Scheduling Constraint Based Abstraction Refinement for Multi-Threaded Program Verification

Article *in* IEEE Transactions on Software Engineering · August 2017

DOI: 10.1109/TSE.2018.2864122

4 authors, including:

Wei Dong
National University of Defense Technology
88 PUBLICATIONS **481** CITATIONS
SEE PROFILE

Wanwei Liu
National University of Defense Technology
46 PUBLICATIONS **131** CITATIONS
SEE PROFILE

Scheduling Constraint Based Abstraction Refinement for Multi-Threaded Program Verification

LIANGZE YIN, School of Computer, National University of Defense Technology, China
WEI DONG, School of Computer, National University of Defense Technology, China
WANWEI LIU, School of Computer, National University of Defense Technology, China
JI WANG, School of Computer, National University of Defense Technology, China

Bounded model checking is among the most efficient techniques for the automatic verification of concurrent programs. However, encoding all possible interleavings often requires a huge and complex formula, which significantly limits the salability. This paper proposes a novel and efficient abstraction refinement method for multi-threaded program verification. Observing that the huge formula is usually dominated by the exact encoding of the scheduling constraint, this paper proposes a scheduling constraint based abstraction refinement method, which avoids the huge and complex encoding of BMC. In addition, to obtain an effective refinement, we have devised two graph-based algorithms over event order graph for counterexample validation and refinement generation, which can always obtain a small yet effective refinement constraint. Enhanced by two constraint-based algorithms for counterexample validation and refinement generation, we have proved that our method is sound and complete w.r.t. the given loop unwinding depth. Experimental results on SV-COMP 2017 benchmarks indicate that our method is promising and significantly outperforms the existing state-of-the-art tools.

CCS Concepts: • **Software and its engineering → Software verification and validation**;

Additional Key Words and Phrases: Multi-Threaded Program, Bounded Model Checking, Scheduling Constraint, Event Order Graph

ACM Reference format:
Liangze Yin, Wei Dong, Wanwei Liu, and Ji Wang. 2017. Scheduling Constraint Based Abstraction Refinement for Multi-Threaded Program Verification. 1, 1, Article 1 (August 2017), 26 pages.
https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

Facilitated by the popularization of multi-core architectures, concurrent programs are becoming popular to take full advantage of the available computing resources. However, due to the nondeterministic thread interleavings, traditional approaches such as testing and simulation are hard to guarantee the correctness of such programs. Automatic program verification has become an important complementary to traditional approaches. Given that most of the errors can be detected with small loop unwinding depths, bounded model checking (BMC) has been proven to be one of the most efficient techniques for the automatic verification of concurrent programs [6, 33]. However, due to the complex inter-thread communication, a huge encoding is usually required to offer an exact description of the concurrent behavior, which greatly limits the scalability of BMC for concurrent programs.

This paper focuses on multi-threaded programs based on shared variables and *sequential consistency (SC)* [2]. For these programs, we have observed that the *scheduling constraint*, which defines that "for any pair $\langle w, r\rangle$ s.t. r reads the value of a variable v written by w, there should be no other write of v between them," significantly contributes to the complexity of the behavior encoding. In

2017. XXXX-XXXX/2017/8-ART1 $15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

the existing work of BMC, to encode the scheduling constraint, each access of a shared variable is associated with a "clock variable". The scheduling constraint is then encoded into a complicated logic formula over the state and clock variables, the size of which is cubic in the number of shared memory accesses [3].

Inspired by this observation, this paper proposes a novel method for multi-threaded program verification which performs abstraction refinement by weakening and strengthening the scheduling constraint. It initially ignores the scheduling constraint and then obtains an over-approximation abstraction of the original program (w.r.t. the given loop unwinding depth). If the property is safe on the abstraction, then it also holds on the original program. Otherwise, a counterexample is obtained and the abstraction is refined if the counterexample is infeasible. N. Sinha and C. Wang also performed abstraction refinement to deal with the overhead of an exact encoding of the concurrent behavior [35]. However, their abstraction model was performed by restricting the sets of read events and read-write links, while we consider all read events and read-write links but ignore the scheduling constraint.

The efficiency of our method depends on the number of iterations required to verify the property and the sizes of the constraints added during the refinement process. Another innovation of this paper is that, to verify the property with a small number of small problems, we have devised two graph-based algorithms over event order graph (EOG) for counterexample validation and refinement generation, s.t. an effective refinement constraint can be obtained in each refinement iteration. Whenever an abstraction counterexample is determined to be infeasible, we can always obtain a set of "core kernel reasons" of the infeasibility, which can usually be encoded into simple constraints and reduce a large amount of space. In our experiments, most of the programs can be verified within dozens of refinement iterations. Meanwhile, the increased size of the abstraction during the refinement process can usually be ignored compared with that of the initial abstraction.

Our graph-based EOG validation method is effective in practice. Given an infeasible EOG, it can usually identify the infeasibility with rare exceptions. If it is not sure whether an EOG is feasible or not, we explore a constraint-based EOG validation process to further validate its feasibility. If an infeasibility is returned, we explore a constraint-based refinement generation process to refine the abstraction. Enhanced by these two constraint-based processes, we have proved that our method is sound and complete w.r.t the given loop unwinding depth.

We have implemented the proposed method on top of CBMC and applied it to the benchmarks in the concurrency track of SV-COMP 2017 [36]. The experimental results demonstrate that our method drastically improves the verification performance. Without the scheduling constraint, the formula size reduces to 1/8 on average, whereas the number of CNF clauses increased during the refinement process can usually be ignored compared with that of the abstraction. Moreover, our tool has successfully verified all these examples within 1550 seconds and 43 GB of memory. By contrast, Lazy-CSeq-Abs | a leading tool for concurrent program verification | spended 9820 seconds and 104 GB memory to achieve the same score. Our tool has won the gold medal in the concurrency track of SV-COMP 2017 [36] (Warning: It will violate our anonymity).

The contributions of this paper are listed as follows.

(1) This paper presents a scheduling constraint based abstraction refinement method for multi-threaded program verification, which avoids the huge and complex constraint to encode the concurrent behavior.
(2) This paper presents two graph-based algorithms over event order graph for counterexample validation and refinement generation, which can always obtain a small yet effective refinement constraint in practice.

(3) To ensure the soundness, we have enhanced our method by two constraint-based algorithms for counterexample validation and refinement generation. In this manner, a both efficient and sound method for multi-threaded program verification is obtained.
(4) We have implemented our method on top of CBMC. The evaluation on the SV-COMP 2017 benchmarks indicates that our method is promising and significantly outperforms the existing state-of-the-art tools.

The rest of this paper is organized as follows. Section 2 introduces the preliminaries. Section 3 outlines and illustrates our proposed method by presenting a running example. Sections 4 and 5 present our EOG-based counterexample validation and refinement generation algorithms, respectively. Section 6 discusses the soundness and efficiency of our method. Section 7 provides the experimental results. Section 8 reviews the related work, and Section 9 concludes the paper.

2 PRELIMINARIES

2.1 Multi-Threaded Program

A *multi-threaded* program P consists of $N \geq 1$ concurrent threads P_i ($1 \leq i \leq N$). It contains a set of variables which are partitioned into *shared variables* and *local variables*. Each thread P_i can read/write both the shared variables and its local variables. We focus on programs based on PThreads, one of the most popular libraries for multi-threaded programming. It uses `pthread_create(&t, &attrib, &f, &args)` to create a new thread `t`, and `pthread_join(t, &_return)` to suspend the current thread until thread `t` terminates [1].

In this paper, we assume each variable access is atomic. We also assume that each statement t of a multi-threaded program is either 1) a *global statement* that contains only one shared variable access (it may further contain multiple local variable accesses) or 2) a *local statement* that only operates on local variables. A statement with multiple shared variable accesses can always be translated to a sequence of global statements. By defining the expressions suitably and using source-to-source transformations, we can model all statements using global and local statements.

We also assume that all functions are inlined and all loops are unwound by a limited depth (it is a basic proviso in BMC). We also omit the discussion on modeling the sophisticated C language data elements, such as pointers, structures, arrays, and heaps, etc., because they are irrelative with the concurrency and we deal with them in the same way as CBMC does. The discussion on PThread primitives, such as `pthread_mutex_lock` and `pthread_mutex_unlock`, etc., are also omitted in this paper. In CBMC, they are implemented by shared variables and we deal with them in the same way as CBMC does.

Given a multi-threaded program, we write $\mathbb{V}$ for the set of shared variables. An *event* e is a read/write access to a shared variable. We use $\mathbb{E}$ to denote all of them. Each global statement corresponds to an event, i.e., the event contained in the global statement. Each $e \in \mathbb{E}$ is associated with an element $\text{var}(e) \in \mathbb{V}$, a type $\text{type}(e)$, and a literal $\text{guard}(e)$, which represent the accessed variable, the type of access, and the guard condition literal, respectively. $\text{type}(e)$ can be either `rr` (i.e., "read") or `ww` (i.e., "write"). Any event e with $\text{var}(e) = v$ and $\text{type}(e) = \texttt{rr}$ (resp. $\text{type}(e) = \texttt{ww}$) is called a read (resp. a write) of v. To express the execution orders of different events, we also associate each event with an unique natural number $\text{clk}(e)$. $\text{clk}(e_1) < \text{clk}(e_2)$ represents that e_1 executes before e_2.

The program P determines a partial order $\prec_P^0 \subset \mathbb{E} \times \mathbb{E}$. Intuitively, $e_1 \prec_P^0 e_2$ (or we write $(e_1, e_2) \in \prec_P^0$) indicates that "e_1 should happen before e_2 according to the *program order* of P". According to the program order, $e_1 \prec_P^0 e_2$ holds in the following three cases.

[1] More information about PThreads can be found at https://computing.llnl.gov/tutorials/pthreads/

- For any two events e_1 and e_2 of the same thread, if e_1 must happen before e_2 according to the sequential semantics, then we have $e_1 \prec_P^0 e_2$.
- If `pthread_create()` is used to create a new thread `t` at some point p of the current thread, then for any event e_1 of the current thread before p and e_2 of the thread `t`, we have $e_1 \prec_P^0 e_2$.
- If `pthread_join(t)` is used to suspend the current thread at some point p, then for any event e_1 of the thread `t` and e_2 of the current thread after p, we have $e_1 \prec_P^0 e_2$.

A read-write link (e_1, e_2) represents that "e_2 reads the value written by e_1". Therefore, type(e_1) = `ww`, type(e_2) = `rr`, var(e_1) = var(e_2), and the value of e_2 is equal to that of e_1. In addition, there should be no other "write" of var(e_1) happening between them. Given a read-write link $\lambda := (e_1, e_2)$, we denote by sel(λ) the *read-write link literal* (a boolean variable) that represents the link.

2.2 Bounded Model Checking

Bounded model checking [7] is one of the most applicable techniques to alleviate the state space explosion problem in concurrent program verification. Given that most of the errors can be detected with small loop unwinding depths, the unwinding depth for those loops and recursions is limited [33]. Instead of explicitly enumerating all thread interleavings, BMC employs a symbolic representation to encode the verification problem, which is then solved by a SAT/SMT solver. If a positive answer is given, then a satisfying assignment corresponding to a feasible counterexample is acquired. Otherwise, the program is proven safe w.r.t. the given loop unwinding depth.

In BMC, the monolithic encoding of a multi-threaded program is usually represented as $\alpha := \phi_{init} \wedge \rho \wedge \zeta \wedge \xi$, where ϕ_{init} is the initial states, ρ encodes each thread in isolation, ζ formulates that "each read of a variable v may read the result of any write of v", and ξ formulates the scheduling constraint, which defines that "for any pair $\langle w, r \rangle$ s.t. r reads the value of a variable v written by w, there should be no other write of v between them" [3].

3 METHOD OVERVIEW AND ILLUSTRATION

3.1 Method Overview

The performance of BMC is usually decided by that of the constraint solving, the performance of which depends significantly on the size of the constraint problem. Hence, an important way to improve the performance of BMC is to reduce the size of the constraint problem. In BMC of multi-threaded programs, we have observed that the monolithic encoding α is usually dominated by the scheduling constraint ξ. To reduce the constraint problem size, we propose to ignore the scheduling constraint in the constraint solving process. An abstraction of the monolithic encoding is then obtained, which is defined as follows.

Definition 3.1. Given a multi-threaded program, the abstraction ignoring the scheduling constraint can be formulated as $\varphi_0 := \phi_{init} \wedge \rho \wedge \zeta$, where ϕ_{init} is the initial states, ρ encodes each thread in isolation, and ζ formulates that "each read of a variable v may read the result of any write of v".

The scheduling constraint ξ defines a set of order requirements among the events. All of them should be satisfied for any concrete execution of a multi-threaded program. Hence, whenever a counterexample π of the abstraction is obtained, further validation is required to determine whether π satisfies all the order requirements defined in ξ. If that is true, π is feasible. Otherwise, π is infeasible. In other words, it may not correspond to a concrete execution. In this case, we continually search the rest of the abstraction space for another new counterexample, until a feasible counterexample is found or all counterexamples of the abstraction have been determined to be infeasible.

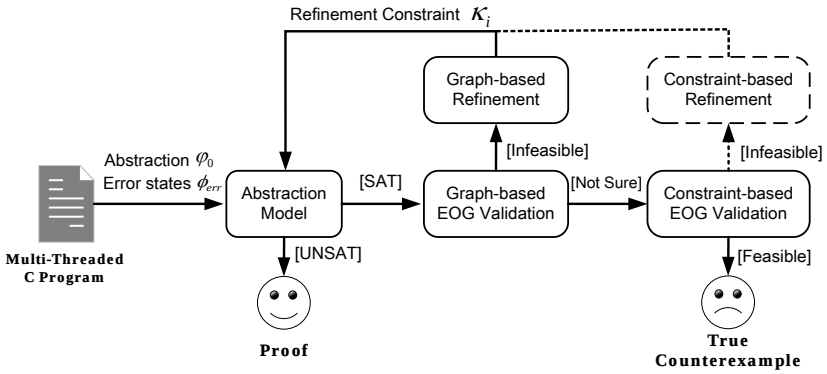


Fig. 1. An overview of our method

Fig. 1 presents an overview of our method. Given a multi-threaded C program, we first add the abstraction φ_0 and the error states ϕ_{err} to the abstraction model. If it is unsatisfiable, then the property is proven safe w.r.t. the given loop unwinding depth. Otherwise, a counterexample of the abstraction is provided. Given that the scheduling constraint is ignored in the abstraction, this counterexample may be infeasible and further validation is required. In our method, the feasibility of an abstraction counterexample is determined via validating the feasibility of its corresponding event order graph (EOG). An intuitive method for EOG validation is constraint solving. If the EOG is infeasible, then the abstraction is refined by exploring the unsatisfiable core. However, this method is not effective for refinement generation (cf. Section 5). To obtain an effective refinement, we have devised two graph-based algorithms over EOG for EOG validation and refinement generation, in which a small yet effective refinement can always be obtained if the EOG is determined to be infeasible. However, this method is not complete. It can only give an infeasible answer (cf. Section 4). To make our method both efficient and sound, we first adopt the graph-based EOG validation method. If the EOG is determined to be infeasible, the graph-based refinement process is performed to obtain an effective refinement constraint. Otherwise, we employ the constraint-based validation process to further validate the EOG. Our experiments demonstrate that our graph-based EOG validation method is effective in practice. It can always identify the infeasibility of an infeasible EOG with rare exceptions. Actually, the constraint-based refinement generation process (the dashed part of Fig. 1) has never been invoked on SV-COMP 2017 benchmarks.

3.2 Method Illustration

We provide a running example to illustrate our method. The program involves three threads, namely, `main`, `thr1`, and `thr2`, as shown in Fig. 2(a). In this example, the set of shared variables $\mathbb{V} := \{x, y, m, n\}$, which are initialized to $\{1, 1, 0, 0\}$, respectively. The `main` thread creates threads `thr1` and `thr2`, and then waits until these two threads terminate. We attempt to verify that it is impossible for both m and n to be 1 after the exit of `thr1` and `thr2`. This program offers a modular proof to this property.

Initial abstraction. We use ϕ_{init} and ϕ_{err} to denote the initial and error states, respectively, while ρ_{main}, ρ_{thr1}, and ρ_{thr2} denote the transition relationships of these three threads, respectively, and ρ is the conjunction of those transition relationships of all threads.

To encode the program, as shown in Fig. 2(b), we convert the original program into a set of *static single assignment* (SSA) statements, in which the program variables are renamed s.t. each

```
int x = 1, y = 1, m = 0, n = 0;
void* thr1(void * arg) {
  x = y + 1;
  m = y;
  x = 0;
}
void* thr2(void * arg) {
  y = x + 1;
  n = x;
  y = 0;
}
void 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 = 1, y1 = 1, m1 = 0, n1 = 0;
void* thr1(void * arg) {
  x2 = y2 + 1;
  m2 = y3;
  x3 = 0;
}
void* thr2(void * arg) {
  y4 = x4 + 1;
  n2 = x5;
  y5 = 0;
}
void 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 (!(m3 == 1 && n3 == 1));
}
```

(b) The SSA statements of the program

Fig. 2. A three-thread program

variable is assigned only once. Particularly, each "read" of any shared variable also has a unique name. Then ϕ_{init}, ϕ_{err}, and ρ are defined as follows. Note that the transition relationship of each thread (such as ρ_{main}, ρ_{thr1}, and ρ_{thr2}) encodes that thread in isolation, i.e., it doesn't consider the thread communications.

$$
\begin{aligned}
\phi_{init} &:= (x_1 = 1) \wedge (y_1 = 1) \wedge (m_1 = 0) \wedge (n_1 = 0) \\
\phi_{err} &:= (m_3 = 1) \wedge (n_3 = 1) \\
\rho_{thr1} &:= (x_2 = y_2 + 1) \wedge (m_2 = y_3) \wedge (x_3 = 0) \\
\rho_{thr2} &:= (y_4 = x_4 + 1) \wedge (n_2 = x_5) \wedge (y_5 = 0) \\
\rho_{main} &:= true \\
\rho &:= \rho_{thr1} \wedge \rho_{thr2} \wedge \rho_{main}
\end{aligned}
$$

To encode ζ, we must identify the behavior of every "read" event. Consider the shared variable x for example. There are five read/write accesses to the variable x. For each access, as shown in Fig. 2(b), we rename x to a unique name in the SSA statements, i.e., $x_1, x_2, \cdots, x_5$. We denote by e_{x_i} $(1 \leq i \leq 5)$ the event corresponding to x_i. Then $\{e_{x_1}, e_{x_2}, e_{x_3}\}$ and $\{e_{x_4}, e_{x_5}\}$ are the sets of "writes" and "reads" of x, respectively. We use a read-write link literal $s_{v,i,j}$ to indicate that e_{v_j} reads the value written by e_{v_i} $(v \in \mathbb{V})$. The encoding ψ_{x_i} $(i = 4, 5)$, defined below, indicates that the value of x_i can take any value of x_1, x_2, and x_3. Given that the variable x_i can not take several different values simultaneously, the formula $s_{x,4,1} \vee s_{x,4,2} \vee s_{x,4,3}$ represents that, among these three literals, there is one and only one true literal. We denote by ζ_x the conjunction of ψ_{x_4} and ψ_{x_5}. It formulates the possible behaviors of all "reads" of x.

$$
\begin{aligned}
\psi_{x_4} := \quad & (s_{x,4,1} \Rightarrow (x_4 = x_1)) \wedge \\
& (s_{x,4,2} \Rightarrow (x_4 = x_2)) \wedge \\
& (s_{x,4,3} \Rightarrow (x_4 = x_3)) \wedge \\
& (s_{x,4,1} \vee s_{x,4,2} \vee s_{x,4,3}) \\
\psi_{x_5} := \quad & (s_{x,5,1} \Rightarrow (x_5 = x_1)) \wedge \\
& (s_{x,5,2} \Rightarrow (x_5 = x_2)) \wedge \\
& (s_{x,5,3} \Rightarrow (x_5 = x_3)) \wedge \\
& (s_{x,5,1} \vee s_{x,5,2} \vee s_{x,5,3}) \\
\zeta_x := \quad & \psi_{x_4} \wedge \psi_{x_5}
\end{aligned}
$$

Similarly, we obtain the corresponding formulas of ζ_y, ζ_m, and ζ_n. Let $\zeta := \zeta_x \wedge \zeta_y \wedge \zeta_m \wedge \zeta_n$. The initial abstraction can then be formulated as follows.

$$\varphi_0 := \phi_{init} \wedge \rho \wedge \zeta \tag{1}$$

Constraint solving of the first round. Using $\varphi_0 \wedge \phi_{err}$ as input to a constraint solver will return SAT and yield a counterexample π_0, which is a set of assignments to the variables in $\varphi_0 \wedge \phi_{err}$.

Counterexample validation of the first round. Given that the scheduling constraint is excluded from the abstraction, such a counterexample may be infeasible. In our method, a counterexample π is validated via validating its corresponding EOG (cf. Section 4), which captures all the order requirements among the events of π. We first employ the graph-based EOG validation method to determine its feasibility.

Fig. 3 shows the EOG corresponding to π_0. In figures that describe EOGs, the white and gray nodes denote "writes" and "reads" occurring in the corresponding counterexample, respectively. A solid arrow with a triangular head from e_1 to e_2 represents a *program order*, which requires that e_1 should happen before e_2. A dashed arrow from e_1 to e_2 represents a *read-write link* (e_1, e_2). It requires that 1) e_1 should happen before e_2, and 2) no "write" of $\text{var}(e_1)$ can happen between them. A solid arrow with a hollow head from e_1 to e_2 represents a *derived order*, which is derived from existing order requirements. It also requires that e_1 should happen before e_2. For brevity, in these figures, we use the subscripts of an event as labels, that is, we use v_i to represent the event e_{v_i}. Now the question is: Is there any total order of all these nodes that satisfies all these order requirements? This is not a trivial problem. However, if there exists some cycle in the graph, then the answer must be "no", and the counterexample is infeasible.

Given a counterexample π and two events e_1 and e_2, we use $e_1 \prec_\pi e_2$ to represent that e_1 should happen before e_2 in π. By applying our graph-based EOG validation algorithm (cf. Section 4), we can deduce two derived orders $e_{y_3} \prec_{\pi_0} e_{y_4}$ and $e_{x_5} \prec_{\pi_0} e_{x_2}$, which are denoted by h_1 and h_2 respectively [2]. Fig. 3 shows two cycles, including $C_1 : e_{x_2} \prec_{\pi_0} e_{y_3} \prec_{\pi_0} e_{y_4} \prec_{\pi_0} e_{x_5} \prec_{\pi_0} e_{x_2}$ and $C_2 : e_{x_2} \prec_{\pi_0} e_{x_4} \prec_{\pi_0} e_{y_4} \prec_{\pi_0} e_{x_5} \prec_{\pi_0} e_{x_2}$. Therefore, π_0 is infeasible.

Refinement of the first time. To prune more search space rather than just one counterexample, we should find the "kernel reasons" that make the counterexample infeasible. According to our graph-based kernel reason analysis algorithm (cf. Section 5), the derived orders h_1 and h_2 are caused by r_1 and r_3, respectively (according to Rule 3). h_1 is derived as follows. According to the order requirements of r_1, we have that $y_1 \prec_{\pi_0} y_3$, and no "write" of y can be executed between y_1 and y_3. According to the program orders, we have $y_1 \prec_{\pi_0} y_4$. Hence, we can deduce that $y_3 \prec_{\pi_0} y_4$. Given that the guard conditions for all these events are true, as long as r_1 holds in the counterexample, we may obtain the derived order h_1. Hence, the reason of h_1 is r_1. Similarly, we can obtain that the

[2] We can deduce more orders from the EOG, but we only list h_1 and h_2, because they will be used later.

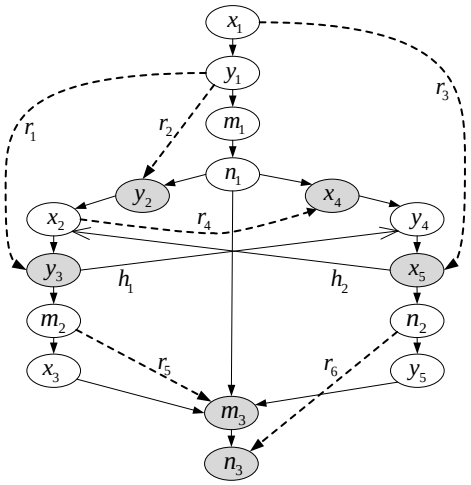


Fig. 3. EOG of counterexample π_0.

reason of h_2 is r_3. Given that the guard conditions for all these events are true, the reason for any program order is TRUE. And according to Section 5, the reason for any read-write link is itself.

A kernel reason of a cycle C is a conjunction of those kernel reasons for those orders constructing C. We can obtain that C_1 is caused by $r_1 \wedge r_3$, and C_2 is caused by $r_3 \wedge r_4$. Given that r_1, r_3, and r_4 are represented by read-write link literals $s_{y,3,1}$, $s_{x,5,1}$, and $s_{x,4,2}$, respectively, we obtain that the kernel reason of C_1 is $\{s_{y,3,1}, s_{x,5,1}\}$, and the kernel reason of C_2 is $\{s_{x,5,1}, s_{x,4,2}\}$. We hence use $\kappa_0 := \neg(s_{y,3,1} \wedge s_{x,5,1}) \wedge \neg(s_{x,5,1} \wedge s_{x,4,2})$ as the refinement constraint, which contains only two simple CNF clauses.

Second constraint solving. Let $\varphi_1 := \varphi_0 \wedge \kappa_0$. When using $\varphi_1 \wedge \phi_{err}$ as input, the constraint solver returns SAT again, and produces a new counterexample π_1.

Second counterexample validation. By applying our graph-based EOG validation algorithm, the EOG corresponding to π_1 has two cycles. Hence, π_1 is also infeasible.

Refinement, the second time. Again, we apply our graph-based kernel reason analysis algorithm. The refinement constraint is formulated as $\kappa_1 := \neg(s_{x,4,3} \wedge s_{x,5,1}) \wedge \neg(s_{y,2,4} \wedge s_{x,4,3}) \wedge \neg(s_{y,4,3} \wedge s_{x,4,3})$, which contains three simple CNF clauses. Here, one of these two cycles has two different kernel reasons.

Third constraint solving. Same as before, let $\varphi_2 := \varphi_1 \wedge \kappa_1$. When using $\varphi_2 \wedge \phi_{err}$ as input to a constraint solver, we get another counterexample π_2.

Third counterexample validation. By applying our graph-based EOG validation algorithm, the EOG corresponding to π_2 also has two cycles. Hence, π_2 is also infeasible.

Refinement, the third time. By applying our graph-based kernel reason analysis algorithm, we obtain a new refinement constraint $\kappa_2 := \neg(s_{y,3,1} \wedge s_{y,2,5}) \wedge \neg(s_{x,5,2} \wedge s_{y,2,5})$, which contains two simple CNF clauses.

Constraint solving, the fourth time. Let $\varphi_3 := \varphi_2 \wedge \kappa_2$. When using $\varphi_3 \wedge \phi_{err}$ as input, the constraint solver returns UNSAT this time, which indicates that the property is safe.

Given that the serial of constraint solving queries are incremental, they can be solved in an incremental manner. From this example, we can observe that:

(1) Without the scheduling constraint, the size of the abstraction is much smaller than that of the monolithic encoding. In this example, excluding the 3049 CNF clauses encoding the `pthread_create` and `pthread_join` function calls, the monolithic encoding contains 10214 CNF clauses, while the abstraction φ_0 contains only 1018 CNF clauses.
(2) With our graph-based refinement generation method, the verification problem can usually be solved with a small number of refinement iterations. In this example, only three refinements are required to verify the property.
(3) With our graph-based refinement generation method, the number of clauses increased during the refinement process can usually be ignored compared with that of the abstraction. In this example, only 7 simple CNF clauses are added during the refinement process, while the initial abstraction contains 4067 CNF clauses.
(4) Our graph-based EOG validation method is effective to identify the infeasibility in practice. In this example, all the four abstractions are infeasible. All of them can be detected by our graph-based EOG validation process. The constraint-based EOG validation and refinement processes have never been invoked.

4 EOG-BASED COUNTEREXAMPLE VALIDATION

4.1 Counterexample and Event Order Graph

Definition 4.1. A counterexample π of an abstraction φ_i, or a counterexample π for short, is a set of assignments to the variables in $\varphi_i \wedge \phi_{err}$, where ϕ_{err} is the error states.

A counterexample π is an execution of the abstraction that falsifies the property. It defines a trace for each thread and the read-write relationship among the "reads" and "writes" occurring in π. Given that the scheduling constraint is ignored in the abstraction, the counterexample may not be feasible, i.e., it may not correspond to any concrete execution. Note that the execution order of those statements from different threads is not defined. If the counterexample is feasible, it may correspond to multiple concrete executions.

Given a counterexample π, we use $\mathbb{E}_\pi \subseteq \mathbb{E}$ to denote the set of events occurring in π. We define a partial order $\prec_\pi \subseteq \mathbb{E}_\pi \times \mathbb{E}_\pi$. Intuitively, $e_1 \prec_\pi e_2$ represents that "e_1 should happen before e_2 in π", i.e., $\mathrm{clk}(e_1) < \mathrm{clk}(e_2)$. We also use $\prec_\pi^0$ to denote the restriction of $\prec_P^0$ on π. In this case, $\prec_\pi^0 \subseteq \prec_P^0$, and we have $\prec_\pi^0 \subseteq \prec_\pi$. We also focus on the partial order $\lhd_\pi$, which is called the *read-from* relationship of π. $e_1 \lhd_\pi e_2$ (or we write $(e_1, e_2) \in \lhd_\pi$) represents that "(e_1, e_2) is a read-write link in π". According to this definition, we obtain $\lhd_\pi \subseteq \prec_\pi$. An element in $\prec_\pi^0$ (resp. in $\lhd_\pi$) is called a *program order* (resp. *read-from order*) of π.

A counterexample π defines a quadruple $\langle s_{0,\pi}, \mathbb{T}_\pi, \mathbb{E}_\pi, \lhd_\pi \rangle$, where $s_{0,\pi}$ is the initial states, $\mathbb{T}_\pi$ is the set of statements contained in π, $\mathbb{E}_\pi$ is the set of events occurring in π, and $\lhd_\pi \subseteq \mathbb{E}_\pi \times \mathbb{E}_\pi$ is the read-from relationship that links each "read" $r \in \mathbb{E}_\pi$ to a "write" $w \in \mathbb{E}_\pi$ s.t. r reads the value written by w. Note that $\prec_\pi^0$ is the restriction of $\prec_P^0$ to $\mathbb{E}_\pi$. Table 1 lists the set of notations we use for a counterexample π.

Definition 4.2. A counterexample π is feasible if a concrete execution τ of the original program can be constructed from π.

To validate a counterexample π, we define a *concrete execution* of a multi-threaded program as follows.

Table 1. Notations for a counterexample π

Notation	Meaning
$s_{0,\pi}$	The initial states of π.
$\mathbb{T}_\pi$	The set of statements contained in π
$\mathbb{E}_\pi$	The set of events occurring in π.
$e_1 \prec_\pi e_2$	e_1 should happen before e_2 in π.
$e_1 \prec^0_\pi e_2$	$e_1 \prec_\pi e_2$ according to the program order.
$e_1 \lhd_\pi e_2$	e_2 reads the value written by e_1.

Definition 4.3. Let s_0 be an initial state of P, $\Lambda := t_0 \cdots t_n$ be a statement sequence, and $s \xrightarrow{t} s'$ indicate that s' is the successor of s by t. The tuple (s_0, Λ) defines a concrete execution of P iff there exists a state sequence $s_0 \cdots s_{n+1}$ s.t. $s_i \xrightarrow{t_i} s_{i+1}$ for all $0 \le i \le n$.

According to Definition 4.3, to validate a counterexample π, one should find a statement sequence of $\mathbb{T}_\pi$ that defines a concrete execution. Note that for a concrete execution τ, the events occurring in τ must satisfy the following order requirements: 1) for each program order (e_1, e_2), we have e_1 happens before e_2; and 2) for each read-write link (e_1, e_2), we have e_1 happens before e_2, and no write of $\mathrm{var}(e_1)$ happens between e_1 and e_2. Given that those local statements do not affect other threads, the crucial issue to construct a concrete execution is to find a total order $<_\pi$ over $\mathbb{E}_\pi$ s.t. $<_\pi$ obeys all the above order requirements. To address this problem, we introduce the *event order graph (EOG)* notion to capture all order requirements of a counterexample.

Definition 4.4. Given a counterexample π, the *event order graph* (EOG) G_π is a triple $\langle \mathbb{E}_\pi, \prec^0_\pi, \lhd_\pi \rangle$, where the nodes are the events in $\mathbb{E}_\pi$, and the edges are the orders defined in $\prec^0_\pi$ and $\lhd_\pi$. Each node corresponds to either a "read" or a "write" of π, and each edge corresponds to either a program order or a read-from order of π. For each edge corresponding to a program order $(e_1, e_2) \in \prec^0_\pi$, it requires that $\mathrm{clk}(e_1) < \mathrm{clk}(e_2)$; and for each edge corresponding to a read-from order $(e_1, e_2) \in \lhd_\pi$, it requires that $\mathrm{clk}(e_1) < \mathrm{clk}(e_2)$ and $\forall e_3 \in \mathbb{E}_\pi, ((\mathrm{var}(e_3) = \mathrm{var}(e_1)) \wedge (\mathrm{type}(e_3) = \mathtt{ww})) \Rightarrow (\mathrm{clk}(e_3) < \mathrm{clk}(e_1) \vee \mathrm{clk}(e_2) < \mathrm{clk}(e_3))$.

Definition 4.5. An EOG G_π is feasible iff there exists a total order $<_\pi$ over $\mathbb{E}_\pi$ s.t. $<_\pi$ obeys all the order requirements defined in G_π.

Theorem 4.6. *A counterexample π is feasible iff the corresponding EOG G_π is feasible.*

Proof. **Sufficiency.** If π is feasible, then we can construct a concrete execution τ from π. Suppose that the statement sequence of τ is Λ. We order all the events in $\mathbb{E}_\pi$ as the execution order of those corresponding global statements in Λ, and obtain a total order $<_\pi$ over $\mathbb{E}_\pi$, which is consistent with $\prec^0_\pi$ and $\lhd_\pi$. Therefore, G_π is feasible.

Necessity. We try to construct a concrete execution τ from π. If G_π is feasible, then there must exist a total order $<_\pi$ over $\mathbb{E}_\pi$ that is consistent with $\prec^0_\pi$ and $\lhd_\pi$. To obtain a statement sequence Λ of $\mathbb{T}_\pi$, we first order all the global statements in $\mathbb{T}_\pi$ as the order of those corresponding events in $<_\pi$, and obtain a total order $<_g$ of all the global statements. Then we "place" the local statements in $\mathbb{T}_\pi$ into $<_g$. Specifically, we first give a total order $t_0 \cdots t_n$ of all local statements according to the program order. Afterward, we insert the local statements into $<_g$ according to this order and obtain a statement sequence Λ. For each local statement t_i, suppose that among all its predecessors of global statements (according to the program order), t^g_i is lastly scheduled in $<_g$, then t_i is scheduled after both t_{i-1} and t^g_i. For instance, if t_{i-1} is scheduled before t^g_i, then t_i is scheduled after t^g_i.

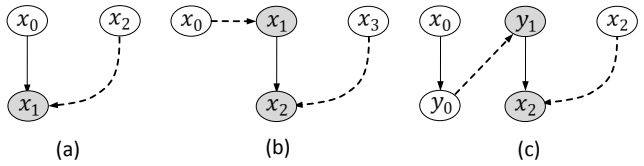


Fig. 4. Three EOG examples

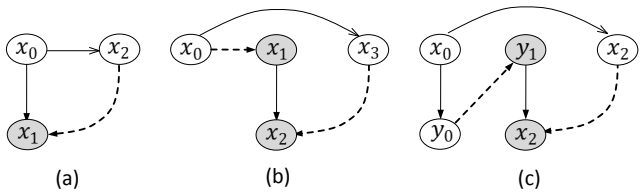


Fig. 5. Derived orders deduced by applying Rule 2 to the three EOGs shown in Fig. 4.

Otherwise, t_i is scheduled after t_{i-1}. Given that $\prec_\pi$ is both consistent with $\prec^0_\pi$ and $\lhd_\pi$, the tuple $(s_{0,\pi}, \Lambda)$ is a concrete execution. Therefore, π is feasible. □

Now we ask, how can we validate the feasibility of an EOG? An intuitive way is to exactly encode all the order requirements defined in Definition 4.4 into a constraint. The EOG is feasible iff the constraint is satisfiable. However, as we will justify later (cf. Section 5 and 7), constraint solving is not effective enough for refinement generation.

4.2 Graph-Based EOG Validation

According to Definition 4.5, any edge (e_1, e_2) of an EOG G_π requires that $e_1 \prec_\pi e_2$. Hence, an EOG must be infeasible if it contains some cycles. Note that a read-from order $(e_1, e_2) \in \lhd_\pi$ further requires that no other write of $\text{var}(e_1)$ could happen between e_1 and e_2. Some implicit order requirements deducible from the EOG must exist. We call them *derived orders* of the EOG. For each derived order (e_1, e_2), it also requires that $\text{clk}(e_1) < \text{clk}(e_2)$, i.e., $e_1 \prec_\pi e_2$.

Consider the EOG shown in Fig. 4(a), where $\{e_{x_0}, e_{x_2}\}$ and $\{e_{x_1}\}$ are the "writes" and "reads" of x, respectively. $e_{x_0} \prec^0_\pi e_{x_1}$, and $e_{x_2} \lhd_\pi e_{x_1}$. We deduce that $e_{x_0} \prec_\pi e_{x_2}$ because $e_{x_0} \prec^0_\pi e_{x_1}$ and $e_{x_2} \lhd_\pi e_{x_1}$, of which the latter implies that no "write" of x can happen between e_{x_2} and e_{x_1}.

Based on this observation, we can deduce as many derived orders as possible first, and add them to $\prec_\pi$. If some cycle exists in $\prec_\pi$, then the EOG must be infeasible. To this end, we propose the following three rules to produce derived orders. We initially obtain $\prec_\pi := \prec^0_\pi \cup \lhd_\pi$.

Rule 1. $$\frac{e_1 \prec_\pi e_2,\ e_2 \prec_\pi e_3}{e_1 \prec_\pi e_3}.$$

Rule 1 only reflects the transitivity of $\prec_\pi$.

Rule 2. $$\frac{e_1 \lhd_\pi e_2,\ e_3 \prec_\pi e_2,\ \text{type}(e_3) = \texttt{ww},\ \text{var}(e_3) = \text{var}(e_1)}{e_3 \prec_\pi e_1}.$$

Given that $e_1 \lhd_\pi e_2$, $\text{type}(e_3) = \texttt{ww}$, and $\text{var}(e_3) = \text{var}(e_2)$, then either $\text{clk}(e_3) < \text{clk}(e_1)$ or $\text{clk}(e_2) < \text{clk}(e_3)$. Given that $e_3 \prec_\pi e_2$ holds, then $\text{clk}(e_3) < \text{clk}(e_2)$ can be obtained. Therefore, we have $\text{clk}(e_3) < \text{clk}(e_1)$, which implies $e_3 \prec_\pi e_1$.

Similarly, we propose the following deductive rule:

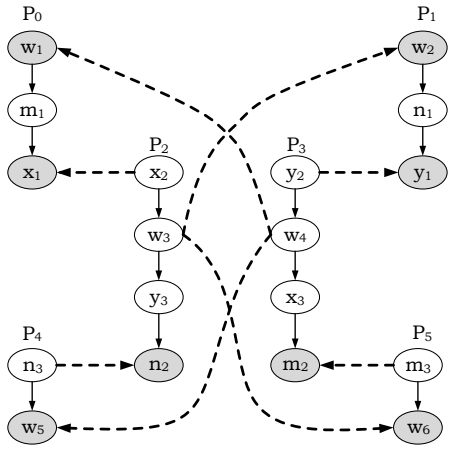


Fig. 6. The "butterfly" example

$$\textsc{Rule 3.}\ \frac{e_1 \lhd_\pi e_2,\ e_1 \prec_\pi e_3,\ \mathrm{type}(e_3) = \mathtt{ww},\ \mathrm{var}(e_3) = \mathrm{var}(e_1)}{e_2 \prec_\pi e_3}.$$

According to these three rules, for the three EOGs presented in Fig. 4, we can deduce $e_{x_0} \prec_\pi e_{x_2}$, $e_{x_0} \prec_\pi e_{x_3}$, and $e_{x_0} \prec_\pi e_{x_2}$, respectively. We add these terms to the corresponding EOGs as shown in Fig. 5.

When a derived order (e_1, e_2) is added to $\prec_\pi$, some new orders may be propagated via Rules 1 to 3. We repeat this process until we reach a *fixpoint*, i.e., no derived order can be deduced any more. If some cycle exists in $\prec_\pi$, then the EOG is infeasible.

Now a conjecture is that, an EOG is feasible if it contains no cycle. Such a conjecture holds for almost all examples in our experiments. However, this conjecture may still be false for some special examples. Consider the "butterfly" example in Fig. 6 that involves six threads $\{P_0, P_1, \cdots, P_5\}$ and five shared variables $\{m, n, x, y, w\}$. No derived order can be deduced, and no cycle exists in the EOG. However, no total order of $\mathbb{E}_\pi$ can satisfy all the order requirements defined in the EOG, and the EOG is infeasible.

Our graph-based EOG validation method is shown in Algorithm 1. It repeatedly applies Rules 1 to 3 to deduce new derived orders, until a fixpoint is reached. If there exists some conflict event e s.t. $e \prec_\pi e$, then the EOG must be infeasible. Otherwise, it is not sure whether the EOG is feasible or not.

We now prove the correctness of this algorithm.

Theorem 4.7. *If Algorithm 1 concludes that G_π is infeasible, then G_π must be infeasible.*

Proof. If Algorithm 1 concludes that G_π is infeasible, then there must exist some conflict event e s.t. $e \prec_\pi e$. Suppose that G_π is feasible. Then according to Definition 4.5, there must exist a total order $\prec_\pi$ s.t. $\prec_\pi$ obeys all order requirements defined in G_π. According to Rules 1 to 3, we have $\prec_\pi$ also obeys all the derived orders deduced in Algorithm 1. Hence, $\mathrm{clk}(e) < clk(e)$ in $\prec_\pi$, which is impossible for a total order.

Therefore, the theorem is proved. □

Input: An EOG $G_\pi = \langle \mathbb{E}_\pi, \prec_\pi^0, \lhd_\pi \rangle$
Output: Infeasible: G_π is infeasible; Not-Sure: not sure whether G_π is feasible or not
repeat
  if there exist $(e_1, e_2), (e_2, e_3) \in \prec_\pi$ and $(e_1, e_3) \notin \prec_\pi$ **then**
    $\prec_\pi = \prec_\pi \cup (e_1, e_3)$;
  end
  if there exists $(e_1, e_2) \in \lhd_\pi$ exists $(e_3, e_2) \in \prec_\pi$ and $(e_3, e_1) \notin \prec_\pi$ **then**
    if type(e_3) = ww and var(e_3) = var(e_1) **then**
      $\prec_\pi := \prec_\pi \cup (e_3, e_1)$;
    end
  end
  if there exists $(e_1, e_2) \in \lhd_\pi$ exists $(e_1, e_3) \in \prec_\pi$ and $(e_2, e_3) \notin \prec_\pi$ **then**
    if type(e_3) = ww and var(e_3) = var(e_1) **then**
      $\prec_\pi := \prec_\pi \cup (e_2, e_3)$;
    end
  end
until $\prec_\pi$ reaches a fixpoint;
if $e \prec_\pi e$ for some e **then**
  return Infeasible;
end
return *Not-Sure*;

Algorithm 1: Graph-based EOG validation

4.3 Constraint-Based EOG Validation

If the graph-based EOG validation method is not sure whether an EOG is feasible or not, we employ a constraint solver to further determine feasibility of the EOG. The method is that, we exactly encode all the order requirements defined in Definition 4.4 into a constraint. The EOG is feasible iff the constraint is satisfiable. If a SAT is returned, then it generates a total order of all the events which satisfies all the order requirements of the EOG as a byproduct. Otherwise, both the EOG and the counterexample are infeasible and an unsatisfiable core is generated as a byproduct.

5 KERNEL REASON BASED REFINEMENT GENERATION

If a counterexample is determined to be infeasible, one should add some constraints to the abstraction to prevent this counterexample from appearing again in the future search. The most intuitive way to address this problem is to add the negation of this counterexample to the abstraction. However, such a manner excludes only one counterexample in each refinement. To prune more search space, a better idea is to analyze the kernel reasons that make the counterexample infeasible, and then adds the negation of these kernel reasons to the next abstraction. Given that a counterexample is validated via validating its corresponding EOG, we analyze the kernel reasons that make an EOG infeasible, s.t. a large amount of space can be pruned in each refinement iteration.

5.1 Representation of Kernel Reasons

Given a counterexample π, the corresponding EOG $G_\pi := \langle \mathbb{E}_\pi, \prec_\pi^0, \lhd_\pi \rangle$ is determined by the values of the following two kinds of literals: 1) The values of those *guard condition literals* for the events in $\mathbb{E}$. They determine both $\mathbb{E}_\pi$ and $\prec_\pi^0$. $\mathbb{E}_\pi$ is the set of events appeared in π. $\prec_\pi^0$ is the restriction of $\prec_P^0$ to $\mathbb{E}_\pi$. 2) The values of those *read-write link literals* which define the read-from relationship $\lhd_\pi$.

Let $\mathbb{G}_\pi$ and $\mathbb{S}_\pi$ denote the sets of true guard condition literals and true read-write link literals in π, respectively. If G_π is infeasible, then the infeasibility could be deduced by the conjunction of all literals in $\mathbb{G}_\pi \cup \mathbb{S}_\pi$. The infeasibility can often be deduced by a subset of $\mathbb{G}_\pi \cup \mathbb{S}_\pi$. Therefore, the *kernel reasons* that make a counterexample π infeasible could be represented by the minimal subsets of $\mathbb{G}_\pi \cup \mathbb{S}_\pi$ that could deduce the infeasibility. Finding the minimal subsets not only reduces the constraint size but also prunes more search space.

5.2 Constraint-Based Kernel Reason Analysis

If the infeasibility is identified by the constraint-based EOG validation process, the constraint solver can return an unsatisfiable core as a byproduct. Modern constraint solvers, such as MiniSat2, allow their users to take a set of literals as assumptions. When an UNSAT is returned, the constraint solver generates a subset of the assumption literals as an unsatisfiable core. Based on this idea, we take all literals in $\mathbb{G}_\pi \cup \mathbb{S}_\pi$ as assumption literals. Whenever the EOG is determined to be infeasible, the constraint solver generates a subset of $\mathbb{G}_\pi \cup \mathbb{S}_\pi$ that can still deduce the infeasibility. Suppose that it is $\{\ell_1, \ell_2, \cdots, \ell_n\}$. Let $\varpi := \ell_1 \wedge \ell_2 \wedge \cdots \wedge \ell_n$. The refinement constraint can then be formulated as follows.

$$\kappa := \neg\varpi \tag{2}$$

If the constraint that exactly encodes all order requirements of an EOG is unsatisfiable, it may have a large number of unsatisfiable cores. To prune as much search space as possible, one should obtain all of them. However, generating all unsatisfiable cores is usually time consuming. Most constraint solvers generate only one unsatisfiable core each time, and it may not be the shortest one, which significantly limits the pruned search space in each refinement. Hence, constraint solving is not a good choice for our refinement generation.

5.3 Graph-Based Kernel Reason Analysis

This section presents our graph-based kernel reason analysis method. Compared with the constraint-based method, it usually obtains a much more effective refinement. It can efficiently obtain all kernel reasons that make an EOG infeasible. In addition, the obtained "core kernel reasons" are always the shortest ones. Hence, it can usually prune much more search space in each iteration.

In Algorithm 1, if an infeasibility is determined, then there must exist some conflict event e s.t. $e \prec_\pi e$. A conflict event can usually be attributed to several kernel reasons, and there are usually many conflict events. To prune more search space, one should find all kernel reasons of all conflict events.

We define "a kernel reason of an order $\lambda \in \prec_\pi$" to be "the minimal subset of $\mathbb{G}_\pi \cup \mathbb{S}_\pi$ that can deduce λ". When a derived order λ is deduced, if the kernel reasons of all existing orders (including existing derived orders) are given, then we can obtain a set of kernel reasons of λ upon its production. Note that a derived order λ may be deduced for multiple times. Whenever it is deduced, we can obtain new kernel reasons of λ. Based on this observation, we initialize the kernel reasons of every order λ to $\emptyset$. The kernel reasons of an order λ are then updated once λ is added (we add the orders in $\prec_\pi^0$ and $\lhd_\pi$ into the graph one by one) or deduced. In this manner, we obtain the kernel reasons of each order $\lambda \in \prec_\pi$ when Algorithm 1 terminates.

We then discuss how to update the kernel reasons of an order $\lambda \in \prec_\pi$ when λ is added or deduced. We hypothesize that the kernel reasons of all existing orders are given. We denote by $o(\lambda)$ and $\mathbb{O}(\lambda)$ a kernel reason and the set of kernel reasons of λ, respectively.

- If $\lambda := (e_1, e_2)$ is added because $\lambda \in \prec_\pi^0$, then $e_1 \prec_\pi e_2$ iff both $e_1, e_2 \in \mathbb{E}_\pi$, i.e., the guard condition literals for both e_1 and e_2 are true. Therefore, $\mathbb{O}(\lambda) := \mathbb{O}(\lambda) \cup \{\{\text{guard}(e_1), \text{guard}(e_2)\}\}$, where guard($e$) is the guard condition literal of e.
- If $\lambda := (e_1, e_2)$ is added because $\lambda \in \lhd_\pi$, λ holds iff e_2 reads the value written by e_1, which already indicates that both guard(e_1) and guard(e_2) are true. Therefore, $\mathbb{O}(\lambda) := \mathbb{O}(\lambda) \cup \{\{\text{sel}(\lambda)\}\}$, where sel($\lambda$) is the read-write link literal of λ.
- If λ is a derived order deduced from λ_1 and λ_2, then $\lambda \in \prec_\pi$ iff both λ_1 and λ_2 belong to $\prec_\pi$. Therefore, $\mathbb{O}(\lambda) := \mathbb{O}(\lambda) \cup \{o_1 \cup o_2 \mid o_i \in \mathbb{O}(\lambda_i)\}$. Note that λ may be deduced for multiple times. We incrementally update $\mathbb{O}(\lambda)$ whenever λ is deduced.

A kernel reason $o \in \mathbb{O}(\lambda)$ is considered *redundant* if some kernel reason $o' \in \mathbb{O}(\lambda)$ exists s.t. $o' \subseteq o$. Such kernel reasons are immediately eliminated from $\mathbb{O}(\lambda)$, and the remaining reasons are called *core kernel reasons* of λ. Using this strategy, we maintain only the set of core kernel reasons in our algorithm. This set is dynamically updated during the running of the algorithm.

In this manner, we obtain the set of core kernel reasons of any order $\lambda \in \prec_\pi$ when Algorithm 1 terminates. Let $\mathbb{A}$ denote the set of conflict events. The kernel reasons that make π infeasible (denoted by $\mathbb{O}(\pi)$) can be expressed as follows.

$$\mathbb{O}(\pi) := \bigcup_{e \in \mathbb{A}} \mathbb{O}(e \prec_\pi e) \tag{3}$$

Again, we eliminate those redundant kernel reasons from $\mathbb{O}(\pi)$, and maintain only those core kernel reasons. In our experiments, hundreds or even thousands of kernel reasons may be observed, but only several or dozens of them are considered core ones.

Suppose that $o := \{\ell_1, \ell_2, \cdots, \ell_m\}$ where each ℓ_i is a literal, we define the following:

$$\varpi_o := \ell_1 \wedge \ell_2 \wedge \cdots \wedge \ell_m \tag{4}$$

Suppose that $\mathbb{O}(\pi) := \{o_1, o_2, \cdots, o_n\}$, then the refinement constraint is formulated as follows.

$$\kappa := \bigwedge_{i=1}^{n} \neg \varpi_{o_i} \tag{5}$$

Algorithm 2 demonstrates our graph-based refinement generation method. We first add all the program orders into $\prec_\pi$, and update their kernel reasons according to the kernel reason updating method we have discussed. Adding a read-from order or a derived order to $\prec_\pi$ may propagate a large number of new orders. Whenever a read-from order $\lambda \in \lhd_\pi$ is added to $\prec_\pi$, we denote by $\mathbb{D}$ the set of orders that will be added to $\prec_\pi$ before another read-from order is added. Adding each order $\lambda' \in \mathbb{D}$ to $\prec_\pi$ may propagate a set of derived orders, which are denoted by $\mathbb{B}$. Whenever an order $\lambda'' \in \mathbb{B}$ is deduced, we update its kernel reasons and add it to $\mathbb{D}$ if it is not contained in $\prec_\pi$. In this manner, once all read-from orders have been added to $\prec_\pi$, we obtain all derived orders and the kernel reasons of all orders in $\prec_\pi$. We then compute the refinement constraint according to equation (3), (4) and (5).

From the above discussion, the graph-based refinement generation method has the following advantages:

(1) It can detect all kernel reasons that make π infeasible, leading to a large amount of search space pruned in each iteration.
(2) It maintains a minimal subset of core kernel reasons, thereby making the refinement constraint small and manageable.

Input: An EOG $G_\pi := \langle \mathbb{E}_\pi, \prec_\pi^0, \lhd_\pi \rangle$.
Output: A refinement constraint κ.
$\prec_\pi := \prec_\pi^0$, and update $\mathbb{O}(\lambda)$ for each $\lambda \in \prec_\pi^0$;
foreach $\lambda \in \lhd_\pi$ **do**
  Update $\mathbb{O}(\lambda)$, and let $\mathbb{D} := \{\lambda\}$;
  foreach $\lambda' \in \mathbb{D}$ **do**
    $\prec_\pi := \prec_\pi \cup \{\lambda'\}$;
    Let $\mathbb{B}$ be the set of propagated orders due to λ';
    foreach $\lambda'' \in \mathbb{B}$ **do**
      Update $\mathbb{O}(\lambda'')$;
      if $\lambda'' \notin \prec_\pi$ **then**
        $\mathbb{D} := \mathbb{D} \cup \{\lambda''\}$;
      end
    end
  end
end
Compute κ according to equation (3), (4) and (5);
return κ;

Algorithm 2: Graph-based refinement generation.

5.4 Correctness of the Graph-Based Kernel Reason Analysis

We prove the correctness of our graph-based refinement generation method, that is, the refinement constraint obtained in equation (5) should be true in any feasible counterexample, i.e., it will not eliminate any feasible counterexample from the abstraction.

Theorem 5.1. *Given a counterexample π and a kernel reason $o(\lambda)$ obtained according to our graph-based kernel reason analysis method, we have $o(\lambda) \models \lambda$. That is, for any other counterexample π', we also have $\lambda \in \prec_{\pi'}$ if $\pi' \models \varpi_{o(\lambda)}$.*

Proof. We prove this theorem by induction.

Inductive Base: If $o(\lambda)$ is obtained because $\lambda \in \prec_\pi^0$ or $\lambda \in \lhd_\pi$, then the conclusion can be immediately inferred from the definition.

Inductive Step: If $o(\lambda)$ is obtained via Rule 1, 2, or 3, then two orders λ_1 and λ_2 must exist, such that $o(\lambda) = o(\lambda_1) \cup o(\lambda_2)$. Given that $\varpi_{o(\lambda)}$ holds w.r.t. π', $\varpi_{o(\lambda_1)}$ and $\varpi_{o(\lambda_2)}$ are also true w.r.t. π'. By applying the induction hypothesis, we obtain $\lambda_1, \lambda_2 \in \prec_{\pi'}$. According to the same rule, we deduce that $\lambda \in \prec_{\pi'}$. □

Theorem 5.2. *Given a kernel reason $o \in \mathbb{O}(\pi)$ of an infeasible counterexample π, for any feasible counterexample π', we have $\pi' \models \neg\varpi_o$.*

Proof. Given that o is a kernel reason that makes π infeasible, according to equation (3), there must exist a corresponding order $\lambda := (e, e) \in \prec_\pi$. Suppose that $\pi' \models \varpi_o$. Then according to Theorem 5.1, we have $\lambda := (e, e) \in \prec_{\pi'}$. It indicates that π' is infeasible, which is contradict with that π' is feasible. Hence, we must have $\pi' \models \neg\varpi_o$. □

Theorem 5.3. *Given a refinement constraint κ obtained in some iteration, for any feasible counterexample π', we have $\pi' \models \kappa$.*

Proof. Suppose that $\kappa := \bigwedge_{i=1}^{n} \neg\varpi_{o_i}$, and the corresponding counterexample is π. According to Theorem 5.2, for any i ($1 \leq i \leq n$), we have $\pi' \models \neg\varpi_{o_i}$. Hence, $\pi' \models \kappa$. □

6 SOUNDNESS AND EFFICIENCY

6.1 Soundness and Completeness

We prove the soundness and completeness of our method (shown in Fig. 1) via three theorems.

Theorem 6.1. *If our method concludes that the property is safe, then it must be safe w.r.t. the given loop unwinding depth.*

Proof. To prove this theorem, we prove that $\alpha \wedge \phi_{err} \models \varphi_0 \wedge \wedge_{i=0}^{n}\kappa_i \wedge \phi_{err}$, where α is the monolithic encoding, φ_0 is the initial abstraction, κ_i is the i-th refinement constraint, and ϕ_{err} is the error states. According to the definition of α and φ_0 (cf. Definition 3.1), we can obtain $\alpha \wedge \phi_{err} \models \varphi_0 \wedge \phi_{err}$. We prove that $\alpha \wedge \phi_{err} \models \kappa_i$ as follows.

If κ_i is obtained from the graph-based refinement process, we prove that for each element $\neg\varpi \in \kappa_i$, $\alpha \wedge \phi_{err} \models \neg\varpi$. Suppose that $\alpha \wedge \phi_{err} \not\models \neg\varpi$. Then there must exist an assignment π of $\alpha \wedge \phi_{err}$ s.t. ϖ holds. Given that π is a feasible counterexample, according to Theorem 5.2, $\pi \models \neg\varpi$, which is contradict with that ϖ holds in π. Hence, we must have $\alpha \wedge \phi_{err} \models \neg\varpi$ and $\alpha \wedge \phi_{err} \models \kappa_i$.

If κ_i is obtained from the constraint-based refinement process, then $\kappa_i = \neg\varpi$ where ϖ is an unsatisfiable core of the formula which encodes all order requirements of an EOG. Suppose that $\alpha \wedge \phi_{err} \not\models \neg\varpi$. Then there must exist an assignment π of $\alpha \wedge \phi_{err}$ s.t. ϖ holds, which is contradict with that ϖ is an unsatisfiable core. Hence, we must have $\alpha \wedge \phi_{err} \models \kappa_i$. □

Theorem 6.2. *If our method concludes that the property is unsafe, then a true counterexample of the property must exist.*

Proof. In our method, the property is concluded unsafe only if the constraint-based EOG validation process returns SAT. Given that the formula in the constraint-based EOG validation process encodes all order requirements of the EOG exactly, according to Definition 4.4 and 4.5, the EOG is feasible iff the formula is satisfiable. Hence, the EOG is feasible. According to Theorem 4.6, the corresponding counterexample must be feasible. Therefore, a true counterexample of the property must exist. □

Theorem 6.3. *Our method will terminate for any program with finite state space.*

Proof. For a multi-threaded program with finite state space, the number of counterexamples of the initial abstraction must be finite. Suppose that the counterexample obtained in the i-th iteration is π_i. According to Section 5, each kernel reason of the counterexample or the unsatisfiable core obtained in the constraint-based refinement process is just a subset of $\mathbb{G}_{\pi_i} \cup \mathbb{S}_{\pi_i}$. According to equation (5) and (2), we have π_i must be absent in the next abstraction. Hence, our method reduces at least one counterexample in each iteration. It terminates when all counterexamples have been reduced or a true counterexample is found. □

6.2 Efficiency

A both efficient and sound way for counterexample validation and refinement generation will be elegant. However, such procedure is usually difficult to devise. As an alternative, we integrate the graph-analysis and constraint solving approaches together to obtain a both efficient and sound method.

We have proved that enhanced by the constraint-based counterexample validation and refinement generation processes, our method is sound and complete. We now analyze the effectiveness of the

graph-based EOG validation method. If the EOG is infeasible, the infeasibility may be determined by either the graph-based EOG validation process or the constraint-based EOG validation process. Although both of these processes can rapidly determine such infeasibility, a much more effective refinement can be obtained if the infeasibility is determined from the graph-based EOG validation process. Fortunately, cases similar to the "butterfly" example rarely occur in practice. In other words, the graph-based EOG validation process can always identify the infeasibility with rare exceptions.

Suppose that the verification problem is solved via n iterations. If the property is determined to be unsafe, then all EOGs generated during the first $n-1$ rounds must be infeasible, which can generally be identified by the graph-based EOG validation process. The constraint-based EOG validation process is only invoked in the last iteration during which the property is violated. If the property is proved safe w.r.t. the given loop unwinding depth, then the infeasibility of all the n infeasible EOGs can generally be identified by the graph-based EOG validation process, and the constraint-based EOG validation process will not be invoked.

In sum, advantages of our method include: 1) Without the scheduling constraint, the initial abstraction φ_0 is usually much smaller than the monolithic encoding. 2) The graph-based refinement process can usually obtain a small yet effective refinement, which reduces a large amount of space in each iteration whereas the size of all those refinement constraints can usually be ignored compared with that of the abstraction. 3) Though the graph-based validation process is not complete, it is effective to identify the infeasibility in practice.

7 EXPERIMENTAL RESULTS

We have implemented our method on top of CBMC-4.9 [3] and employed MiniSat2 as the back-end constraint solver. Our tool is named Yogar-CBMC, and it is available at [36]. We use the 1047 multi-threaded programs of SV-COMP 2017 [36] as our benchmarks. In the experiments, our tool supports nearly all features of C language and PThreads.

7.1 Benchmark of SV-COMP 2017

The open-source, representative, and reproducible benchmarks of Competition on Software Verification (SV-COMP) have been widely accepted for program verification. Given that these benchmarks are devised for comparison of those state-of-the-art techniques and tools, a significant number of studies on concurrent program verification have performed their experiments on them.

The concurrency benchmarks of SV-COMP 2017 include 1047 examples and cover most of the publicly available concurrent C programs that are used for verification. Though many of these examples are small in size in previous years, dozens of complex examples have been added to these benchmarks in recent years. For instance, the examples in the `pthread-complex` directory (collected by the CSeq team) are taken from the papers on PLDIą́15 [28], POPLą́15 [8], and PPOPPą́14 [37], which are used for concurrent program debugging and testing; the examples in the `pthread-driver-races` directory (collected by the SMACK+CORRAL team) are used for symbolic analysis of the drivers from the Linux 4.0 kernel [13]; and the examples in the `pthread-C-DAC` directory (collected by C-DAC) are from the industrial problems of Centre for Development of Advanced Computing, Pune, India. These programs contain hundreds of lines, 4 to 8 threads, complex structure variables with 2D pointers, and hundreds or even a thousand read/write accesses [4]. Given these complex features, these programs are challenging for existing state-of-the-art concurrency verification techniques and tools.

[3] Downloaded from https://github.com/diffblue/cbmc/releases on Nov 20, 2015

[4] A read/write access of a complex structure variable may contain hundreds of read/write accesses of boolean variables. Here a read/write access of a complex structure variable is considered just one read/write access.

7.2 Experimental Setup

We conduct all of our experiments using a computer with Intel(R) Core(TM) i5-4210M CPU 2.60 GHz and 12 GB memory. A 900-second time limit is observed.

We select and compare two classes of the state-of-the-art tools with our method. The first class is those top winners in recent competitions, including MU-CSeq[5] [39] and Lazy-CSeq-Abs[5] [30]. MU-CSeq is the gold medal winner of SV-COMP 2016, and Lazy-CSeq-Abs is the silver medal winner of SV-COMP 2017. The second class comprises those tools which methods are closely related to ours, including CBMC [3] and Threader[6] [31]. CBMC is a highly popular verifier for program verification. Different from our method, it provides an exact encoding of the scheduling constraint for multi-threaded program verification. For Threader, to the best of our knowledge, it is the best CEGAR-based verifier for multi-threaded C programs. It has received the gold medal of the concurrency track of SV-COMP 2013.

Given that different tools employ different techniques, each of which has its own features, it is difficult to make the comparison absolutely fair. For example, Threader performs unbounded verification, while all the other tools perform BMC; and given a loop unwinding depth, both MU-CSeq and Lazy-CSeq-Abs are incomplete whereas all the other tools are complete. To make the comparison as fair as possible, we select the latest available version of them, and set the parameters of them to be that of the competition. We believe that these tools should perform best under these parameters. For the loop unwinding depth, we set that of CBMC and Yogar-CBMC the same as that of MU-CSeq. Specifically, it is dynamically determined through syntax analysis. The bound is set to 2 for programs with arrays, and n if some of the program's `for`-loops are upper bounded by a constant n [39]. Given that our method is implemented on top of CBMC. The only difference between CBMC and Yogar-CBMC is that CBMC employs the monolithic encoding while we perform our abstraction refinement on the scheduling constraint.

7.3 Effectiveness and Efficiency

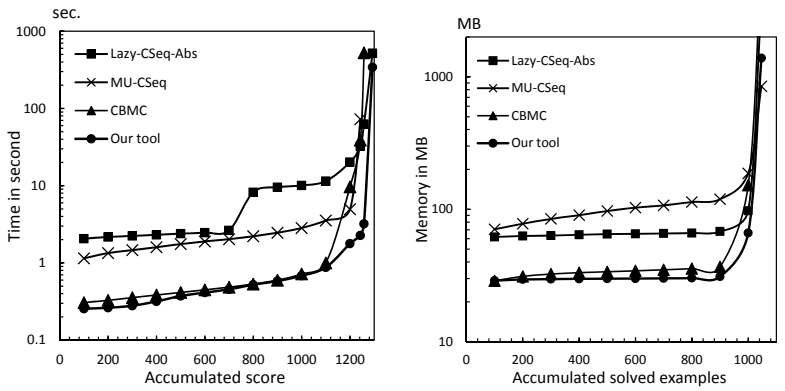
Yogar-CBMC solved[7] all the 1047 concurrent programs and has received the highest score of 1293 points. It has won the gold medal in the concurrency track of SV-COMP 2017 [36] (Warning: It will violate our anonymity).

Overall comparison with state-of-the-art tools. Fig. 7 compares our tool with the state-of-the-art tools, including MU-CSeq, Lazy-CSeq-Abs, and CBMC. Similar to SV-COMP 2017, we perform our experiments on BenchExec [8] to achieve a reliable and repeatable benchmarking.

The experimental results for Lazy-CSeq-Abs and MU-CSeq are consistent with those in SV-COMP 2017. However, the results in our experiments for CBMC is better than those in SV-COMP 2017. The reason is that we have improved CBMC in several aspects, and we have also realized some concurrency-related improvements in CBMC-5.5 [9].

Fig. 7(a) compares the overall performance of Lazy-CSeq-Abs, MU-CSeq, CBMC, and Yogar-CBMC based on the SV-COMP rules. The x-axis represents the accumulated score, while the y-axis represents the time needed to achieve a certain score [10]. Both our tool and Lazy-CSeq-Abs have successfully solved all examples and obtained 1293 points, while MU-CSeq and CBMC obtained

[5]Downloaded from http://sv-comp.sosy-lab.org/2017/systems.php on January 24, 2017
[6]Downloaded from http://sv-comp.sosy-lab.org/2014/participants.php on January 24, 2017
[7]*Solve* means that the verifier gives a correct answer (true/false) within the time limit. Refer to [36] for the rules of the competition.
[8]https://github.com/sosy-lab/benchexec
[9]Released on August 20, 2016 in https://github.com/diffblue/cbmc/releases
[10]The rules to assign the score can be found in http://sv-comp.sosy-lab.org/2017/rules.php



(a) Time comparison (b) Memory comparison

Fig. 7. Compare with state-of-the-art tools

1243 and 1258 points, respectively. Lazy-CSeq-Abs, MU-CSeq, and CBMC spent 9820, 2540, and 12300 s to finish all examples, respectively, while our tool completed all examples within 1550 s.

Fig. 7(b) shows the overall memory consumption of the aforementioned tools. Lazy-CSeq-Abs, MU-CSeq, CBMC, and Yogar-CBMC require 104, 103, 84 and 43 GB to solve all 1047 examples, respectively. Given that the scheduling constraint is ignored in the abstraction, our tool always solves small problems and consumes much less memory than the three other tools.

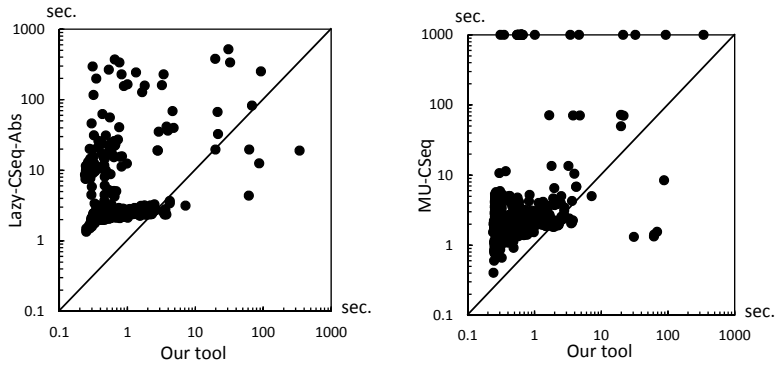
We further compare our tool with Lazy-CSeq-Abs, MU-CSeq, CBMC, and Threader to evaluate its performance.

Yogar-CBMC versus Lazy-CSeq-Abs. Compared with Lazy-CSeq-Abs, our tool runs 6.34 times faster on average, and consumes only 41% of the memory over all 1047 examples. As shown in Fig. 8(a), Lazy-CSeq-Abs outperforms our tool in only 12 of the 1047 examples. With its *abstract interpretation* technique, Lazy-CSeq-Abs outperforms our tool in those examples where the numerical analysis dominates the complexity.

Yogar-CBMC versus MU-CSeq. MU-CSeq fails to solve 30 of the 1047 examples. Compared with this tool, our tool runs 2.43 times faster on average and consumes only 36% of the memory for the remaining 1017 examples. As shown in Fig. 8(b), MU-CSeq outperforms our tool in only 33 of the 1047 examples. By applying the *memory unwinding* technique to limit the number of writes, the encoding size of MU-CSeq is insensitive to the scale of the data structures, thereby outperforming our tool for some special examples.

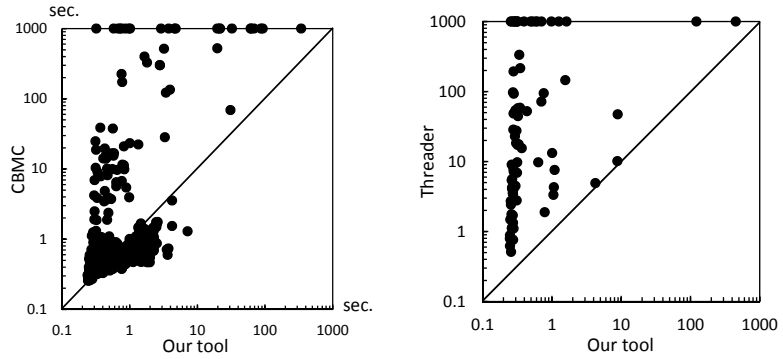
Yogar-CBMC versus CBMC. Fig. 9(a) compares CBMC with our tool. CBMC fails to solve 23 of the 1047 examples. Both CBMC and our tool can easily solve 92% of these examples. For these trivial examples, the monolithic encoding may sometimes run faster. However, our tool outperforms CBMC by 35.8 times on average in the 56 complex cases in which CBMC needs more than two seconds to solve them.

Yogar-CBMC versus Threader. Threader only participated in SV-COMP 2013 and 2014. Given that this tool cannot solve many of the examples in SV-COMP 2017, we compare it with our tool on the benchmarks of SV-COMP 2014, which contain only 78 examples. Fig. 9(b) presents the results. Our tool has completed all 78 examples within the time limit, while Threader completed only



(a) Our tool versus Lazy-CSeq-Abs (b) Our tool versus MU-CSeq

Fig. 8. Compare with Lazy-CSeq-Abs and MU-CSeq



(a) Our tool versus CBMC (b) Our tool versus Threader

Fig. 9. Compare with CBMC and Threader

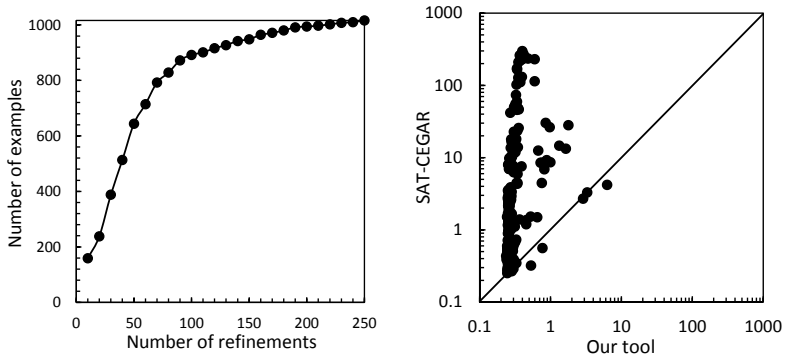
59 examples. Moreover, our tool and Threader require 140 s and 6865 s to solve these examples respectively, thereby showing that our tool is 49 times faster than Threader on average. However, as we have declared before, Threader performs unbounded verification, while we perform BMC.

7.4 Essence Analysis

The performance of our tool is mainly affected by the number of refinements, size of constraints and cost of constraint solving, etc. We also justify the benefits from the graph-based refinement generation method.

Number of refinements. Fig. 10(a) presents the number of refinements of our tool in the experiments. The point (50, 644) indicates that 644 examples can be solved in less than 50 refinements. Fig. 10(a) shows that most of the examples can be solved in less than 80 refinements. In our method, we successfully decomposed the complex verification problem into dozens of small problems.

Size of refinements. Our experimental results reveal that without the scheduling constraint, the formula size reduces to 1/8 on average and to 1/1200 in the extreme case, thereby allowing for the



(a) Number of refinements (b) Our tool versus SAT-CEGAR

Fig. 10. Number of refinements and benefits from EOG

abstractions to be solved quickly. However, the number of clauses increased during the refinement process can usually be ignored. Most of the examples in our experiments show hundreds or even thousands of increase in the number of CNF clauses during the refinements. However, the CNF clause number of the abstraction may reach millions.

Cost of constraint solving. Without the scheduling constraint, the abstractions can usually be solved instantly. Meanwhile, the graph-based EOG validation and refinement generation processes are not trivial. We have observed that in our experiments, our tool has spent most of its time on graph analysis for those examples where the scheduling constraint dominates the encoding. Meanwhile, for those examples where the complexity mainly stems from the complex data structures and numerical calculation, our tool has spent most of its time on constraint solving.

Efficiency benefit from the graph-based refinement generation. The efficiency of our method mainly benefits from the graph-based refinement generation. We have implemented another scheduling constraint based abstraction refinement method, SAT-CEGAR, which employs only *constraint solving* for EOG validation and refinement generation. SAT-CEGAR has solved only 265 of the 1047 examples in the experiments. Fig. 10(b) compares the performance of our tool with SAT-CEGAR in solving these examples. Our tool outperforms SAT-CEGAR for most examples, and runs 58 times faster on average. On average, our tool finds 9.3 core kernel reasons in each refinement, with each kernel reason having an average clause length of 3.06. Meanwhile, SAT-CEGAR only finds one kernel reason in each refinement, with each reason having an average clause length of 7.5.

Exceptions where the graph-based EOG validation method fails. In our method, if the constraint solving based refinement process is invoked and returns UNSAT, then we cannot achieve an effective refinement. How often does this case occur in the experiments? Fortunately, we have not observed cases similar to the "butterfly" example in our experiments, and the constraint solving based refinement process has never been invoked.

7.5 Threats to Validity

One threat to the validity is the limited benchmarks we have used. For those examples where the scheduling constraint is not a major part of the encoding, our method may still need dozens of refinements. Given that those abstractions may have similar size with the monolithic encoding, our method may perform worse than the monolithic method.

Another threat to the validity is the tools we have used. Given that different technologies have different advantages, it is difficult to give an absolute fair comparison. In some other scenarios, one may prefer Threader and other tools.

The third threat to the validity is the parameters we have used for each tool. Most tools have some parameters related with their techniques, such as the loop unwinding depth. Given that different tools may have different parameters, it is difficult to compare all the tools under the same parameters.

8 RELATED WORK

Addressing the control state explosion resulting from concurrency poses a significant challenge to concurrent program verification. Several techniques have recently been studied to overcome this problem, including stateless model checking [1, 4, 9, 24], compositional reasoning [14, 22, 29, 32], bounded model checking [3, 10, 20, 25, 38, 43], and abstraction refinement [11, 12, 21, 29, 35], etc.

The general idea of stateless model checking is to employ partial order reduction (POR) or dynamic partial order reduction (DPOR) [1, 4, 9, 41] to explore only non-redundant interleavings. There are also some work which reduces the search space by restricting the schedules of the program [5, 40]. In compositional reasoning, rather than considering all possible interleavings of a program, the property is decomposed into different components. Each component is then considered in isolation, without any knowledge of the precise concurrent context. Recent work on compositional reasoning includes assume-guarantee reasoning [15, 16], rely-guarantee reasoning [19, 22, 27], thread-modular reasoning [29], and compositional reasoning [14, 32], etc.

Our scheduling constraint based abstraction refinement method explores both bounded model checking and abstraction refinement. Afterward, we compare our study with the recent work on these two methods.

On Bounded Model Checking. Bounded model checking has been considered an efficient technique to address the interleaving problem. In SV-COMP 2017, 16 out of the 18 participants in the concurrency track have adopted this technique [6]. However, pure BMC is still not efficient enough. Many existing tools combine this method with other techniques. ESBMC combines symbolic model checking with explicit state space exploration [10]. VVT employs a CTIGAR method, an SMT-based IC3 algorithm that incorporates CEGAR [20]. The interleaving problem can also be addressed by translating the concurrent programs into sequential programs. Tools implementing this technique include MU-CSeq [38], Lazy-CSeq-Abs [25], and SMACK[34], etc.

However, all of these work gives an exact encoding of the scheduling constraint, while we ignore this constraint and employ a scheduling constraint based abstraction refinement method to obtain a small yet effective abstraction w.r.t. the property.

On Abstraction Refinement. Abstraction refinement has been widely studied in concurrent program verification. Most of these work employs predicate abstraction to address the data space explosion problem [11, 12, 21–23, 42]. In predicate abstraction, it uses a finite number of predicates to abstract the program. If an abstraction counterexample is spurious, it finds predicates that add more details of the program to refine the abstraction, s.t. the spurious counterexample is absent in the latter abstraction models. To find the right set of predicates in less iterations, many heuristics have been proposed. For example, Ashutosh Gupta and Thomas A. Henzinger et al. accelerated the search for the right predicates by exploring the bad abstraction traces [21]. By contrast, we employ abstraction refinement to address the control space explosion problem resulting from the thread interleavings. Our abstraction and refinement methods are both different with that of predicate abstraction.

The work most related to ours focuses on interference abstraction (IA) [35]. N. Sinha and C. Wang also performed abstraction refinement to deal with the overhead of the exact encoding of the concurrent behavior. However, they abstracted the behavior by restricting the sets of read events and read-write links, while we consider all read events and read-write links but relax the scheduling constraint. Accordingly, their abstraction was refined by introducing new read events and read-write links, while we perform the refinement by exploring a graph-based method to analyze core kernel reasons that make an counterexample infeasible. Moreover, they employ a mixed framework of over- and under-approximations, while our method produces only over-approximation abstractions. Given that their implementation was for Java program slices, an empirical comparision between their and our method is difficult.

Another work closely related to ours is [26]. In this work, M. Kusano and C. Wang also presented a set of deduction rules to help determine the infeasibility of an interference combination. However, our task is to determine the feasibility of a counterexample which contains a large number of read-write links, and our main innovation is to devise a graph-based refinement generation method to obtain an effective refinement constraint. In addition, our deduction rules are much simpler yet stronger than theirs.

To deal with the interleaving problem, A. Farzan and Z. Kincaid also divided the verification into data and control modules, and incorporated them into an abstraction refinement framework [17, 18]. The difference is that in their work, the verification is reasoned by data-flow analysis, while we represent the program by SSA statements and employ graph and constraint based EOG analysis approaches to do the refinement. In addition, their work focuses on parameterized programs, while we concentrate on multi-threaded programs based on PThreads.

9 CONCLUSIONS

This paper proposed a scheduling constraint based abstraction refinement method for multi-threaded program verification. To obtain an effective refinement, we also devised two graph-based algorithms for counterexample validation and refinement generation. Our experiment results on benchmarks of SV-COMP 2017 show that our method is promising and significantly outperforms the existing state-of-the-art tools. We plan to extend this technique to weak memory models, such as TSO, PSO, POWER, in the future.

REFERENCES

[1] Parosh Aziz Abdulla, Stavros Aronis, Bengt Jonsson, and Konstantinos F. Sagonas. 2014. Optimal dynamic partial order reduction. In *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL*. 373–384.

[2] Sarita V. Adve and Kourosh Gharachorloo. 1996. Shared Memory Consistency Models: A Tutorial. *IEEE Computer* 29, 12 (1996), 66–76.

[3] Jade Alglave, Daniel Kroening, and Michael Tautschnig. 2013. Partial Orders for Efficient Bounded Model Checking of Concurrent Software. In *International Conference on Computer Aided Verification, CAV*. 141–157.

[4] Jiri Barnat, Lubos Brim, Vojtech Havel, Jan Havlícek, Jan Kriho, Milan Lenco, Petr Rockai, Vladimír Still, and Jirí Weiser. 2013. DiVinE 3.0 - An Explicit-State Model Checker for Multithreaded C & C++ Programs. In *International Conference on Computer Aided Verification, CAV*. 863–868.

[5] Tom Bergan, Luis Ceze, and Dan Grossman. 2013. Input-covering schedules for multithreaded programs. In *ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA*. 677–692.

[6] Dirk Beyer. 2017. Reliable and Reproducible Competition Results with BenchExec and Witnesses (Report on SV-COMP 2017). In *International Conference on Tools and Algorithms for Construction and Analysis of Systems, TACAS*.

[7] Armin Biere, Alessandro Cimatti, Edmund M. Clarke, and Yunshan Zhu. 1999. Symbolic Model Checking without BDDs. In *International Conference on Tools and Algorithms for Construction and Analysis of Systems, TACAS*. 193–207.

[8] Ahmed Bouajjani, Michael Emmi, Constantin Enea, and Jad Hamza. 2015. Tractable Refinement Checking for Concurrent Objects. In *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL*. 651–662.

[9] Katherine E. Coons, Madan Musuvathi, and Kathryn S. McKinley. 2013. Bounded partial-order reduction. In *ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications, OOPSLA*. 833–848.

[10] Lucas C. Cordeiro, Jeremy Morse, Denis Nicole, and Bernd Fischer. 2012. Context-Bounded Model Checking with ESBMC 1.17 - (Competition Contribution). In *International Conference on Tools and Algorithms for Construction and Analysis of Systems, TACAS*. 534–537.

[11] Andrei Marian Dan, Yuri Meshman, Martin T. Vechev, and Eran Yahav. 2013. Predicate Abstraction for Relaxed Memory Models. In *International Static Analysis Symposium, SAS*. 84–104.

[12] Andrei Marian Dan, Yuri Meshman, Martin T. Vechev, and Eran Yahav. 2015. Effective Abstractions for Verification under Relaxed Memory Models. In *International Conference on Verification, Model Checking and Abstract Interpretation, VMCAI*. 449–466.

[13] Pantazis Deligiannis, Alastair F. Donaldson, and Zvonimir Rakamaric. 2015. Fast and Precise Symbolic Analysis of Concurrency Bugs in Device Drivers (T). In *30th IEEE/ACM International Conference on Automated Software Engineering, ASE 2015, Lincoln, NE, USA*. 166–177.

[14] Thomas Dinsdale-Young, Lars Birkedal, Philippa Gardner, Matthew J. Parkinson, and Hongseok Yang. 2013. Views: compositional reasoning for concurrent programs. In *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL*. 287–300.

[15] Karam Abd Elkader, Orna Grumberg, Corina S. Pasareanu, and Sharon Shoham. 2015. Automated Circular Assume-Guarantee Reasoning. In *International Symposium on Formal Methods, FM*. 23–39.

[16] Karam Abd Elkader, Orna Grumberg, Corina S. Pasareanu, and Sharon Shoham. 2016. Automated Circular Assume-Guarantee Reasoning with N-way Decomposition and Alphabet Refinement. In *International Conference on Computer Aided Verification, CAV*. 329–351.

[17] Azadeh Farzan and Zachary Kincaid. 2012. Verification of parameterized concurrent programs by modular reasoning about data and control. In *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL*. 297–308.

[18] Azadeh Farzan and Zachary Kincaid. 2013. Duet: Static Analysis for Unbounded Parallelism. In *International Conference on Computer Aided Verification, CAV*. 191–196.

[19] Ivan Gavran, Filip Niksic, Aditya Kanade, Rupak Majumdar, and Viktor Vafeiadis. 2015. Rely/Guarantee Reasoning for Asynchronous Programs. In *International Conference on Concurrency Theory, CONCUR*. 483–496.

[20] Henning Günther, Alfons Laarman, and Georg Weissenbacher. 2016. Vienna Verification Tool: IC3 for Parallel Software - (Competition Contribution). In *International Conference on Tools and Algorithms for Construction and Analysis of Systems, TACAS*. 954–957.

[21] Ashutosh Gupta, Thomas A. Henzinger, Arjun Radhakrishna, Roopsha Samanta, and Thorsten Tarrach. 2015. Succinct Representation of Concurrent Trace Sets. In *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL*. 433–444.

[22] Ashutosh Gupta, Corneliu Popeea, and Andrey Rybalchenko. 2011. Predicate abstraction and refinement for verifying multi-threaded programs. In *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL*. 331–344.

[23] Ashutosh Gupta, Corneliu Popeea, and Andrey Rybalchenko. 2011. Threader: A Constraint-Based Verifier for Multi-threaded Programs. In *International Conference on Computer Aided Verification, CAV*. 412–417.

[24] Jeff Huang. 2015. Stateless model checking concurrent programs with maximal causality reduction. In *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI*. 165–174.

[25] 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 *International Conference on Computer Aided Verification, CAV*. 585–602.

[26] Markus Kusano and Chao Wang. 2016. Flow-sensitive composition of thread-modular abstract interpretation. In *Proceedings of the 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering, FSE 2016, Seattle, WA, USA, November 13-18*. 799–809.

[27] Ori Lahav and Viktor Vafeiadis. 2015. Owicki-Gries Reasoning for Weak Memory Models. In *International Colloquium on Automata, Languages and Programming, ICALP*. 311–323.

[28] Nuno Machado, Brandon Lucia, and Luís E. T. Rodrigues. 2015. Concurrency debugging with differential schedule projections. In *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI*. 586–595.

[29] Alexander Malkis, Andreas Podelski, and Andrey Rybalchenko. 2010. Thread-Modular Counterexample-Guided Abstraction Refinement. In *International Static Analysis Symposium, SAS*. 356–372.

[30] Truc L. Nguyen, Omar Inverso, Bernd Fischer, Salvatore La Torre, and Gennaro Parlato. 2017. Lazy-CSeq 2.0: Combining lazy sequentialization with abstract interpretation - (Competition Contribution). In *International Conference on Tools and Algorithms for Construction and Analysis of Systems, TACAS*.

[31] Corneliu Popeea and Andrey Rybalchenko. 2013. Threader: A Verifier for Multi-threaded Programs - (Competition Contribution). In *International Conference on Tools and Algorithms for Construction and Analysis of Systems, TACAS*. 633–636.
[32] Corneliu Popeea, Andrey Rybalchenko, and Andreas Wilhelm. 2014. Reduction for compositional verification of multi-threaded programs. In *Formal Methods in Computer-Aided Design, FMCAD*. 187–194.
[33] Shaz Qadeer and Jakob Rehof. 2005. Context-Bounded Model Checking of Concurrent Software. In *International Conference on Tools and Algorithms for Construction and Analysis of Systems, TACAS*. 93–107.
[34] Zvonimir Rakamaric and Michael Emmi. 2014. SMACK: Decoupling Source Language Details from Verifier Implementations. In *Proceedings of the 26th International Conference on Computer Aided Verification, CAV 2014, Vienna, Austria, July 18-22*. 106–113.
[35] Nishant Sinha and Chao Wang. 2011. On interference abstractions. In *ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL*. 423–434.
[36] SV-COMP. 2017. 2017 software verification competition. (Warning: It will violate our anonymity). http://sv-comp.sosy-lab.org/2017/. (2017).
[37] Paul Thomson, Alastair F. Donaldson, and Adam Betts. 2014. Concurrency testing using schedule bounding: an empirical study. In *ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, PPoPP*. 15–28.
[38] Ermenegildo Tomasco, Omar Inverso, Bernd Fischer, Salvatore La Torre, and Gennaro Parlato. 2015. Verifying Concurrent Programs by Memory Unwinding. In *International Conference on Tools and Algorithms for Construction and Analysis of Systems, TACAS*. 551–565.
[39] Ermenegildo Tomasco, Truc L. Nguyen, Omar Inverso, Bernd Fischer, Salvatore La Torre, and Gennaro Parlato. 2016. MU-CSeq 0.4: Individual Memory Location Unwindings - (Competition Contribution). In *International Conference on Tools and Algorithms for Construction and Analysis of Systems, TACAS*. 938–941.
[40] Jingyue Wu, Yang Tang, Gang Hu, Heming Cui, and Junfeng Yang. 2012. Sound and precise analysis of parallel programs through schedule specialization. In *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI*. 205–216.
[41] Naling Zhang, Markus Kusano, and Chao Wang. 2015. Dynamic partial order reduction for relaxed memory models. In *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI*. 250–259.
[42] Xin Zhang, Ravi Mangal, Radu Grigore, Mayur Naik, and Hongseok Yang. 2014. On abstraction refinement for program analyses in Datalog. In *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI*. 27.
[43] Manchun Zheng, Michael S. Rogers, Ziqing Luo, Matthew B. Dwyer, and Stephen F. Siegel. 2015. CIVL: Formal Verification of Parallel Programs. In *International Conference on Automated Software Engineering, ASE*. 830–835.