilize the interference relation to guide the search process of SMT solving. Our tactic is different from the above techniques.

Partial Order Reduction (POR) eliminates redundant traces by equivalence. Godefroid [25] systematically explores the state space of a concurrent program by dividing its executions via a run-time scheduler. DPOR[22] explores arbitrary interleaving of concurrent threads; it dynamically records backtrack points that identify alternative transitions that are not "equivalent" until it explores all alternative traces.

Abstraction divides actual program execution into fewer predicates, efficiently addressing the state space explosion problem. The method in [28] applies transition predicate abstraction to extract environment transitions about thread interleaving and uses recursion-free horn clauses to state abstraction refinement. An SAT-based framework named scheduling constraints-based abstraction refinement (SCAR) [50] adds conflict clauses in abstraction refinement iteratively to enhance the original formula. They use transitive closures to check the consistency of order relations efficiently.

Weak memory models (WMM) introduce an extra hurdle for verifying concurrent programs since it relaxes some ordering constraints between memory access events. Many groundbreaking works extend their approaches from SC to WMMs. State-of-the-art DPOR techniques [2–5, 29, 32] are elaborated to achieve maximal possible reduction for stateless model checking under WMMs. Tomasco [46] extends *Lazy-Sequentialization* to TSO and PSO. They replace memory accesses under WMMs with abstract operations on shared memory under SC and verify their validity. Yin [48] extends their SCAR technique from SC to WMMs by relaxing some order restrictions when building EOG.

7 Conclusions

In this paper, we presented an interference relation-guided tactic to accelerate SMT solving for multi-threaded program verification. Our tactic forces a decision order on interference variables and utilizes this decision order to guide the search process of DPLL(T). We implemented these heuristics in a prototype tool named Zpre and conducted experiments to compare Zpre with Z3 in SC, TSO, and PSO. Experimental results indicate that our tactic is effective and efficient.

Acknowledgments

This work is supported in part by the National Natural Science Foundation of China under Grant No. 62072267 and Grant No. 62021002, 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] [n.d.]. Software Verification Competition Benchmarks. https://gitlab.com/sosy-lab/benchmarking/sv-benchmarks/-/tree/svcomp19/.

[2] 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). Association for Computing Machinery, New York, NY, USA, 373–384. https://doi.org/10.1145/2535838.2535845

[3] Parosh Aziz Abdulla, Stavros Aronis, Mohamed Faouzi Atig, Bengt Jonsson, Carl Leonardsson, and Konstantinos Sagonas. 2015. Stateless Model Checking for TSO and PSO. *CoRR* abs/1501.02069 (2015). arXiv:1501.02069

[4] 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.

[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] 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.

[7] 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.

[8] 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

[9] M. Berzish, V. Ganesh, and Y. Zheng. 2017. Z3str3: A String Solver with Theory-aware Heuristics. In *2017 Formal Methods in Computer Aided Design (FMCAD)*. 55–59. https://doi.org/10.23919/FMCAD.2017.8102241

[10] Roberto Bruttomesso, Alessandro Cimatti, Anders Franzén, Alberto Griggio, Ziyad Hanna, Alexander Nadel, Amit Palti, and Roberto Sebastiani. 2007. A Lazy and Layered SMT(BV) Solver for Hard Industrial Verification Problems. In *Proceedings of the 19th International Conference on Computer Aided Verification* (Berlin, Germany). Springer-Verlag, Berlin, Heidelberg, 547–560.

[11] Franck Cassez and Frowin Ziegler. 2015. Verification of Concurrent Programs Using Trace Abstraction Refinement. In *Logic for Programming, Artificial Intelligence, and Reasoning*, Martin Davis, Ansgar Fehnker, Annabelle McIver, and Andrei Voronkov (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 233–248.

[12] 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

[13] Chao Wang, HoonSang Jin, G. D. Hachtel, and F. Somenzi. 2004. Refining the SAT decision ordering for bounded model checking. In *Proceedings. 41st Design Automation Conference, 2004.* 535–538.

[14] Jianhui Chen and Fei He. 2018. Control Flow-Guided SMT Solving for Program Verification. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering* (Montpellier, France). Association for Computing Machinery, New York, NY, USA, 351–361. https://doi.org/10.1145/3238147.3238218

[15] 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

[16] L. Cordeiro and B. Fischer. 2011. Verifying multi-threaded software using smt-based context-bounded model checking. In *2011 33rd International Conference on Software Engineering (ICSE)*. 331–340. https://doi.org/10.1145/1985793.1985839

[17] 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). Association for Computing Machinery, New York, NY, USA, 331–340. https://doi.org/10.1145/1985793.1985839

[18] 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

[19] 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.

[20] 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

[21] 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.

[22] 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). Association for Computing Machinery, New York, NY, USA, 110–121. https://doi.org/10.1145/1040305.1040315

[23] Jon W. Freeman. 1995. Improvements to propositional satisfiability search algorithms.

[24] Harald Ganzinger, George Hagen, Robert Nieuwenhuis, Albert Oliveras, and Cesare Tinelli. 2004. DPLL(T): Fast Decision Procedures. In *Computer Aided Verification*, Rajeev Alur and Doron A. Peled (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 175–188.

[25] 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). Association for Computing Machinery, New York, NY, USA, 174–186. https://doi.org/10.1145/263699.263717

[26] Dan Goldwasser, Ofer Strichman, and Shai Fine. 2008. A Theory-Based Decision Heuristic for DPLL(T). In *Proceedings of the 2008 International Conference on Formal Methods in Computer-Aided Design* (Portland, Oregon). IEEE Press, Article 13, 8 pages.

[27] A. Gupta, M. Ganai, Chao Wang, Zijiang Yang, and P. Ashar. 2003. Learning from BDDs in SAT-based bounded model checking. In *Proceedings of Design Automation Conference*. 824–829.

[28] 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). Association for Computing Machinery, New York, NY, USA, 331–344. https://doi.org/10.1145/1926385.1926424

[29] 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

[30] 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.

[31] 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

[32] 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

[33] 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.

[34] Andreas Kuehlmann, Malay K. Ganai, and Viresh Paruthi. 2001. Circuit-Based Boolean Reasoning. In *Proceedings of the 38th Annual Design Automation Conference* (Las Vegas, Nevada, USA). Association for Computing Machinery, New York, NY, USA, 232–237. https://doi.org/10.1145/378239.378470

[35] Leslie Lamport. 1978. Time, Clocks, and the Ordering of Events in a Distributed System. *Commun. ACM* 21, 7 (July 1978), 558–565. https://doi.org/10.1145/359545.359563

[36] Leslie Lamport. 1979. How to Make a Multiprocessor Computer That Correctly Executes Multiprocess Programs. *IEEE Transactions on Computers C-28* 9 (September 1979), 690–691.

[37] Chu Min Li and Anbulagan. 1997. Look-ahead versus look-back for satisfiability problems. In *Principles and Practice of Constraint Programming-CP97*, Gert Smolka (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 341–355.

[38] Jia Hui (Jimmy) Liang, Vijay Ganesh, Ed Zulkoski, Atulan Zaman, and Krzysztof Czarnecki. 2015. Understanding VSIDS Branching Heuristics in Conflict-Driven Clause-Learning, SAT Solvers. *CoRR* abs/1506.08905 (2015). arXiv:1506.08905

[39] João Marques-Silva. 1999. The Impact of Branching Heuristics in Propositional Satisfiability Algorithms. In *Progress in Artificial Intelligence*, Pedro Barahona and José J. Alferes (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 62–74.

[40] Joao Marques-Silva, Ines Lynce, and Sharad Malik. 2009. *Conflict-driven clause learning SAT solvers* (1 ed.). Number 1 in Frontiers in Artificial Intelligence and Applications. IOS Press, Netherlands, 131–153. https://doi.org/10.3233/978-1-58603-929-5-131

[41] J. P. Marques-Silva and K. A. Sakallah. 1999. GRASP: a search algorithm for propositional satisfiability. *IEEE Trans. Comput.* 48, 5 (May 1999), 506–521. https://doi.org/10.1109/12.769433

[42] 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). Association for Computing Machinery, New York, NY, USA, 530–535. https://doi.org/10.1145/378239.379017

[43] Scott Owens, Susmit Sarkar, and Peter Sewell. 2009. A Better x86 Memory Model: x86-TSO. In *Theorem Proving in Higher Order Logics*, Stefan Berghofer, Tobias Nipkow, Christian Urban, and Makarius Wenzel (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 391–407.

[44] Dennis Shasha and Marc Snir. 1988. Efficient and correct execution of parallel programs that share memory. *ACM Transactions on Programming Languages and Systems (TOPLAS)* 10, 2 (1988), 282–312.

[45] 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). Association for Computing Machinery, New York, NY, USA, 423–434. https://doi.org/10.1145/1926385.1926433

[46] E. Tomasco, T. L. Nguyen, O. Inverso, B. Fischer, S. L. Torre, and G. Parlato. 2016. Lazy sequentialization for TSO and PSO via shared memory abstractions. In *2016 Formal Methods in Computer-Aided Design (FMCAD)*. 193–200.

[47] D. Weaver and Tom Gremond. 1994. The SPARC architecture manual : version 9.

[48] 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). Association for Computing Machinery, New York, NY, USA, 645–655. https://doi.org/10.1145/3238147.3238223

[49] Liangze Yin, Wei Dong, Wanwei Liu, and Ji Wang. 2019. Parallel Refinement for Multi-Threaded Program Verification. In *Proceedings of the 41st International Conference on Software Engineering* (Montreal, Quebec, Canada) *(ICSE '19)*. IEEE Press, 643–653. https://doi.org/10.1109/ICSE.2019.00074

[50] 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 (2020), 549–565. https://doi.org/10.1109/TSE.2018.2864122

[51] L. Yin, F. He, and M. Gu. 2013. Optimizing the SAT Decision Ordering of Bounded Model Checking by Structural Information. In *2013 International Symposium on Theoretical Aspects of Software Engineering*. 23–26. https://doi.org/10.1109/TASE.2013.11

A Artifacts Appendix

The artifact of this paper consists of the source code of our modified CBMC and Z3, benchmarks, and scripts for performing the evaluation described in Section 5.

A.1 Preparation

For running this experiment, some preparation is required:

- To install dependencies, just run the following command:

  ./install_dependencies.sh

- Pre-compiled (under `Ubuntu 20.04`) binaries `cbmc` `z3` which implement our algorithms are available. If your want to recompile them, just run the following command:

  ./compile.sh

  then `cbmc` and `z3` will be compiled and copied to the current folder.

A.2 Getting Started

This section shows how to set up the artifact quickly in a small subset of benchmarks (in *benchmarks/* folder).

- To conduct the evaluation, just run the following command:

  ./run.sh

  it will finish within 30 minutes.

Firstly, *run.sh* calls `cbmc` to generate SMT files from the *benchmarks/* folder, which contains a small subset of benchmarks (randomly selected from solvable examples). Three new folders – *smt_sc/*, *smt_tso/*, and *smt_pso/* will be created; they contain the generated SMT files under SC/TSO/PSO memory models.

Secondly, *run.sh* calls `z3` to perform SMT solving with *default/partial-pre/all-pre* solving strategies. Three new folders – *results-sc/*, *results-tso/*, and *results-pso/* will be generated; they contain log files under SC/TSO/PSO memory models.

Finally, three excel files – *sc.xlsx*, *tso.xlsx*, and *pso.xlsx* will be generated; they correspond to the solving time of `z3` with different strategies under SC/TSO/PSO memory models.

A.3 Full Experiment

This section shows how to set up the artifact in all the benchmarks (in *benchmarks_all folder*):

- To conduct the evaluation, just run:

  ./run.sh ./benchmarks_all

  it will take dozens of hours to finish this experiment.

The detailed procedure is the same as in *Getting Started.* 16GB memory and 80GB free disk are needed to run the complete experiment.