A Concurrency Problem with Exponential DPLL(T) Proofs

Extended Version

Liana Hadarean[1], Alex Horn[1], and Tim King[2]

[1] University of Oxford
liana.hadarean@cs.ox.ac.uk, alex.horn@cs.ox.ac.uk
[2] Verimag
tim.king@imag.fr

Abstract

Many satisfiability modulo theories solvers implement a variant of the DPLL($\mathcal{T}$) framework which separates theory-specific reasoning from reasoning on the propositional abstraction of the formula. Such solvers conclude that a formula is unsatisfiable once they have learned enough theory conflicts to derive a propositional contradiction. However some problems, such as the diamonds problem, require learning exponentially many conflicts. We give a general criterion for establishing lower bounds on the number of theory conflicts in any DPLL($\mathcal{T}$) proof for a given problem. We apply our criterion to two different state-of-the-art symbolic partial-order encodings of a simple, yet representative concurrency problem. Even though one of the encodings is asymptotically smaller than the other, we establish the same exponential lower bound proof complexity for both. Our experiments confirm this theoretical lower bound across multiple solvers and theory combinations.

1 Introduction

Many high-level verification tools rely on satisfiability modulo theories (SMT) solvers to discharge verification conditions in a variety of first-order logic theory theories. State-of-the-art SMT solvers decide such problems by implementing variations on the DPLL($\mathcal{T}$) framework. The DPLL($\mathcal{T}$) framework integrates a theory-specific solver with efficient search over the propositional abstraction of the formula. For this, DPLL($\mathcal{T}$) uses a propositional (SAT) solver that searches for a satisfying assignment to the propositional abstraction of the formula. When such an assignment is found, a theory solver checks that this propositional assignment is theory consistent. If it is not, a theory conflict (or $\mathcal{T}$-conflict) clause is added, summarizing the inconsistency and preventing the SAT solver from exploring this part of the search space again. The process continues until either a theory consistent satisfying assignment is found, or a contradiction can be derived purely on the propositional level using the learned theory conflicts. While usually efficient in practice, there are well-known problems, such as the "diamonds problem" [21], on which the DPLL($\mathcal{T}$) framework cannot derive a contradiction using a polynomial number of theory conflicts. This issue has resurfaced in recent work on worst-case execution time [13]. This limitation stems from the fixed alphabet of the DPLL($\mathcal{T}$) theory conflicts. Despite work on addressing this inherent inefficiency, it is still an open problem [7, 22].

In this paper, we prove a general theorem for establishing lower bounds on the number of $\mathcal{T}$-conflicts in the DPLL($\mathcal{T}$) calculus [19] required to prove that a given formula is unsatisfiable. The theorem relies on the notion of *non-interfering critical assignments*: propositionally satisfying assignments that contain disjoint $\mathcal{T}$-conflicts. To the best of our knowledge, this is the first attempt at establishing a general framework for establishing lower bounds for DPLL($\mathcal{T}$) proofs.

We apply this theorem to study the DPLL($\mathcal{T}$) proof complexity of proving a safety property of a simple, yet challenging concurrency problem. The problem appears in the software verification competition (SV-COMP) and is of broad historical interest [20, 11]. We focus on encodings recently implemented in a bounded model checker [3] because they have been successfully used to find concurrency-related bugs in software such as the Apache HTTP server, PostgreSQL and the Linux kernel [3]. Informally, these encodings symbolically model a certain partial-ordering between memory accesses, similar to the *happens-before relations* in distributed systems [16].

Contributions. The main contributions of this paper are as follows: (1) we give a new result for establishing lower bounds on the size of DPLL($\mathcal{T}$) proofs of unsatisfiability; (2) we propose a new problem challenge for the SMT community, whose solution is directly relevant to finding concurrency-related bugs in software; (3) we establish a factorial lower bound on the size of DPLL($\mathcal{T}$) proofs of unsatisfiability for this challenge problem; finally, (4) we experimentally confirm the hardness of this problem.

Organization. We prove the lower bound theorem in section 2. We introduce the problem challenge and explain how to generate two equisatisfiable partial-order encodings in section 3. Given these encodings, we formalize the DPLL($\mathcal{T}$) proof size complexity of the challenge problem (section 4) and experimentally confirm its complexity (section 5). We conclude with a discussion of related work and future research directions in section 6.

2 Non-interfering Critical Assignments

In this section, we give a general theorem for establishing lower bounds on the number of $\mathcal{T}$-conflicts in all proofs that a formula ϕ is unsatisfiable in the DPLL($\mathcal{T}$) calculus [19]. The theorem is based on the notion of sets of *non-interfering critical assignments* for ϕ.

We assume readers are familiar with standard notions from SMT such as $\mathcal{T}$-conflicts, $\mathcal{T}$-validity, $\mathcal{T}$-lemmas, DPLL($\mathcal{T}$), etc. In DPLL($\mathcal{T}$), a proof of unsatisfiability for a $\mathcal{T}$-formula consists of a combination of learning $\mathcal{T}$-valid lemmas and performing resolution steps on the propositional abstraction, until the empty clause is derived. As in [19], we restrict the proofs to work over the fixed alphabet $\mathcal{A}$ of $\mathcal{T}$-atoms in the input formula and that all $\mathcal{T}$-lemmas are clauses. We use a simplified view of the DPLL($\mathcal{T}$) calculus [19] that only uses two rules: (i) propositional resolution (Res) and (ii) learning $\mathcal{T}$-valid clauses over the literals of $\mathcal{A}$ ($\mathcal{T}$-learn). We ignore $\mathcal{T}$-propagation and splitting-on-demand [6].

Notation. We fix a set of propositional variables $\mathcal{X}$ and use ℓ to denote literals over this set. A clause C is a set of literals interpreted as their disjunction. The empty clause $\square$ denotes false. A *partial assignment* M is a set of literals that does not contain both a variable and its negation. Partial assignments are interpreted as a conjunction $\bigwedge_{\ell \in M} \ell$ and are always propositionally consistent. An *assignment* M is a partial assignment s.t. for all $v \in \mathcal{X}$ either $v \in M$ or $\neg v \in M$. The negation of a clause is a set of literals $\neg C = \{\neg \ell \mid \ell \in C\}$ and is interpreted as a conjunction.

The propositional abstraction function $\_^{\mathbb{B}}$ is an injective map from $\mathcal{A}$ into $\mathcal{X}$. The $\mathcal{T}$-literals, written $\mathcal{L}_{\mathcal{A}}$, are the set of literals over $\mathcal{A}$. We lift $\_^{\mathbb{B}}$ to work over $\mathcal{T}$-literals and sets of $\mathcal{T}$-literals. We denote by L a $\mathcal{T}$-valid clause over $\mathcal{L}_{\mathcal{A}}$, $\models_{\mathcal{T}} \bigvee_{t \in L} t$, and $\neg L$ will denote a $\mathcal{T}$-conflict. A $\mathcal{T}$-conflict is a set of $\mathcal{T}$-literals whose conjunction is $\mathcal{T}$-unsatisfiable, $\neg L \models_{\mathcal{T}} \square$. A *minimal* $\mathcal{T}$-conflict has the additional property that every strict subset is $\mathcal{T}$-satisfiable.

Proofs. We assume the input $\mathcal{T}$-formula ϕ has already been converted to CNF and is represented as a finite set of clauses $C_1, \ldots C_\alpha$ over the variables in $\mathcal{X}$, the set of $\mathcal{T}$-atoms $\mathcal{A}$, and the boolean abstraction function $\_^{\mathbb{B}} : \mathcal{A} \to \mathcal{X}$. A Fixed-Alphabet-DPLL($\mathcal{T}$) proof has the form:

$$C_1, \ldots, C_\alpha, \ldots, C_k, \ldots, C_\beta = \square$$

where each C_k for $\alpha < k \leq \beta$ is derived from a previous clause using either the resolution rule (Res) or theory learning ($\mathcal{T}$-learn). Let $C_i \otimes_\ell C_j$ denote propositional resolution on ℓ.

$$\frac{C_1, \ldots, C_k \quad L \subseteq \mathcal{L}_{\mathcal{A}} \quad \models_{\mathcal{T}} \bigvee_{t \in L} t}{C_1, \ldots, C_k, L^{\mathbb{B}}} \ \mathcal{T}\text{-LEARN} \qquad \frac{C_1, \ldots, C_k \quad 1 \leq i < j \leq k \quad \ell \in C_i \quad \neg\ell \in C_j}{C_1, \ldots, C_k, C_i \otimes_\ell C_j} \ \text{RES}$$

The rule $\mathcal{T}$-learn adds a new clause $L^{\mathbb{B}}$ that corresponds to the propositional abstraction of a $\mathcal{T}$-valid clause. Clauses derived by $\mathcal{T}$-learn are called $\mathcal{T}$-lemmas. $\mathcal{T}$-learn is more general than Lazy Theory Learning [19], which requires the literals to be in the partial assignment.

Critical Assignments. Given a $\mathcal{T}$-formula ϕ, an assignment M is *critical* if it satisfies the initial propositional abstraction of ϕ (i.e., $M \models \bigwedge_{i=1}^{\alpha} C_i$) and there is exactly one minimal $\mathcal{T}$-conflict $\neg L$ such that $\neg L^{\mathbb{B}} \subseteq M$. We denote by Q a set of critical assignments for ϕ, all of which can be enumerated as $M_1, \ldots, M_{|Q|}$ and where $\neg L_i$ denotes the minimal $\mathcal{T}$-conflict for M_i. We say that Q is *non-interfering* whenever, for all $M_i \neq M_j$ in Q, $\neg L_i^{\mathbb{B}}$ is not a subset of M_j. In other words, no two assignments in Q contain the same $\mathcal{T}$-conflict.

Lemma 2.1. *Let M be a critical assignment for ϕ with the minimal $\mathcal{T}$-conflict $\neg L$, and Π be a Fixed-Alphabet-DPLL($\mathcal{T}$) proof that ϕ is unsatisfiable. There is a $\mathcal{T}$-learn application $C_k \in \Pi$ such that $\neg L^{\mathbb{B}} \subseteq \neg C_k \subseteq M$.*

Proof. The assignment M does not satisfy the last clause $C_\beta = \square$ in Π. Therefore, there is some first clause C_k that M does not satisfy in Π. The clause C_k cannot be an input clause as $M \models C_i$ for $1 \leq i \leq \alpha$. Additionally, C_k cannot be the result of Res: since C_k is the first unsatisfied clause, all $M \models C_i$ for $i < k$, and resolving C_i and $C_{i'}$ for $i \neq i' < k$ results in a clause satisfied by M. Thus C_k must be the result of a $\mathcal{T}$-learn application and $M \not\models C_k$. Since M is an assignment which does not satisfy C_k, M must contain the negation of all literals in C_k. Equivalently, $\neg C_k \subseteq M$. Let T be the $\mathcal{T}$-lemma corresponding to C_k: $C_k = T^{\mathbb{B}}$. As $\neg L^{\mathbb{B}}$ is the unique minimal subset of M that maps to a minimal theory conflict, $L \subseteq T$. Therefore, $\neg L^{\mathbb{B}} \subseteq \neg C_k \subseteq M$. $\square$

Intuitively Lemma 2.1 states that, for each critical assignment M, the proof of unsatisfiability must contain a clause, derived by $\mathcal{T}$-learn, which rules out M as a model of ϕ in the theory $\mathcal{T}$.

Theorem 2.2. *Let ϕ be an unsatisfiable $\mathcal{T}$-formula, and let Q be a non-interfering set of critical assignments for ϕ. Then all Fixed-Alphabet-DPLL($\mathcal{T}$) proofs that ϕ is unsatisfiable contain at least $|Q|$ applications of $\mathcal{T}$-learn.*

Proof. Let Π be any Fixed-Alphabet-*DPLL*($\mathcal{T}$) proof. We will show that there exists a surjective partial map from $\mathcal{T}$-lemmas in Π onto critical assignments in Q that contain the same $\mathcal{T}$-inconsistency. We examine the set of partial maps $\mathbf{F}$ over $(\alpha, \beta]$ indices such that $\mathbf{F}(k) = j$ only if $L_j^{\mathbb{B}} \subseteq C_k$ and C_k is a $\mathcal{T}$-learn application. Let the partial function $\mathbf{F}^*$ be a partial function that maps onto the maximal number of distinct $M \in Q$ among all such maps $\mathbf{F}$. If $\mathbf{F}^*$ maps onto all elements in Q, there are at least $|Q|$ applications $\mathcal{T}$-learn in Π. If $|Q| = 0$, the property trivially holds on Π.

For the remainder of this proof, assume that $|Q| \geq 1$. Suppose for contradiction that $\mathbf{F}^*$ is not surjective. We can then select some critical assignment M_j such that for all $k \in (\alpha, \beta]$ either k is not in the domain of $\mathbf{F}^*$ or $\mathbf{F}^*(k) \neq j$.

By Lemma 2.1, there exists a $\mathcal{T}$-LEARN application $C_k \in \Pi$ such that $\neg L_j^{\mathbb{B}} \subseteq \neg C_k \subseteq M_j$. As $L_j^{\mathbb{B}} \subseteq C_k$, we know that it is possible for $\mathbf{F}^*$ to map C_k to some $M_m \in Q$. As $\mathbf{F}^*$ is maximal and there is no conflict mapped to M_j, $\mathbf{F}^*(k) = m$ for some $m \neq j$. By the construction of $\mathbf{F}^*$, $L_m^{\mathbb{B}} \subseteq C_k$. Recall that $\neg C_k \subseteq M_j$. Thus $\neg L_m^{\mathbb{B}} \subseteq \neg C_k \subseteq M_j$. As M_j contains both $\neg L_j^{\mathbb{B}}$ and $\neg L_m^{\mathbb{B}}$ for some distinct M_m in Q, this contradicts the assumption that Q is non-interfering.

We can now conclude by contradiction that $\mathbf{F}^*$ maps some clause that is the result of $\mathcal{T}$-LEARN in Π onto each $M \in Q$. Therefore Π contains at least $|Q|$ applications of $\mathcal{T}$-LEARN. □

There are many instances in the literature of *diamond* benchmarks for which exponential lower bounds on the number of $\mathcal{T}$-conflicts have been given [21, 7, 17, 2, 13]. Theorem 2.2 can be seen as a generalization of the lower bound arguments for the diamond benchmarks. The rest of this paper is devoted to a novel application of Theorem 2.2.

3 Challenge problem

In this section we present a challenge problem based on the `fpk2013` SV-COMP concurrency benchmark [1]. This problem was first introduced in 1976 to illustrate the need for auxiliary variables in compositional proof rules for concurrent programs [20], and most recently it has resurfaced as a challenge problem for automated verification tools [11]. Consider the following simple shared memory program with $N + 1$ threads and a shared memory location x:

Thread $\mathtt{T}_0$		Thread $\mathtt{T}_1$				Thread $\mathtt{T}_N$
local $v_0 := [x]$	$\Vert$	**local** $v_1 := [x]$	$\Vert$	$\cdots$	$\Vert$	**local** $v_N := [x]$
assert$(v_0 \leq N)$	$\Vert$	$[x] := v_1 + 1$	$\Vert$		$\Vert$	$[x] := v_N + 1$

The memory at location x is denoted by $[x]$. We assume that $[x]$ is initially 0. Each thread $\mathtt{T}_i$ reads the value at memory location x into a CPU-local register v_i. For $i \geq 1$, thread $\mathtt{T}_i$ overwrites the memory at location x with the new value $v_i + 1$. For the rest of the paper, we denote the read of memory location x in $\mathtt{T}_0$ by r_{assert}. The reads and writes on memory location x in thread $\mathtt{T}_i$ for $i \geq 1$ are denoted by r_i and w_i, respectively. We follow the SV-COMP convention and assume sequential consistency [15]. Therefore, if we just consider the concurrent program $\mathtt{T}_1 \parallel \mathtt{T}_2$, we get the following six interleavings of shared memory accesses: (1) $r_1; w_1; r_2; w_2$, (2) $r_1; r_2; w_1; w_2$, (3) $r_1; r_2; w_2; w_1$, (4) $r_2; r_1; w_1; w_2$, (5) $r_2; r_1; w_2; w_1$, (6) $r_2; w_2; r_1; w_1$. The different orders can result in different final values of $[x]$. For example, $r_1; w_1; r_2; w_2$ results in the final value 2 at memory location x, whereas $r_1; r_2; w_1; w_2$ results in the final value $[x] = 1$.

We want to check that the assertion $v_0 \leq N$ in thread $\mathtt{T}_0$ cannot be violated. Intuitively, this assertion holds because each of the other N threads increments $[x]$ at most once. For a fixed N, we want to prove this automatically using bounded model checking. While it is easy to automatically prove this property on each separate interleaving, the number of interleavings grows exponentially ($(2N + 1)! \div 2^N$). Next, we explain how to generate symbolic partial-order encodings that formalize all interleavings as a single quantifier-free SMT query.

Partial-order encodings. We formalize two quantifier-free and equisatisfiable partial-order encodings of a concurrency semantics called SC-relaxed consistency [14]: a cubic-sized encoding ($\mathcal{E}^3$) and a quadratic-sized encoding ($\mathcal{E}^2$). The formula generated by each encoding is satisfiable if and only if the safety property in the shared memory program can be violated.

$$\mathbf{PPO} \triangleq \bigwedge \{(guard(e) \wedge guard(e')) \Rightarrow (\mathsf{c}_e \prec \mathsf{c}_{e'}) \mid e, e' \in \mathsf{E}\colon e \ll e'\}$$

$$\mathbf{WW}[x] \triangleq \bigwedge \{(\mathsf{c}_w \prec \mathsf{c}_{w'} \vee \mathsf{c}_{w'} \prec \mathsf{c}_w) \wedge \mathsf{s}_w \neq \mathsf{s}_{w'} \mid w, w' \in \mathsf{W}_x \wedge w \neq w'\}$$

$$\mathbf{RW}[x] \triangleq \bigwedge \{\mathsf{c}_w \prec \mathsf{c}_r \vee \mathsf{c}_r \prec \mathsf{c}_w \mid w \in \mathsf{W}_x \wedge r \in \mathsf{R}_x\}$$

$$\mathbf{RF_{TO}}[x] \triangleq \bigwedge \{guard(r) \Rightarrow \bigvee \{\mathsf{s}_w = \mathsf{s}_r \mid w \in \mathsf{W}_x\} \mid r \in \mathsf{R}_x\}$$

$$\mathbf{RF}^3[x] \triangleq \bigwedge \{(\mathsf{s}_w = \mathsf{s}_r) \Rightarrow (guard(w) \wedge val(w) = \mathsf{rv}_r \wedge \mathsf{c}_w \prec \mathsf{c}_r) \mid r \in \mathsf{R}_x \wedge w \in \mathsf{W}_x\}$$

$$\mathbf{FR}[x] \triangleq \bigwedge \{(\mathsf{s}_w = \mathsf{s}_r \wedge \mathsf{c}_w \prec \mathsf{c}_{w'} \wedge guard(w')) \Rightarrow (\mathsf{c}_r \prec \mathsf{c}_{w'}) \mid w, w' \in \mathsf{W}_x \wedge r \in \mathsf{R}_x\}$$

$$\mathcal{E}^3 \triangleq \bigwedge \left\{\mathbf{RF_{TO}}[x] \wedge \mathbf{RF}^3[x] \wedge \mathbf{FR}[x] \wedge \mathbf{WW}[x] \wedge \mathbf{RW}[x] \mid x \in \langle ADDRESS \rangle\right\} \wedge \mathbf{PPO}$$

$$\mathbf{RF}^2[x] \triangleq \bigwedge \{(\mathsf{s}_w = \mathsf{s}_r) \Rightarrow (\mathsf{c}_w = \mathsf{sup}_r \wedge guard(w) \wedge val(w) = \mathsf{rv}_r \wedge \mathsf{c}_w \prec \mathsf{c}_r) \mid r \in \mathsf{R}_x \wedge w \in \mathsf{W}_x\}$$

$$\mathbf{SUP}[x] \triangleq \bigwedge \{(\mathsf{c}_w \preceq \mathsf{c}_r \wedge guard(w)) \Rightarrow (\mathsf{c}_w \preceq \mathsf{sup}_r) \mid r \in \mathsf{R}_x \wedge w \in \mathsf{W}_x\}$$

$$\mathcal{E}^2 \triangleq \bigwedge \left\{\mathbf{RF_{TO}}[x] \wedge \mathbf{RF}^2[x] \wedge \mathbf{SUP}[x] \wedge \mathbf{WW}[x] \wedge \mathbf{RW}[x] \mid x \in \langle ADDRESS \rangle\right\} \wedge \mathbf{PPO}$$

Figure 1: Given a shared memory program structure $P = \langle \mathsf{E}, \ll, val, guard \rangle$, $\mathcal{E}^3$ and $\mathcal{E}^2$ encode P's SC-relaxed consistency [14] with a cubic and quadratic number of constraints, respectively.

To get $\mathcal{E}^3$ and $\mathcal{E}^2$, we make four simplifying assumptions about the program P under scrutiny: (i) P's weak memory concurrency semantics equates to *SC-relaxed consistency* [14]; (ii) P is well-structured; (iii) all loops in P have been unrolled so that the only remaining control-flow statements in P are if-then-else branches; finally, (iv) every shared memory location accessed by P is known at compile-time. Avoiding these restrictions is beyond the scope of this paper that concerns itself with SMT solvers rather than program analysis techniques.

The formulas generated by both encodings $\mathcal{E}^3$ and $\mathcal{E}^2$ have three parts: (i) *clock constraints* that partially order memory accesses, similar to the happens-before relation in distributed systems [16]; (ii) *value constraints* that determine what values are read or written by the program if those clock constraints hold; and (iii) *selection constraints* that associate each read to a specific write event. Our symbolic partial-order encoding is therefore parameterized by three theories: $\mathcal{T}_C$ for encoding the clock constraints, $\mathcal{T}_V$ for encoding constraints on the symbolic program values, and $\mathcal{T}_S$ for encoding selection constraints. We assume that $\mathcal{T}_C$'s signature includes strict and non-strict partial-order relations, denoted by $\prec$ and $\preceq$, respectively. We also assume that $\mathcal{T}_V$'s signature can encode a decidable fragment of common machine arithmetic such as bitvector or Presburger arithmetic. $\mathcal{T}_S$ is an uninterpreted theory.

Definition 3.1. *A* shared memory program structure *is a tuple $P = \langle \mathsf{E}, \ll, val, guard \rangle$ where E is a finite set of* events, *$\ll$ is a partial order on E, $val : \mathsf{E} \to \mathcal{T}_V$-terms and $guard : \mathsf{E} \to \mathcal{T}_V$-formulas. Let $\langle ADDRESS \rangle$ be the set of memory locations. We assume that the set of events E in P can be partitioned into reads R_x and writes W_x on memory location $x \in \langle ADDRESS \rangle$. Given an event e in E, let c_e and s_e be a $\mathcal{T}_C$-variable (*clock variables*) and $\mathcal{T}_S$-variable (*selection variables*), respectively. For each read $r \in \mathsf{R}$, let rv_r be a unique $\mathcal{T}_V$-variable, called* read variable*. The function val maps a write event $w \in \mathsf{W}$ to a $\mathcal{T}_V$-term $val(w)$ built from read variables.*

The partial order $\ll$ is the *preserved program order* (PPO) [4, 3]. The intuition behind PPO is that it determines which events cannot be reordered in any execution of the program. For sequentially consistent programs, the preserved program order corresponds to the order of instructions in each thread. Note that $\langle \mathsf{E}, \ll \rangle$ can be relaxed for weaker forms of consistency such as TSO, e.g. [3]. Intuitively, given an event e in E, $guard(e)$ denotes the necessary condition for e to be enabled. The equality $\mathsf{s}_w = \mathsf{s}_r$ in the theory $\mathcal{T}_S$ means that a read event r is 'selected' so that its input value is equal to the output of a write event w. That is to say, when $\mathsf{s}_w = \mathsf{s}_r$ holds, the $\mathcal{T}_V$-variable rv_r is equal to the term $val(w)$.

Example 3.2. *The program described in section 3 for $N = 2$ corresponds to the following:*

- $\mathsf{E} = \{w_{init}, r_1, w_1, r_2, w_2, r_{assert}\}$ *is partitioned into* $\mathsf{R}_x = \{r_1, r_2, r_{assert}\}$ *and* $\mathsf{W}_x = \{w_{init}, w_1, w_2\}$ *where $x \in \langle ADDRESS \rangle$ is the concrete memory location accessed by threads T_0, T_1 and T_2.*
- *According to PPO: $w_{init} \ll r_1 \ll w_1$, $w_{init} \ll r_2 \ll w_2$, and $w_{init} \ll r_{assert}$.*
- *The val function is defined as $val(w_{init}) \triangleq 0$, $val(w_1) \triangleq \mathsf{rv}_{r_1} + 1$ and $val(w_2) \triangleq \mathsf{rv}_{r_2} + 1$.*
- *Since the program has no if-then-else statements, $guard(e) =$ **true** for all events e in E.*

Figure 1 shows how to generate the cubic-size $\mathcal{E}^3$ and quadratic-size $\mathcal{E}^2$ partial-order encoding for a given shared memory program structure $P = \langle \mathsf{E}, \ll, val, guard \rangle$. The first four formulas, **PPO**, **WW**$[x]$, **RW**$[x]$, and **RF$_{\mathbf{TO}}$**$[x]$, are shared by $\mathcal{E}^3$ and $\mathcal{E}^2$. The constraint **PPO** encodes the preserved program order $\ll$. The remaining constraints are with respect to some concrete memory location x. To model the information flow in the program, we encode a form of the *read-from* relation [4, 3]. For a fixed memory location x this relation defines a function from R_x to W_x. We model this through the selection variables s_r and s_w, for each read $r \in \mathsf{R}_x$ and write $w \in \mathsf{W}_x$, together with the equality $\mathsf{s}_r = \mathsf{s}_w$. The intuition is that the value of a write event $w \in \mathsf{W}_x$ is observed by a read event $r \in \mathsf{R}_x$ iff $\mathsf{s}_r = \mathsf{s}_w$. The **RF$_{\mathbf{TO}}$** constraints ensures that at least one such equality holds for every read. **WW** encodes that all writes on the same shared memory location are totally ordered in the happens-before relation and cannot have the same selection value, and **RW** encodes that every read r and write w on the same shared memory location satisfy that r happens-before w, or vice versa. Note that if $\prec$ is a total order, then **WW** is equivalent to the clock and selection variables being distinct. (In practice, the s_w variables are optimized out as distinct constants.) The same is not true for **RW** because two reads can have the same clock variables.

The main difference between $\mathcal{E}^3$ and $\mathcal{E}^2$ is how they encode values being overwritten in memory. A read r in R_x can read from a write w in W_x if w is the most recent write to x that happens before r. In the case of $\mathcal{E}^3$, this is encoded by **FR** which corresponds to the 'from-read' axiom [4, 3], also known as the 'conflict relation' [8]. This formula introduces a cubic number of constraints. By contrast, $\mathcal{E}^2$ encodes the **SUP** constraint that requires only a quadratic number of constraints. For this, **SUP** introduces a new variable sup_r for every read r in R_x to encode the least upper bound (supremum) of all writes in W_x that happen-before r. Since the set $\{\mathsf{c}_w \mid w \in \mathsf{W}_x\}$, for all memory locations x, is totally ordered with respect to $\prec$ in $\mathcal{T}_C$ by **WW**$[x]$, sup_r is the maximum of all writes in W_x that happen-before r in R_x according to $\prec$. It was previously shown in [14, Theorem 4] that for a given shared memory program structure P the formulas $\mathcal{E}^3$ and $\mathcal{E}^2$ are equisatisfiable.

4 Lower Bounds for Quadratic and Cubic Encodings

We show that the challenge problem from section 3 requires DPLL($\mathcal{T}$) to enumerate at least $N!$ theory conflicts before it finds a proof of unsatisfiability, for either of the $\mathcal{E}^3$ or $\mathcal{E}^2$ encoding where N is the number of threads.

$$\phi^3 \equiv \underbrace{\mathsf{c}_{w_{init}} \prec \mathsf{c}_{r_{assert}}}_{\textbf{PPO}} \wedge \underbrace{\bigwedge_{i=1\ldots N} \mathsf{c}_{w_{init}} \prec \mathsf{c}_{r_i} \prec \mathsf{c}_{w_i} \wedge}_{\textbf{PPO}} \underbrace{\bigwedge_{w,w' \in \mathsf{W}, w \neq w'} \mathsf{c}_w \neq \mathsf{c}_{w'} \wedge \mathsf{s}_w \neq \mathsf{s}_{w'} \wedge}_{\textbf{WW}[x]} \underbrace{\bigwedge_{w \in \mathsf{W}, r \in \mathsf{R}} \mathsf{c}_w \neq \mathsf{c}_r \wedge}_{\textbf{RW}[x]}$$

$$\underbrace{\bigwedge_{w \in \mathsf{W}, r \in \mathsf{R}} (\mathsf{s}_w = \mathsf{s}_r) \Rightarrow \mathsf{c}_w \prec \mathsf{c}_r \wedge}_{\textbf{RF}^3[x]} \underbrace{\bigwedge_{r \in \mathsf{R}} (\mathsf{s}_{w_{init}} = \mathsf{s}_r) \Rightarrow 0 = \mathsf{rv}_r \wedge}_{\textbf{RF}^3[x]} \underbrace{\bigwedge_{i=1\ldots N, r \in \mathsf{R}} (\mathsf{s}_{w_i} = \mathsf{s}_r) \Rightarrow \mathsf{rv}_{r_i} + 1 = \mathsf{rv}_r}_{\textbf{RF}^3[x]}$$

$$\underbrace{\bigwedge_{w,w' \in \mathsf{W}, r \in \mathsf{R}} (\mathsf{s}_w = \mathsf{s}_r \wedge \mathsf{c}_w \prec \mathsf{c}_{w'}) \Rightarrow \mathsf{c}_r \prec \mathsf{c}_{w'} \wedge}_{\textbf{FR}[x]} \underbrace{\bigwedge_{r \in \mathsf{R}} \left(\bigvee_{w \in \mathsf{W}} \mathsf{s}_w = \mathsf{s}_r \right)}_{\textbf{RF}_{\textbf{TO}}[x]} \wedge \underbrace{\mathsf{rv}_{r_{assert}} > N}_{\textbf{assert}(v_0 \leq N)}$$

Figure 2: The $\mathcal{E}^3$ encoding for the challenge problem (when $\prec$ is total).

We begin by constructing a formula that encodes the challenge program using the $\mathcal{E}^3$ encoding. As $\mathcal{E}^3$ is not directly in CNF, we perform the following simplifications in order to apply Theorem 2.2: (i) all of the guards *guard*(e) are ignored because they always evaluate to **true**, and (ii) implications are distributed across conjunctions in the $\textbf{RF}^3[x]$ constraints $[A \Rightarrow (B \wedge C)$ iff $(A \Rightarrow B) \wedge (A \Rightarrow C)]$. We also assume that $\prec$ is a total order in $\mathcal{T}_C$, and that $\mathcal{T}_V$ is either bit-vector, Presburger, or real arithmetic. We denote by $\mathcal{T}$ the standard combined theory $\mathcal{T}_C + \mathcal{T}_V + \mathcal{T}_S$. Figure 2 shows the resulting quantifier-free $\mathcal{T}$-formula, denoted by ϕ^3. Note that ϕ^3 is in CNF if we interpret implications in the obvious way. Note that in the $\textbf{RF}^3[x]$ constraints, each $val(w)$ term has been replaced by either 0 or $\mathsf{rv}_{r_i} + 1$.

Let S_N be the set of all permutations over $[1, N]$. Consider the following sequence of events that can be constructed from the permutation function π in S_N:

$$\sigma(\pi) : w_{init}, r_{\pi(1)}, w_{\pi(1)}, r_{\pi(2)}, w_{\pi(2)}, \ldots, r_{\pi(N)}, w_{\pi(N)}, r_{assert}.$$

The run of $\sigma(\pi)$ corresponds to satisfying the following clock and selection constraints:

$$\mathsf{c}_{w_{init}} \prec \mathsf{c}_{r_{\pi(1)}} \prec \mathsf{c}_{w_{\pi(1)}} \prec \cdots \prec \mathsf{c}_{r_{assert}}, \quad \mathsf{s}_{w_{init}} = \mathsf{s}_{r_{\pi(1)}}, \quad \bigwedge_{i=1\ldots N-1} \mathsf{s}_{w_{\pi(i)}} = \mathsf{s}_{r_{\pi(i+1)}}, \text{ and } \quad \mathsf{s}_{w_{\pi(N)}} = \mathsf{s}_{r_{assert}}$$

with distinct values for all s_w variables. A first-order variable assignment ν_π can be constructed to satisfy the above constraints. (An explicit construction of ν_π and proofs for Lemma 4.1 and Theorem 4.3 are given in Appendix A.) For each $\mathcal{T}_C$ or $\mathcal{T}_S$ literal ℓ, we include $\ell^{\mathbb{B}}$ in an assignment M_π if ℓ holds under ν_π. Consider the following $\mathcal{T}_V$-conflict:

$$\neg L_\pi = \left\{ \mathsf{rv}_{r_{\pi(1)}} = 0 \right\} \cup \left\{ \mathsf{rv}_{r_{\pi(i)}} + 1 = \mathsf{rv}_{r_{\pi(i+1)}} \mid i = 1 \ldots N-1 \right\} \cup \left\{ \mathsf{rv}_{r_{\pi(N)}} + 1 = \mathsf{rv}_{r_{assert}} \right\} \cup \left\{ \mathsf{rv}_{r_{assert}} > N \right\}.$$

Note that each $\ell \in \neg L_\pi$ is unit-propagated by the $\mathcal{T}_C$ and $\mathcal{T}_S$ literals already in M_π on the propositional abstraction of ϕ^3. We add $\neg L_\pi^{\mathbb{B}}$ to M_π. The remaining $\mathcal{T}_V$ equality atoms in ϕ^3 are added negatively. Now M_π satisfies the propositional abstraction of ϕ^3.

Lemma 4.1. *The assignment M_π is a critical assignment for ϕ^3 with the theory conflict $\neg L_\pi$.*

Theorem 4.2. *All Fixed-Alphabet-DPLL($\mathcal{T}$) proofs for ϕ^3 contain at least $N!$ applications of $\mathcal{T}$-*LEARN.

Proof. Let $Q = \{M_\pi \mid \pi \in S_N\}$. For each pair of distinct π and π' in S_N, there is some adjacent pair of events with a different order in $\sigma(\pi)$ and $\sigma(\pi')$. Select k so that $\left\langle r_{\pi(k)}, r_{\pi(k+1)} \right\rangle \neq \left\langle r_{\pi'(k)}, r_{\pi'(k+1)} \right\rangle$. The literal $(\mathsf{rv}_{r_{\pi(k)}} + 1 = \mathsf{rv}_{r_{\pi(k+1)}})^{\mathbb{B}}$ is in $\neg L_\pi$ and is not in $M_{\pi'}$. Thus $\neg L^{\mathbb{B}}_\pi$ is not a subset of $M_{\pi'}$, and Q is non-interfering. The lemma follows directly from Theorem 2.2. □

Theorem 4.3. *Let ϕ^2 be the $\mathcal{E}^2$ encoding of the challenge problem. All Fixed-Alphabet-DPLL($\mathcal{T}$) proofs that ϕ^2 are unsatisfiable contain at least $N!$ application of $\mathcal{T}$-LEARN.*

An important difference between the diamond benchmarks and this problem is that for diamonds it is reasonable to describe all minimal $\mathcal{T}$-conflicts as they each also correspond to critical models. For the `fkp` problem, the encoding is more complex, and there are other classes of $\mathcal{T}$-conflicts. The set Q identifies those $\mathcal{T}$-lemmas that *must* appear during solving.

5 Experiments

In this section, we give experimental results that confirm the lower bounds on the DPLL($\mathcal{T}$) proofs for the two encodings of the problem challenge (section 3). Our experiments are carried out along three dimensions: we use four SMT solvers (Boolector v2.0.6 [9], CVC4 2015-03-14 [5], Yices v2.3.0 [10], and Z3 2015-03-29 [18]), and we evaluate both the cubic-size and quadratic-size encoding ($\mathcal{E}^3$ and $\mathcal{E}^2$) with respect to four different SMT-LIB theory combinations.

We performed all experiments on a 64-bit machine running GNU/Linux 3.16 with 2 Intel Xeon 2.5 GHz cores and 4 GB of memory. The timeout for each individual benchmark is 1 hour. Recall that $\mathcal{E}^3$ and $\mathcal{E}^2$ are parameterized by three theories, $\mathcal{T}_C$, $\mathcal{T}_S$ and $\mathcal{T}_V$. We experiment with the theory of reals $\mathcal{T}_{\mathbb{R}}$, the theory of integers $\mathcal{T}_{\mathbb{Z}}$, and the theory of bit-vectors $\mathcal{T}_{\mathbb{BV}}$. In our experiments, we instantiate $\langle \mathcal{T}_C, \mathcal{T}_S, \mathcal{T}_V \rangle$ to four configurations such that $\mathcal{T}_C = \mathcal{T}_S$:

(1) "real-clocks-int-val": $\langle \mathcal{T}_{\mathbb{R}}, \mathcal{T}_{\mathbb{R}}, \mathcal{T}_{\mathbb{Z}} \rangle$, (3) "bv-clocks-int-val": $\langle \mathcal{T}_{\mathbb{BV}}, \mathcal{T}_{\mathbb{BV}}, \mathcal{T}_{\mathbb{Z}} \rangle$, and
(2) "real-clocks-bv-val": $\langle \mathcal{T}_{\mathbb{R}}, \mathcal{T}_{\mathbb{R}}, \mathcal{T}_{\mathbb{BV}} \rangle$, (4) "bv-clocks-bv-val": $\langle \mathcal{T}_{\mathbb{BV}}, \mathcal{T}_{\mathbb{BV}}, \mathcal{T}_{\mathbb{BV}} \rangle$.

CVC4 and Z3 were run on all benchmarks. Boolector is only used on the fourth configuration, i.e. purely $\mathcal{T}_{\mathbb{BV}}$ benchmarks. Yices was run on the "real-clocks-int-val" and "bv-clocks-bv-val" configurations. We further distinguish between the SMT-LIB benchmarks by labelling them with $\mathcal{E}^3$ or $\mathcal{E}^2$. For example, 'real-clocks-bv-val-$\mathcal{E}^3$' identifies benchmarks generated with the cubic encoding in which $\mathcal{T}_C$, $\mathcal{T}_S$ and $\mathcal{T}_V$ are respectively instantiated as $\mathcal{T}_{\mathbb{R}}$, $\mathcal{T}_{\mathbb{R}}$, and $\mathcal{T}_{\mathbb{BV}}$.

For all the "*-bv-val" benchmarks (except CVC4 for "real-clocks-bv-val"), the solvers are essentially encoding the problem in propositional logic and using a SAT solver.[1] The process of encoding into propositional logic (*bit-blasting*) enables the solver to learn clauses not necessarily expressible in the original alphabet of the input atoms. We therefore call these solver and configuration pairs *bit-blasted combinations*. All other solver and configuration pairs are called *DPLL($\mathcal{T}$) combinations*. The DPLL($\mathcal{T}$) combinations are the "*-int-val" configurations, and the run of CVC4 on "real-clocks-bv-val".[2] DPLL($\mathcal{T}$) combinations use Fixed-Alphabet-DPLL($\mathcal{T}$) proofs, whereas bit-blasted combinations generally do not.

Given an instantiation of $\langle \mathcal{T}_C, \mathcal{T}_S, \mathcal{T}_V \rangle$, we separately encode the `fkp2013-unsat` concurrency benchmarks with $\mathcal{E}^3$ and $\mathcal{E}^2$ for all $N \in [3, 9]$. There are a total of 56 different unsatisfiable SMT-LIB benchmarks. The size of each benchmark depends on N and whether we used $\mathcal{E}^3$ or $\mathcal{E}^2$. For example, for $N = 9$, the total number of symbolic expressions in $\mathcal{E}^3$ is 4085, whereas $\mathcal{E}^2$ yields only 1604 symbolic expressions.

[1] CVC4 was run with the flag `--bitblast=eager` on "bv-clocks-bv-val" benchmarks [12].
[2] In this configuration CVC4 does not eagerly reduce $\mathcal{T}_{\mathbb{BV}}$ to SAT.

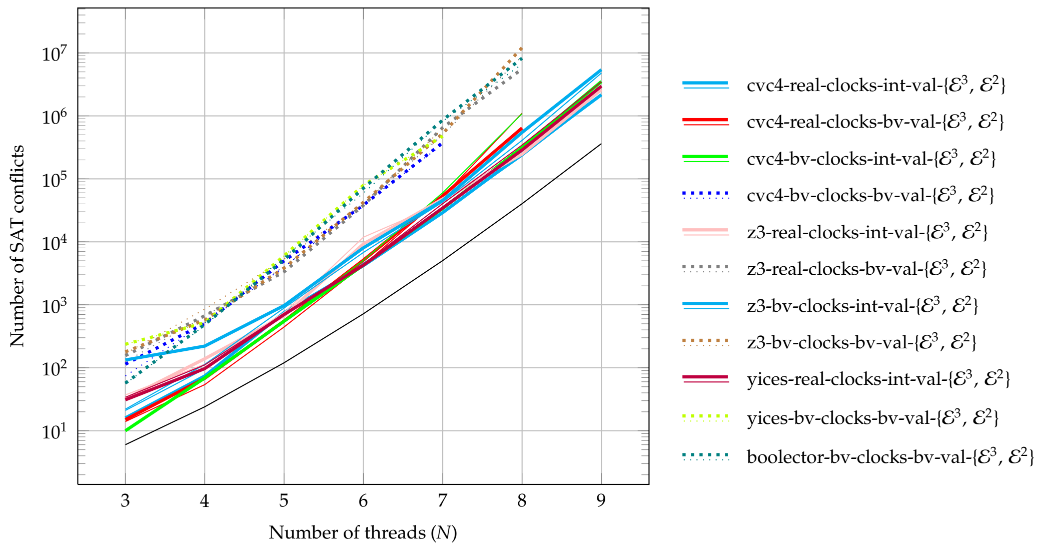

Figure 3: Experimental results for the `fkp2013-unsat` benchmark using four SMT solvers and four SMT-LIB theory combinations. The graph shows the factorial growth of the number of SAT conflicts in both the cubic-size and quadratic-size partial-order encoding as N increases.

Figure 3 charts the number of conflicts reported by each solver during execution.[3] Executions that exceeded the time limit of 1 hour are not included. The x-axis corresponds to N. The y-axis corresponds to the number of conflicts generated by the solver and has a logarithmic scale. The legend for the chart groups together both the $\mathcal{E}^3$ (bold lines) and $\mathcal{E}^2$ (thin lines) for a solver and theory specification. These are further grouped into bit-blasted benchmarks (dotted lines) and DPLL($\mathcal{T}$) (solid lines). We also plot $N!$ as a black line. The goal of the Figure 3 is to convey the overall trends instead of compare individual data points.

We examine the number of SAT conflicts as it is a uniform and readily available statistic that is a lower bound on the number of proof steps taken by each solver. Across all combinations, the number of conflicts observed is above the $N!$ line. Thus the $N!$ theory conflict lower bound proofs given in section 4 holds for the DPLL($\mathcal{T}$) combinations. Our theoretical lower bounds do not extend to the bit-blasted combinations. Nevertheless, our experiments show that the number of SAT conflicts are two orders of magnitude higher than $N!$ for bit-blasted combinations. We therefore conjecture that a similar $N!$ lower bound exists for Res proofs for the bit-blasted combinations. We also examined CVC4's more detailed statistics on the DPLL($\mathcal{T}$) combinations. We confirmed that the number of $\mathcal{T}_V$-conflicts is always above $N!$ on the DPLL($\mathcal{T}$) combinations.

6 Conclusion

In this paper, we have demonstrated a theoretical factorial lower bound on the number of $\mathcal{T}$-learn applications in all DPLL($\mathcal{T}$) proofs for a challenge problem of historical interest using two state-of-the-art encodings. Our encodings are most closely related to [3, 14]. Experiments

[3] Elapsed time and memory usage for the experiment is available in Appendix B.

confirm the theoretical lower bound for DPLL($\mathcal{T}$) proofs and show a strong relationship to the number of SAT conflicts in Res-proofs for bitblasted bitvector encodings. Both the theoretical relationships and the empirical relationships hold over a cubic $\mathcal{E}^3$ and a quadratic $\mathcal{E}^2$ encoding. Our experiments are therefore particularly significant for state-of-the-art tools such as CBMC (which implements a variant of $\mathcal{E}^3$). We believe that the kind of analysis we have undertaken throughout this paper provides an important diagnostic practice in the development of SMT encodings. Future work will focus on handling the value constraints for partial-order encodings of weak memory concurrency and improving the performance of the SMT solvers on such benchmarks by moving outside of Fixed-Alphabet-DPLL($\mathcal{T}$) proofs.

Acknowledgments Work partially supported by European Research Council project 280053 (CPROVER)"CPROVER" and Grant Agreement nr. 306595 "STATOR".

References

[1] fkp2013 SV-COMP Pthreads concurrency benchmark. `https://svn.sosy-lab.org/software/sv-benchmarks/trunk/c/pthread-lit/fkp2013_false-unreach-call.c?p=588`

[2] Albarghouthi, A., McMillan, K.L.: Beautiful Interpolants. CAV (2013)

[3] Alglave, J., Kroening, D., Tautschnig, M.: Partial orders for efficient bounded model checking of concurrent software. CAV (2013)

[4] Alglave, J., Maranget, L., Sarkar, S., Sewell, P.: Fences in weak memory models (extended version). FMSD (2012)

[5] Barrett, C., Conway, C.L., Deters, M., Hadarean, L., Jovanović, D., King, T., Reynolds, A., Tinelli, C.: CVC4. In: CAV, pp. 171–177 (2011)

[6] Barrett, C., Nieuwenhuis, R., Oliveras, A., Tinelli, C.: Splitting on demand in sat modulo theories. In: LPAR (2006)

[7] Bjørner, N., Dutertre, B., de Moura, L.: Accelerating Lemma Learning using Joins - DPLL(Join). In: LPAR (2008)

[8] Bouajjani, A., Derevenetc, E., Meyer, R.: Checking and enforcing robustness against tso. ESOP (2013)

[9] Brummayer, R., Biere, A.: Boolector: An Efficient SMT Solver for Bit-Vectors and Arrays. TACAS (2009)

[10] Dutertre, B.: Yices 2.2. In: Computer Aided Verification, pp. 737–744 (2014)

[11] Farzan, A., Kincaid, Z., Podelski, A.: Inductive data flow graphs. POPL (2013)

[12] Hadarean, L., Bansal, K., Jovanović, D., Barrett, C., Tinelli, C.: A Tale of Two Solvers: Eager and Lazy Approaches to Bit-Vectors. CAV (2014)

[13] Henry, J., Asavoae, M., Monniaux, D., Maiza, C.: How to Compute Worst-Case Execution Time by Optimization Modulo Theory and a Clever Encoding of Program Semantics. In: LCTES (2014)

[14] Horn, A., Kroening, D.: On partial order semantics for SAT/SMT-based symbolic encodings of weak memory concurrency. FORTE (2015), `http://arxiv.org/abs/1504.00037`, to appear.

[15] Lamport, L.: How to make a multiprocessor computer that correctly executes multiprocess programs. IEEE Trans. Comput. (1979)

[16] Lamport, L.: Time, clocks, and the ordering of events in a distributed system. CACM (1978)

[17] Mcmillan, K.L., Kuehlmann, A., Sagiv, M.: Generalizing DPLL to Richer Logics. CAV (2009)

[18] de Moura, L., Bjørner, N.: Z3: An Efficient SMT Solver. In: TACAS, pp. 337–340 (2008)

[19] Nieuwenhuis, R., Oliveras, A., Tinelli, C.: Solving SAT and SAT Modulo Theories: From an Abstract Davis–Putnam–Logemann–Loveland Procedure to DPLL(T). J. ACM (2006)

[20] Owicki, S., Gries, D.: Verifying properties of parallel programs: An axiomatic approach. CACM (1976)

[21] Strichman, O., Seshia, S.A., Bryant, R.E.: Deciding separation formulas with SAT. CAV (2002)

[22] Thakur, A., Reps, T.: A Method for Symbolic Computation of Abstract Operations. CAV (2012)

A Proofs for Lower Bounds

This section gives a more formal derivation for ν_π and M_π the concepts discussed in section 4, and proofs for Lemma 4.1 and Theorem 4.3. We use $\mu, \nu \models \phi$ to denote that a Σ-structure μ and a variable assignment over μ satisfies a Σ-formula ϕ.

Let μ be any $(\mathcal{T}_C + \mathcal{T}_S + \mathcal{T}_V)$-structure with the additional constraint that $\mathcal{T}_C$, $\mathcal{T}_S$, and $\mathcal{T}_V$ sorts are mapped to domains with cardinalities at least $|E|$, $N+1$, and $N+1$ respectively. Such a structure exists unless $\mathcal{T}_V$ is bit-vectors and the bit-width is insufficiently large. We now construct a first-order variable assignment ν_π over $\mathcal{T}_C$ and $\mathcal{T}_S$ variables to match $\sigma(\pi)$. Let $x_1 <^\mu \cdots <^\mu x_{|E|}$ be any arbitrary chain in the $\mathcal{T}_C$ domain of μ, and let $\langle y_0, y_1, \ldots, y_N \rangle$ be an arbitrary enumeration of $N+1$ distinct elements in the $\mathcal{T}_S$ domain. Both the x_i chain and the y_i sequence exist as the cardinalities are large enough. We now assign the c_e and s_e variables.

$$\nu_\pi(\mathsf{c}_e) = \begin{cases} x_1 & e = w_{init} \\ x_{2i} & e = r_{\pi(i)} \\ x_{2i+1} & e = w_{\pi(i)} \\ x_{2N+2} & e = r_{assert} \end{cases} \qquad \nu_\pi(\mathsf{s}_w) = \begin{cases} y_0 & w = w_{init} \\ y_i & w = w_{\pi(i)} \end{cases} \qquad \nu_\pi(\mathsf{s}_r) = \begin{cases} y_i & r = r_{\pi(i+1)} \\ y_N & r = r_{assert} \end{cases}$$

We construct a complete set of $\mathcal{T}$-literals H_π (either $\ell \in H_\pi$ or $\neg\ell \in H_\pi$ for all $\ell \in \mathcal{L}_{\mathcal{A}}$). This will correspond to M_π before abstraction. For any literal ℓ over $\mathcal{T}_C$ or $\mathcal{T}_S$ atoms, we evaluate ℓ w.r.t. μ and ν_π to assign it in H_π, i.e. $\ell \in H_\pi$ if $\mu, \nu_\pi \models \ell$. For atoms over $\mathcal{T}_V$, we include the $\neg L_\pi$ literals in H_π (defined in section 4). For all other $\mathcal{T}_V$ equalities ℓ in ϕ^3, we include $\neg\ell \in H_\pi$. We now let $M_\pi = {H_\pi}^{\mathbb{B}}$.

Proof of Lemma 4.1. Since $\neg L_\pi \subseteq H_\pi$ and $M_\pi = {H_\pi}^{\mathbb{B}}$, $\neg L_\pi^{\mathbb{B}} \subseteq M_\pi$. We now show that for each $\ell \in \neg L_\pi$, we can extend ν_π to a new assignment ν_π^ℓ so that $\mu, \nu_\pi^\ell \models h$ for all $h \in H_\pi \setminus \{\ell\}$. For brevity, we denote by $\ell_0 = \left(\mathsf{rv}_{r_{\pi(1)}} = 0\right)$, $\ell_i = \left(\mathsf{rv}_{r_{\pi(i)}} + 1 = \mathsf{rv}_{r_{\pi(i+1)}}\right)$ for $i \in 1 \ldots N-1$, $\ell_{assert1} = \left(\mathsf{rv}_{r_{\pi(N)}} + 1 = \mathsf{rv}_{r_{assert}}\right)$, and $\ell_{assert2} = (\mathsf{rv}_{r_{assert}} > N)$.

$$\nu_\pi^{\ell_0}(\mathsf{rv}_r) = \begin{cases} 1 & r = r_{\pi(1)} \\ j+1 & r = r_{\pi(j)} \\ N+1 & r = r_{assert} \end{cases} \qquad \nu_\pi^{\ell_i}(\mathsf{rv}_r) = \begin{cases} 0 & r = r_{\pi(1)} \\ j & r = r_{\pi(j)}, j < i \\ k+1 & r = r_{\pi(k)}, k \geq i \\ N+1 & r = r_{assert} \end{cases}$$

$$\nu_\pi^{\ell_{assert1}}(\mathsf{rv}_r) = \begin{cases} 0 & r = r_{\pi(1)} \\ j & r = r_{\pi(j)} \\ N+1 & r = r_{assert} \end{cases} \qquad \nu_\pi^{\ell_{assert2}}(\mathsf{rv}_r) = \begin{cases} 0 & r = r_{\pi(1)} \\ j & r = r_{\pi(j)} \\ N & r = r_{assert} \end{cases}$$

We omit $\_^\mu$ from the $\mathcal{T}_V$-constants $0, \ldots, N+1$ above. It is now that case that $\mu, \nu_\pi^\ell \models h$ for all $h \in H_\pi \setminus \{\ell\}$. Thus $H_\pi \setminus \{\ell\}$ is satisfiable modulo $\mathcal{T}$. As every subset of H_π excluding exactly one literal in $\neg L_\pi$ is satisfiable modulo $\mathcal{T}$, $\neg L_\pi$ is the unique minimal $\mathcal{T}$-conflict in H_π. Thus M_π is a critical assignment. □

Proof of Theorem 4.3. We extend ν_π to assign sup_r to match $\sigma(\pi)$: $\nu_\pi(\mathsf{sup}_{r_{\pi(1)}}) = \nu_\pi(\mathsf{c}_{w_{init}})$, $\nu_\pi(\mathsf{sup}_{r_{\pi(i)}}) = \nu_\pi(\mathsf{c}_{w_{\pi(i-1)}})$, and $\nu_\pi(\mathsf{sup}_{r_{assert}}) = \nu_\pi(\mathsf{c}_{w_{\pi(N)}})$. We follow the same construction of H_π, M_π, ν_π^ℓ, and Q as before for ϕ^3. Q is a set of non-interfering critical assignments for ϕ^2. □

B Time and Memory Usage

Elapsed time and memory usage for `fkp2013-unsat` benchmark; TIMEOUT = 1 hour.

CVC4			**Z3**			**Yices** and **Boolector**		
N	Time (s)	Memory (MB)	*N*	Time (s)	Memory (MB)	*N*	Time (s)	Memory (MB)
	cvc4-real-clocks-int-val-$\mathcal{E}^3$			*z3-real-clocks-int-val-$\mathcal{E}^3$*			*yices-real-clocks-int-val-$\mathcal{E}^3$*	
3	0.00	0.0	3	0.00	0.0	3	0.00	0.0
4	0.00	0.0	4	0.00	0.0	4	0.00	0.0
5	0.28	15.3	5	0.00	0.0	5	0.00	0.0
6	1.74	18.0	6	0.99	17.1	6	0.10	3.4
7	15.50	24.5	7	4.80	21.4	7	1.30	3.6
8	200.28	81.0	8	37.89	28.7	8	26.59	7.1
9	2718.45	557.3	9	697.24	45.8	9	1086.21	34.0
	cvc4-real-clocks-int-val-$\mathcal{E}^2$			*z3-real-clocks-int-val-$\mathcal{E}^2$*			*yices-real-clocks-int-val-$\mathcal{E}^2$*	
3	0.00	0.0	3	0.00	0.0	3	0.00	0.0
4	0.00	0.0	4	0.00	0.0	4	0.00	0.0
5	0.30	14.4	5	0.00	0.0	5	0.00	0.0
6	2.19	16.9	6	1.39	18.3	6	0.10	3.0
7	18.79	22.4	7	5.50	21.2	7	1.90	4.0
8	199.17	68.0	8	50.99	28.5	8	38.49	8.3
9	2906.83	579.2	9	694.27	42.4	9	1416.26	40.9
	cvc4-real-clocks-bv-val-$\mathcal{E}^3$			*z3-real-clocks-bv-val-$\mathcal{E}^3$*			*yices-bv-clocks-bv-val-$\mathcal{E}^3$*	
3	0.00	0.0	3	0.00	0.0	3	0.00	0.0
4	0.00	0.0	4	0.09	15.7	4	0.00	0.0
5	0.39	17.4	5	0.49	16.4	5	0.10	4.0
6	3.39	21.3	6	7.19	19.6	6	3.89	5.5
7	36.00	32.1	7	112.19	27.9	7	74.48	12.2
8	512.64	147.1	8	1415.20	49.8	8	TIMEOUT	70.9
9	TIMEOUT	597.3	9	TIMEOUT	68.5	9	TIMEOUT	96.4
	cvc4-real-clocks-bv-val-$\mathcal{E}^2$			*z3-real-clocks-bv-val-$\mathcal{E}^2$*			*yices-bv-clocks-bv-val-$\mathcal{E}^2$*	
3	0.00	0.0	3	0.00	0.0	3	0.00	0.0
4	0.09	15.5	4	0.10	15.7	4	0.00	0.0
5	0.59	16.6	5	0.79	16.9	5	0.10	4.0
6	5.20	19.8	6	8.29	20.3	6	3.90	5.6
7	56.09	32.2	7	97.79	25.9	7	76.59	12.0
8	1277.46	274.5	8	2441.23	56.9	8	TIMEOUT	74.3
9	TIMEOUT	554.1	9	TIMEOUT	62.6	9	TIMEOUT	96.0
	cvc4-bv-clocks-int-val-$\mathcal{E}^3$			*z3-bv-clocks-int-val-$\mathcal{E}^3$*			*boolector-bv-clocks-bv-val-$\mathcal{E}^3$*	
3	0.00	0.0	3	0.00	0.0	3	0.00	0.0
4	0.00	0.0	4	0.00	0.0	4	0.10	4.3
5	0.20	15.0	5	0.18	16.9	5	0.69	5.6
6	1.40	17.8	6	1.69	18.6	6	5.39	10.0
7	13.09	26.5	7	13.98	22.9	7	94.29	34.2
8	141.88	80.7	8	270.86	31.7	8	1491.46	95.2
9	1811.85	642.8	9	1755.95	57.1	9	TIMEOUT	140.7
	cvc4-bv-clocks-int-val-$\mathcal{E}^2$			*z3-bv-clocks-int-val-$\mathcal{E}^2$*			*boolector-bv-clocks-bv-val-$\mathcal{E}^2$*	
3	0.00	0.0	3	0.00	0.0	3	0.00	0.0
4	0.00	0.0	4	0.00	0.0	4	0.09	4.6
5	0.19	14.5	5	0.29	16.5	5	0.69	6.1
6	1.99	17.6	6	2.69	18.0	6	4.30	9.6
7	23.89	39.6	7	26.67	21.2	7	86.49	29.1
8	600.84	359.4	8	394.60	32.4	8	1122.07	87.4
9	TIMEOUT	978.5	9	2862.76	54.1	9	TIMEOUT	133.8
	cvc4-bv-clocks-bv-val-$\mathcal{E}^3$			*z3-bv-clocks-bv-val-$\mathcal{E}^3$*				
3	0.00	0.0	3	0.00	0.0			
4	0.00	0.0	4	0.00	0.0			
5	0.20	22.4	5	0.10	14.8			
6	2.69	40.9	6	3.10	25.1			
7	116.99	280.2	7	70.59	40.8			
8	TIMEOUT	1911.2	8	3521.01	145.8			
9	TIMEOUT	2022.0	9	TIMEOUT	133.6			
	cvc4-bv-clocks-bv-val-$\mathcal{E}^2$			*z3-bv-clocks-bv-val-$\mathcal{E}^2$*				
3	0.03	15.2	3	0.10	12.6			
4	0.00	0.0	4	0.10	13.1			
5	0.19	20.1	5	0.79	14.7			
6	2.98	40.5	6	10.59	26.1			
7	188.26	307.8	7	252.13	40.5			
8	TIMEOUT	1801.0	8	TIMEOUT	101.8			
9	TIMEOUT	1785.3	9	TIMEOUT	114.4			