# SMT Solving for the Theory of Ordering Constraints

Cunjing Ge[1(✉)], Feifei Ma[1], Jeff Huang[2], and Jian Zhang[1]

[1] Institute of Software, Chinese Academy of Sciences, Beijing, China
{gecj,maff,zj}@ios.ac.cn
[2] Department of Computer Science and Engineering, Texas A&M University, College Station, USA
jeff@cse.tamu.edu

**Abstract.** Constraint solving and satisfiability checking play an important role in various tasks such as formal verification, software analysis and testing. In this paper, we identify a particular kind of constraints called ordering constraints, and study the problem of deciding satisfiability modulo such constraints. The theory of ordering constraints can be regarded as a special case of difference logic, and is essential for many important problems in symbolic analysis of concurrent programs. We propose a new approach for checking satisfiability modulo ordering constraints based on the DPLL(T) framework, and present our experimental results compared with state-of-the-art SMT solvers on both benchmarks and instances of real symbolic constraints.

## 1 Introduction

In the past decade, constraint solving and satisfiability checking techniques and tools have found more and more applications in various fields like formal methods, software engineering and security. In particular, Satisfiability Modulo Theories (SMT) solvers play a vital role in program analysis and testing. This work is motivated by the increasingly important use of SMT solving for symbolic analysis of concurrent programs.

It is well-known that concurrent programs are error-prone. Analyzing concurrent programs has been a big challenge due to subtle interactions among the concurrent threads exacerbated by the huge thread scheduling space. Among the broad spectrum of concurrency analysis techniques, symbolic analysis is probably the most promising approach that has attracted significant research attention in recent years [7,9,16–18,20,23,25,27,30]. Generally speaking, it models the scheduling of threads as symbolic constraints over *order variables* corresponding to the execution order of critical operations performed by threads (such as

This work is supported in part by National Basic Research (973) Program of China (No. 2014CB340701), National Natural Science Foundation of China (Grant No. 91418206, 91118007) and the CAS/SAFEA International Partnership Program for Creative Research Teams.


X. Shen et al. (Eds.): LCPC 2015, LNCS 9519, pp. 287–302, 2016.
DOI: 10.1007/978-3-319-29778-1_18

shared data accesses and synchronizations). The symbolic constraints capture both data and control dependencies among threads such that any solution to the constraints corresponds to a valid schedule.

A key advantage of symbolic analysis is that it allows reasoning about thread schedules with the help of automated constraint solving. By encoding interesting properties (such as race conditions) as additional constraints and solving them with a constraint solver, we can verify if there exists any valid schedule that can satisfy the property. Such an approach has been used for finding concurrency bugs such as data races [18,25], atomicity violations [30], deadlocks [7], null pointer executions [9], etc., and has also been used to reproduce concurrency failures [20,23], to generate tests [8], and to verify general properties [16,17]. In our prior work [18], we developed a tool called RVPredict, which is able to detect data races based on symbolic analysis of the program execution trace.

Despite its huge potential, symbolic analysis has not been widely adopted in practice. The main obstacle is the performance of constraint solvers. For real world applications, the size of complex constraints can be extremely large that is very challenging for existing SMT solvers to solve. For example, for data race detection in RVPredict, the number of constraints is cubic in the trace size, which can grow to *exascale* for large programs such as Apache Derby[1], the traces of which contain tens of millions of critical events [18]. We provide an illustrative example for RVPredict in Sect. 2.

To improve the scalability of symbolic analysis for analyzing concurrent programs, we need highly efficient constraint solvers. Fortunately, we note that the symbolic constraints in many problems [9,16–18,20,23,25] take a simple form. Each constraint consists of conjunctions and disjunctions of many simple Boolean expressions over atomic predicates which are just simple *ordering* comparisons. An example is: $O_1 < O_2 \wedge O_3 < O_4 \wedge (O_2 < O_3 \vee O_4 < O_1)$. Here each variable $O_i$ denotes the occurrence of an event; and the relation $O_i < O_j$ means that event $e_i$ happens before event $e_j$ in certain schedules. A constraint like this is called an *ordering constraint* (OC). The relational operator could also be $\leq$, $\geq$, etc. However, the specific value difference between variables is irrelevant, because in many applications we do not concern about the real-time properties among events. Therefore, to solve ordering constraints, it is not necessary to use the full (integer) difference logic (DL), which is the most efficient decision procedure used by existing solvers for OC.

In this paper, we study properties and decision procedures for ordering constraints (OCs). The theory of ordering constraints is a fragment of difference logic, which can be decided by detecting negative cycles in the weighted digraph. However, we find that detecting negative cycles is not essential to the consistency checking of ordering constraints. In fact, the problem is closely related to the decomposition of a digraph into its strongly connected components. Based on Tarjan's strongly connected components algorithm, we propose a linear time decision procedure for checking satisfiability of ordering constraints, and investigate how to integrate it with the DPLL(T) framework. We have also

[1] http://db.apache.org/derby/.

```
  initially x=y=0  resource z=0
Thread t1            Thread t2
1.  fork t2
2.  lock l
3.  x = 1
4.  y = 1
5.  unlock l
                     6.  {      //begin
                     7.    lock l
                     8.    r1 = y
                     9.    unlock l
                     10.   r2 = x
                     11.   if(r1 == r2)
                     12.   z = 1    (auth)
                     13. }      //end
14. join t2
15. r3 = z    (use)
16. if(r3 == 0)
17.   Error
```

**Fig. 1.** An example with a race (3,10).

```
      initially   x = y = z = 0
1.  fork(t1, t2)
2.  lock(t1, l)
3.  write(t1, x, 1)
4.  write(t1, y, 1)
5.  unlock(t1, l)
                        6.  begin(t2)
                        7.  lock(t2, l)
                        8.  read(t2, y, 1)
                        9.  unlock(t2, l)
                        10. read(t2, x, 1)
                        11. branch(t2)
                        12. write(t2, z, 1)
                        13. end(t2)
14. join(t1, t2)
15. read(t1, z, 1)
16. branch(t1)
```

**Fig. 2.** A trace corresponding to the example

developed a customized solver for SMT(OC), and conducted extensive evaluation of its performance compared with two state-of-the-art SMT solvers, Z3 [5] and OpenSMT [3], on both benchmarks and real symbolic constraints from RVPredict. Though not optimized, our tool achieves comparable performance as that of Z3 and OpenSMT both of which are highly optimized. We present our experimental results in Sect. 6.

The rest of the paper is organized as follows. We first provide a motivating example to show how ordering constraints are derived from symbolic analysis of concurrent programs in Sect. 2. We then formally define ordering constraints and the constraint graph in Sect. 3 and present a linear time decision procedure for OC in Sect. 4. We further discuss how to integrate the decision procedure with the DPLL(T) framework to solve SMT(OC) formulas in Sect. 5.

## 2 Motivation

To elucidate the ordering constraints, let's consider a data race detection problem based on the symbolic analysis proposed in RVPredict [18].

The program in Fig. 1 contains a race condition between lines (3,10) on a shared variable $x$ that may cause an authentication failure of resource $z$ at line 12, which in consequence causes an error to occur when $z$ is used at line 15. Non-symbolic analysis techniques such as happens-before [10], causal-precedes [28], and the lockset algorithm [19, 26] either cannot detect this race or report false alarms. RVPredict is able to detect this race by observing an execution trace of the program following an interleaving denoted by the line numbers (which does not manifest the race). The trace (shown in Fig. 2) contains a sequence of events emitted in the execution, including thread *fork* and *join*, *begin* and *end*, *read* and *write*, *lock* and *unlock*, as well as *branch* events.

The constructed symbolic constraints (shown in Fig. 3) based on the trace consist of three parts: (A) the must happen-before (MHB) constraints, (B) the locking constraints, and (C) the race constraints. The MHB constraints encode the ordering requirements among events that must always hold. For example, the *fork* event at line 1 must happen before the *lock* event at line 2 and the *begin* event of $t2$ at line 6, so we have $O_1 < O_2$ and $O_1 < O_6$. The locking constraints encode lock mutual exclusion consistency over *lock* and *unlock* events. For example, $O_5 < O_7 \vee O_9 < O_2$ means that either $t1$ acquires the lock $l$ first and $t2$ second, or $t2$ acquires $l$ first and $t1$ second. If $t1$ first, then the *lock* at line 7 must happen after the *unlock* at line 5; otherwise if $t2$ first, the *lock* at line 2 should happen after the *unlock* at line 9.

The race constraints encode the data race condition. For example, for (3,10), the race constraint is written as $O_{10} = O_3$, meaning that these two events are un-ordered. For (12,15), because there is a *branch* event (at line 11) before line 12, the control-flow condition at the *branch* event needs to be satisfied as well. So the race constraint is written as $O_{10} = O_3 \wedge O_3 < O_{10} \wedge O_4 < O_8$, to ensure that the *read* event at line 10 reads value 1 on $x$, and that the *read* event at line 8 reads value 1 on $y$. The size of symbolic constraints, in the worst case, is cubic in the number of reads and writes in the trace.

Putting all these constraints together, the technique then invokes a solver to compute a solution for these unknown order variables. For (3,10), the solver returns a solution which corresponds to the interleaving 1-6-7-8-9-2-3-10, so (3,10) is a race. For (12,15), the solver reports no solution, so it is not a race.

The symbolic constraints above are easy to solve, since the size of the trace is small in this simple example. However, for real world programs with long running executions, the constraints can quickly exceed the capability of existing solvers such as Z3 [5] as the constraint size is cubic in the trace size. As a result, RVPredict has to cut the trace into smaller chunks and only detects races in each chunk separately, resulting in missing races across chunks. Hence, to scale RVPredict to larger traces and to find more races, it is important to design more efficient solvers that are customized for solving the ordering constraints. Although we focus on motivating this problem with RVPredict, the ordering constraints are applicable to many other concurrency analysis problems such as replay [23], failure reproduction [20], concurrency property violation detection [9,17], model checking [16], etc.

We next formalize the ordering constraints and present our algorithm to solve this problem with a linear time decision procedure.

## 3 Preliminaries

**Definition 1.** *An **ordering constraint** (OC) is a comparison between two numeric variables. It can be represented as* $(x \ op \ y)$*, where* $op \in \{<, \leq, >, \geq, =, \neq\}$.

| | |
|---|---|
| A. MHB | $O_1 < O_2 < \ldots < O_5$ |
| | $O_{14} < \ldots < O_{16}$ |
| | $O_6 < O_7 < \ldots < O_{13}$ |
| | $O_1 < O_6 \wedge O_{13} < O_{14}$ |
| B. Locking | $O_5 < O_7 \vee O_9 < O_2$ |
| C. Race (3,10) | $O_{10} = O_3$ |
| Race (12,15) | $O_{15} = O_{12}$ |
| | $O_3 < O_{10} \wedge O_4 < O_8$ |

**Fig. 3.** Symbolic constraints of the trace

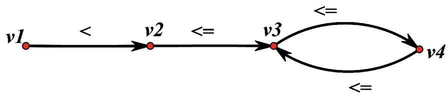


**Fig. 4.** Example 1

The theory of ordering constraints is a special case of difference logic, where the constant $c$ in the difference theory atom $((x - y)\ op\ c)$ is restricted to 0.

**Definition 2.** *An SMT formula $\phi$ over ordering constraints, i.e., an* ***SMT(OC) formula****, can be represented as a Boolean formula $PS_\phi(b_1, \ldots, b_n)$ together with definitions in the form: $b_i \equiv x\ op\ y$, where $op \in \{<, \leq, >, \geq, =, \neq\}$. That means, the Boolean variable $b_i$ stands for the ordering constraint $(x\ op\ y)$. $PS_\phi$ is the* propositional skeleton *of the formula $\phi$.*

Without loss of generality, we can restrict the relational operators to $<$ and $\leq$. In other words, the problem at hand is a Boolean combination of atoms of the form $x < y$ or $x \leq y$.

A set of ordering constraints can be naturally represented with a directed graph.

**Definition 3.** *Given a set of ordering constraints, the* ***constraint graph of the ordering constraints*** *is a digraph $G = \{V, E\}$ which is constructed in the following way:*

1. *For each variable $x_i$, introduce a vertex $v_i \in V$.*
2. *For each constraint $x_i < x_j$, introduce an edge $e^{<}_{i,j} \in E$ from $v_i$ to $v_j$.*
3. *For each constraint $x_i \leq x_j$, introduce an edge $e^{\leq}_{i,j} \in E$ from $v_i$ to $v_j$.*

**Definition 4.** *The* ***out-degree*** *of a vertex $v$ of digraph $G$ is the number of edges that start from $v$, and is denoted by* $\boldsymbol{outdeg}(\boldsymbol{v})$*. Similarly, the* ***in-degree*** *of $v$ is the number of edges that end at $v$, and is denoted by* $\boldsymbol{indeg}(\boldsymbol{v})$*.*

*Example 1.* Consider a set of ordering constraints: $\{x_1 < x_2,\ x_2 \leq x_3,\ x_3 \leq x_4, x_4 \leq x_3\}$. Figure 4 shows the constraint graph constructed by Definition 3. The variables $\{x_1,\ x_2,\ x_3,\ x_4\}$ are represented by the nodes $\{v_1,\ v_2,\ v_3,\ v_4\}$, respectively, and $\text{outdeg}(x_3) = 1$ and $\text{indeg}(x_3) = 2$.

Recall that difference logic also has a graph representation. A set of difference arithmetic atoms can be represented by a weighted directed graph, where each node corresponds to a variable, and each edge with weight corresponds to a difference arithmetic atom. Obviously the constraint graph of ordering constraints can be viewed as a special case of that of difference logic, where all weights can only take two values. The distinction between ordering constraints and difference logic seems to be slight. However, in the rest of the paper we will show how this minor difference leads to a new decision procedure with lower time complexity.

## 4 The Decision Procedure for Ordering Constraints

It is well known that DL can be decided by detecting negative cycles in the weighted directed graph with the Bellman-Ford algorithm [24]. The complexity of the classical decision procedure for DL is $O(nm)$, where $n$ is the number of variables, and $m$ is the number of constraints. As a fragment of difference logic, ordering constraints can be directly checked with the aforementioned algorithm. However, through exploring the structure of the constraint graph of ordering constraints, we observe that detecting negative cycles is not essential to the consistency checking of OC. In this section, we propose a new way to check the inconsistency of OC, which needs only to examine the constraint graph in linear time.

Before presenting the decision procedure for OC, we first introduce some theoretical results on OC and its constraint graph.

**Lemma 1.** *If digraph $G$ has no cycle, then $G$ has a vertex of out-degree 0 and a vertex of in-degree 0.*

*Proof.* We prove this lemma via reduction to absurdity. Assume for each vertex $v$ of $G$, outdeg$(v) > 0$. Let $v_1$ be a vertex in $V$. Since outdeg$(v_1) > 0$ by the assumption, there exists an edge $e_1$ which starts from $v_1$ and ends at $v_2$. Since outdeg$(v_2) > 0$, there exists an edge $e_2$ which starts from $v_2$ and ends at $v_3$, and so on and so forth. In this way, we obtain an infinite sequence of vertices $\{v_1, \ldots, v_k, \ldots\}$. Note that $|V|$ is finite, there must exist a cycle in this sequence, which contradicts the precondition that $G$ has no cycle. The proof of case of in-degree is analogous.

**Lemma 2.** *Given a set of ordering constraints $\alpha$, if its constraint graph $G$ has no cycle, then $\alpha$ is consistent.*

*Proof.* Based on the acyclic digraph $G$, we construct a feasible solution to the variables of $\alpha$ in the following way:

(1) Set $i = 0$, and $G_0 = G$.
(2) Find the set $V_i'$ of vertices of in-degree 0 in $G_i = (V_i, E_i)$. For each vertex $v_t$ in $V_i'$, let the corresponding variable $x_t = i$.
(3) Let $E_i' = \{e | e \in E_i$ and $e$ starts from a vertex in $V_i'\}$. Construct the subgraph $G_{i+1}$ of $G_i$ by $G_{i+1} = (V_{i+1}, E_{i+1}) = (V_i - V_i', E_i - E_i')$.
(4) Repeat step (2) and (3) until $G_i$ is empty.

We now show that this procedure terminates with a solution that satisfies $\alpha$. Note that $G$ is acyclic and each $G_i$ is a subgraph of $G$, so $G_i$ is acyclic. According to Lemma 1, we have $|V_i'| > 0$ every time the iteration reaches step (2). Therefore, this procedure will terminate.

Consider two adjacent vertices $v_p$ and $v_q$ with an edge $\langle v_p, v_q \rangle$. As long as $v_p$ remains in the current graph $G_i$, indeg$(v_q) > 0$. Hence $v_p$ must be deleted earlier than $v_q$, and we have $x_p < x_q$. In general, for an arbitrary pair of vertices ($v_p$ and $v_q$), if there exists a path from $v_p$ to $v_q$, namely $\langle v_p, v_{p_1}, \ldots, v_{p_k}, v_q \rangle$, then we have $x_p < x_{p_1} < \cdots < x_{p_k} < x_q \Rightarrow x_p < x_q$.

**Theorem 1.** *Given a set of ordering constraints $\alpha$ and its constraint graph $G$, $\alpha$ is inconsistent if and only if there exists a maximal strongly connected component of $G$ that contains an $e^<$ edge.*

*Proof.* $\Longleftarrow$ Let $G'$ be a maximal strongly connected component of $G$ which contains an $e^<$ edge $\langle v_1, v_2 \rangle$. Since $v_1$ and $v_2$ are reachable from each other, there exists a path from $v_2$ to $v_1$ in $G'$. Without loss of generality, we assume the path is $\{v_2, \ldots, v_n, v_1\}$. The path and the edge $\langle v_1, v_2 \rangle$ form a cycle in $G'$, which implies that $x_1 < x_2 \leq \cdots \leq x_n \leq x_1$. Thus $x_1 < x_1$, and $\alpha$ is inconsistent.

$\Longrightarrow$ We prove this via reduction to absurdity. Suppose every maximal strongly connected component of $G$ does not contain an $e^<$ edge. Consider an arbitrary pair of vertices $v_p$ and $v_q$ that are reachable from each other. Since $v_p$ and $v_q$ belong to a maximal strongly connected component, there only exist $e^\leq$ edges in the path from $v_p$ to $v_q$, then $x_p \leq x_q$. On the other hand, we have $x_p \geq x_q$. As a result, $x_p = x_q$. Let $G_s = (V_s, E_s)$ be a maximal strongly connected component of $G$. We could merge vertices of $V_s$ into one vertex $v$ and obtain a new graph $G' = (V', E')$, where $V' = (V - V_s) \cup \{v\}$ and $E' = \{\langle v_i, v_j \rangle | \langle v_i, v_j \rangle \in E, v_i \notin V_s, v_j \notin V_s\} \cup \{\langle v, v_j \rangle | \langle v_i, v_j \rangle \in E, v_i \in V_s, v_j \notin V_s\} \cup \{\langle v_i, v \rangle | \langle v_i, v_j \rangle \in E, v_i \notin V_s, v_j \in V_s\}$. In addition, $x = x_i, \forall v_i \in V_s$. Consider the following way to construct a solution to $\alpha$. For each maximal strongly connected component of $G$, we merge it into a vertex and finally obtain $G' = (V', E')$. Note that such $G'$ is unique and acyclic. We could construct a solution from $G'$ by Lemma 2.

We now show the solution constructed by this procedure satisfies $\alpha$. That is, for each pair of vertices $(v_p, v_q)$, if there exists a path from $v_p$ to $v_q$, then $x_p \leq x_q$. Furthermore, if there exists an $e^<$ edge in a path from $v_p$ to $v_q$, then $x_p < x_q$. Let $v_p$ and $v_q$ map to $v_p'$ and $v_q'$ of $G'$. If $v_p' = v_q'$, then $x_p = x_p' = x_q' = x_q$. Otherwise, there exists a path from $v_p'$ to $v_q'$. By Lemma 2, $x_p = x_p' < x_q' = x_q$. Hence $x_p \leq x_q$ always holds. If there exists an $e^<$ edge in a path from $v_p$ to $v_q$, then $v_p$ and $v_q$ cannot be in the same maximal strongly connected component. Therefore, $v_p' \neq v_q' \Rightarrow x_p < x_q$. It can be concluded that $\alpha$ is consistent since the solution satisfies the constraints of $\alpha$.

*Example 2.* Recall in Example 1 that there are 3 strongly connected components $\{\{v_1\},\{v_2\},\{v_3,v_4\}\}$. If we add a constraint $x_3 \leq x_1$, the resulting constraint graph is shown in Fig. 5. There is only one strongly connected component, which

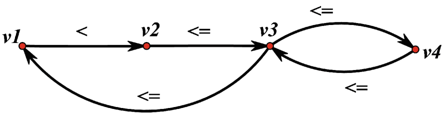


**Fig. 5.** Example 2

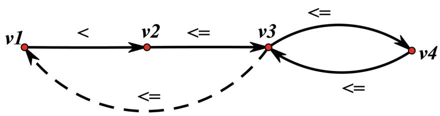


**Fig. 6.** Example 3

itself is a connected graph. Since $\langle v_1, v_2\rangle$ is an $e^<$ edge, the conjunction of ordering constraints is inconsistent by Theorem 1. The conflict $x_1 < x_1$ can be drawn from $\{x_1 < x_2, x_2 \leq x_3, x_3 \leq x_1\}$.

Theorem 1 suggests that, to check the consistency of ordering constraints, we can decompose its constraint graph into maximal strongly connected components and then examine the edges. We use Tarjan's algorithm [29] to find the maximal strongly connected components in our ordering constraints theory solver. It produces a unique partition of the graph's vertices into the graph's strongly connected components. Each vertex of the graph appears in exactly one of these components. Then we check each edge in these components whether it is an $e^<$ edge. Therefore the consistency of conjunctions of ordering constraints can be decided in $O(n + m)$ time.

## 5 Integrating $DP_{OC}$ into DPLL(T)

### 5.1 The DPLL(T) Framework

DPLL(T) is a generalization of DPLL for solving a decidable first order theory $T$. The DPLL(T) system consists of two parts: the global DPLL(X) module and a decision procedure $DP_T$ for the given theory $T$. The DPLL(X) part is a general DPLL engine that is independent of any particular theory $T$ [13]. It interacts with $DP_T$ through a well-defined interface. The DPLL(T) framework is illustrated in Fig. 7. We assume that the readers are familiar with DPLL components, such as `Decide`, `BCP`, `Analyze` and `Backtrack`. The component `TP` represents theory propagation, which is invoked when no more implications can be made by `BCP`. It deduces literals that are implied by the current assignment in theory $T$, and communicates the implications to the BCP part. Although theory propagation is not essential to the functionality of the solving procedure, it is vital to the efficiency of the procedure. The component `Check` encapsulates the decision procedure $DP_T$ for consistency checking of the current assignment. If inconsistencies are detected, it generates theory-level minimal conflict clauses.

### 5.2 Theory-Level Lemma Learning

We now discuss how to integrate the decision procedure $DP_{OC}$ into the DPLL(T) framework. In DPLL(T), the decision procedure is called repeatedly to check the consistency of (partial) assignments. To avoid frequent construction/destruction of constraint graphs, at the beginning of the solving process, we construct the

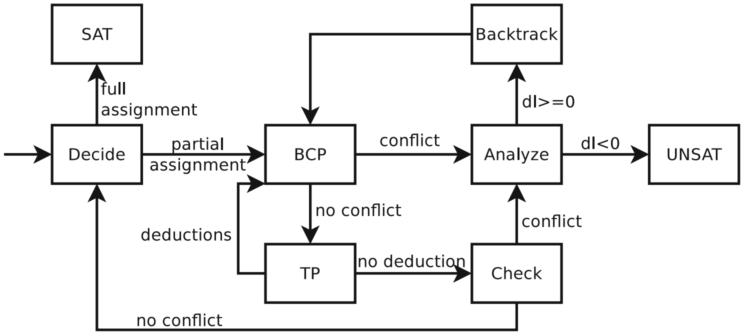


**Fig. 7.** The DPLL(T) Framework

constraint graph $G$ of the set of all predicates in the target SMT(OC) formula. In this graph, each edge has two states: an edge is **active** if its corresponding boolean variable is assigned a value (true, false); and is **inactive** if its corresponding boolean variable is undefined.

Notice that initially all edges are inactive. When the solver finds a partial assignment $\alpha$, the edges in $G$ corresponding to $\alpha$ are activated. Hence the constraint graph $G_\alpha$ of the ordering constraints of $\alpha$ consists of every active edge in $G$, and is a subgraph of $G$. The decision procedure $DP_{OC}$ checks the consistency of $\alpha$ based on $G_\alpha$.

*Example 3.* Consider a formula $PS_\phi(b_1, b_2, b_3, b_4, b_5) = (b_1 \wedge (\neg b_2) \wedge (b_3 \vee b_4 \vee b_5))$, $\{b_1 \equiv x_1 < x_2,\ b_2 \equiv x_3 < x_2,\ b_3 \equiv x_3 \leq x_4, b_4 \equiv x_4 \leq x_3, b_5 \equiv x_3 \leq x_1\}$. Figure 6 shows the constraint graph $G_\beta$ of all predicates in this formula with a possible partial assignment $\beta$, $\{b_1 = True, b_2 = False, b_3 = True, b_4 = True, b_5 = Undefined\}$. Note that $\{\langle v_1, v_2\rangle, \langle v_2, v_3\rangle, \langle v_3, v_4\rangle, \langle v_4, v_3\rangle\}$ are active and $\langle v_3, v_1\rangle$ is inactive. Actually, the graph of Example 1 is a subgraph of $G_\beta$, which can be constructed by choosing all active edges in $G_\beta$.

To maximize the benefits of integration, the OC solver should be able to communicate theory lemmas to the SAT engine, including conflict clauses and deduction clauses at the OC theory level. We next discuss two such techniques.

**Minimal Conflict Explanation.** According to Theorem 1, the OC solver detects an inconsistency of the current assignment if it finds an $e^<$ edge in a strongly connected component of the constraint graph $G$. Without loss of generality, we assume the $e^<$ edge is $e = \langle v_1, v_2\rangle$, and denote the strongly connected component by $G'$. The inconsistency is essentially caused by a cycle that contains $e$. Note that all paths from $v_2$ to $v_1$ are in $G'$. Hence we only have to find a shortest path from $v_2$ to $v_1$ in $G'$ instead of $G$. The shortest path from $v_2$ to $v_1$ and the edge $e = \langle v_1, v_2\rangle$ form a shortest cycle with an $e^<$ edge, corresponding to the minimal conflict that gives rise to the inconsistency. Therefore, we generate theory-level conflict clauses according to such cycles.

**Algorithm 1.** Tarjan's Algorithm Combined With Theory Propagation

```
Tarjan() initialize v, S, index, scc;
for each v that v.index is undefined in V do
    Tarjan_DFS(v);

Tarjan_DFS(v) v.index, v.lowlink ← index, index ← index + 1, S.push(v);
for each active edge ⟨v, w⟩ in E do
    if w is not visited then
        w.father ← v;
        if ⟨v, w⟩ is an e^< edge then
            w.nf ← v.nf + 1;
        else
            w.nf ← v.nf;
        Tarjan_DFS(w);
        v.lowlink ← min(v.lowlink, w.lowlink);
    else if w in S then
        v.lowlink ← min(v.lowlink, w.index);

if v.lowlink = v.index then
    repeat
        s, t ← S.pop(), s.scc ← scc;
        while t.father is defined do
            t ← t.father;
            if (⟨s, t⟩ or ⟨t, s⟩ is inactive) and (s.nf > t.nf or ⟨s, t⟩ is an e^<
            edge) then
                generate TP clause from s to t by father vertex records;
    until (s = v);
    scc ← scc + 1;
```

**Theory Propagation.** In order to improve performance, we apply a "cheap" theory propagation technique. Our theory propagation is combined with the consistency check to reduce its cost. However, it is an incomplete algorithm.

Algorithm 1 is the pseudocode of the whole consistency check procedure. It is mainly based on the Tarjan algorithm on the graph $G' = (V, active(E))$. Like the original Tarjan algorithm, the *index* variable counts the number of visited nodes in DFS order. The value of $v$.index numbers the nodes consecutively in the order in which they are discovered. And the value of $v$.lowlink represents the smallest index of any node known to be reachable from $v$, including $v$ itself. The *scc* variable counts the number of strongly connected components. And the attribute scc of a vertex records the strongly connected component it belongs to. $S$ is the node stack, which stores the history of nodes explored but not yet committed to a strongly connected component.

We introduce two values for a vertex $v$, $v$.father and $v$.nf, for theory propagation. The value of $v$.father represents a vertex $w$, that the DFS procedure visits $v$ through edge $\langle w, v\rangle$. Assume the DFS procedure starts from vertex $u$. Then

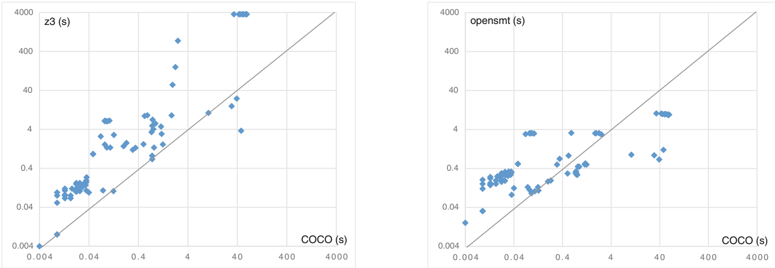


**Fig. 8.** Experiments on instances generated from RVPredict

we can generate a path from $u$ to $v$ by retrieving the father attribute of each vertex on this path from $v$. The number of $e^<$ edges on this path is recorded by $v$.nf. We add two parts into the original Tarjan algorithm. In Algorithm 1, the statements from line 7 to line 12 record the "father" and the "nf" attribute of $w$. The loop from line 23 to line 27 recursively checks the vertex $t$ by retrieving father records from $s$. We can obtain a path $p_{ts}$ from $t$ to $s$ in this way. If $t$.nf<$s$.nf, there exists at least one $e^<$ edge on this path. Thus $p_{ts}$ and edge $st$ compose a negative cycle if $t$.nf<$s$.nf or $st$ is an $e^<$ edge. We can determine the assignment of the Boolean variable which corresponds to the edge $ts$ or $st$ and generate the Boolean clause of this deduction.

In Example 3, our algorithm starts from $v_1$, and then applies a DFS procedure. When the algorithm visits the last vertex, $v_4$, we have $v_4.\mathrm{nf} = v_3.\mathrm{nf} = v_2.\mathrm{nf} = v_1.\mathrm{nf} + 1$. Then the algorithm starts popping stack $S$ and constructing strongly connected components. At vertex $v_3$, we find $v_1$ is the father of $v_3$.father, $\langle v_3, v_1\rangle$ is inactive and $v_3.\mathrm{nf} > v_1.\mathrm{nf}$, so we deduce that $b_5 \equiv \langle v_3, v_1\rangle$ should be $False$ and generate a clause, $(\neg b_1) \vee b_2 \vee (\neg b_5)$.

## 6 Experimental Evaluation

We have implemented our decision procedure in a tool called `COCO` (which stands for **C**ombating **O**rdering **CO**nstraints) based on MiniSat 2.0[2]. We have evaluated `COCO` with a collection of ordering constraints generated from RVPredict and two series of QF_IDL benchmarks (`diamonds` and `parity`) in SMT-Lib[3], which are also SMT(OC) formulas. The experiments were performed on a workstation with 3.40 GHz Intel Core i7-2600 CPU and 8 GB memory. For comparison, we also evaluated with two other state-of-the-art SMT solvers, i.e., `OpenSMT`[4] and `Z3`[5]. The experimental results are shown in Figs. 8 and 9. Note that each point represents an instance. Its $x$-coordinate and $y$-coordinate represent the running

[2] N. Eén and N. Sörensson. The `MiniSat` Page. http://minisat.se/.
[3] They are available at: http://www.cs.nyu.edu/~barrett/smtlib/.
[4] The `OpenSMT` Page. http://code.google.com/p/opensmt/.
[5] The `Z3` Page. http://z3.codeplex.com/.

**Table 1.** More details about the "Hard" instances.

| Instance | | OpenSMT | | | COCO | | | Z3 |
|---|---|---|---|---|---|---|---|---|
| Name | Dims | TS sat calls | TS unsat calls | Time(s) | TS sat calls | TS unsat calls | Time(s) | Time(s) |
| Harness_1 | 19783 | 40460 | 1 | 9.489 | 21664 | 12775 | 59.768 | — |
| Harness_2 | 19783 | 41278 | 1 | 9.929 | 18703 | 12011 | 50.937 | — |
| JigsawDriver_3 | 1548 | 5796 | 0 | 0.892 | 12797 | 15604 | 10.447 | 10.549 |
| JigsawDriver_7 | 1548 | 6198 | 0 | 0.848 | 997 | 1671 | 0.538 | 8.813 |
| BubbleSort_3 | 1195 | 36989 | 71 | 0.868 | 47643 | 52508 | 30.708 | 15.761 |
| JGFMolDynA_1 | 7718 | 11448 | 0 | 3.028 | 3 | 17 | 0.074 | 2.64 |
| JGFMolDynA_2 | 7718 | 12914 | 4 | 2.972 | 2214 | 3181 | 2.522 | 748.207 |
| BoundedBuffer_39 | 828 | 5640 | 1 | 0.500 | 787 | 1109 | 0.312 | 1.196 |
| BoundedBuffer_40 | 828 | 11464 | 47 | 0.444 | 2621 | 2924 | 0.830 | 1.360 |
| BoundedBuffer_41 | 828 | 5537 | 1 | 0.500 | 3256 | 3327 | 1.252 | 1.640 |
| main_15 | 9707 | 12882 | 1 | 3.228 | 2132 | 2122 | 2.184 | 158.214 |

times of COCO and Z3/OpenSMT on this instance, respectively. All figures are in logarithmic coordinates.

Figure 8 shows the results on instances that are generated from RVPredict. Our tool performs well on some small instances. It takes dozens of milliseconds for COCO to solve them. Z3 usually consumes more time and memory than COCO, and it fails to solve some large instances, due to the limit on memory usage. For such instances, we regard the running time of Z3 as more than 3600 s. Nevertheless, on some larger instances OpenSMT is more efficient. Our investigation of OpenSMT reveals that it adopts an efficient incremental consistency checking algorithm and integrates minimal conflict with a theory propagation technique, which COCO currently does not fully support. The advantage of theory propagation is that it allows the solver to effectively learn useful facts that can help reduce the chances of conflicts. On the instances generated from RVPredict, theory propagations are very effective, because the Boolean structures of the SMT(OC) formulas are quite simple.

Table 1 gives more details on some "hard" instances in Fig. 8. "TS sat calls" and "TS unsat calls" represent the number of satisfiable/unsatisfiable calls of the theory solver, respectively. "Dims" denotes the number of numeric variables, i.e., dimension of the search space. The running times of both OpenSMT and COCO are closely related to the dimension of the instance and the number of calls of the theory solver. An unsatisfiable call of the theory solver causes backtracking and retrieving reasons; so it consumes much more time than a satisfiable call. Notice that OpenSMT hardly encounters unsatisfiable calls. Its theory propagation procedure greatly reduces the number of unsatisfiable calls. On the contrary, COCO even encounters more unsatisfiable calls than satisfiable calls in some circumstances, because its theory propagation is incomplete.

Figure 9 shows the experimental results on SMT-Lib benchmarks "diamonds" and "parity". It appears that OpenSMT is often slower than COCO, and Z3 performs well in these cases, in contrast to Fig. 8. OpenSMT only applies the incremental algorithm which cannot skip steps, so it checks consistency incrementally whenever it makes decision or propagation. On instances that contain complicated Boolean components, like some SMT-Lib benchmarks, OpenSMT is not so efficient, because it has to backtrack often and applies the consistency checking algorithm step by step again even with complete theory propagations. On the other hand, Z3 tightly integrates many strategies, some of which are handcrafted and fall outside the scope of DPLL(T), such as formula preprocessing, which COCO does not implement. These may be the reasons for the good performance of Z3 in Fig. 9.

In addition to the running time, we also compared the memory usage of these three solvers. It turned out that COCO always occupies the least memory. The memory usage of OpenSMT is about 5 to 10 times as much as that of COCO, and Z3 consumes tens of times even hundreds of times higher memories than COCO. The detailed data are omitted, due to the lack of space.

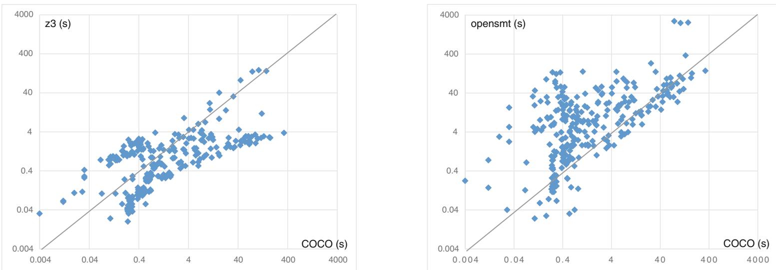


**Fig. 9.** Experiments on QF_IDL benchmarks in SMT-Lib

To summarize, COCO achieves better scalability than Z3 on the real instances generated by RVPredict. On the other hand, when comparing COCO with OpenSMT, there seems no clear winner. The incremental decision procedure with complete theory propagation enables OpenSMT to perform well on many instances generated by RVPredict, whereas it results in poor performance of OpenSMT on the classical SMT-Lib instances. Besides, our current tool has potential to achieve better performance as we have not designed a complete theory propagation, as demonstrated by OpenSMT, and many other optimization strategies used by Z3.

## 7 Related Work

As we mentioned earlier, there has been a large body of work on solving (integer) difference constraints. See, for example, [4,12,22,24]. Nieuwenhuis and Oliveras presented a DPLL(T) system with exhaustive theory propagation for solving SMT(DL) formulas [24]. They reduced the consistency checking for DL to detecting negative cycles in the weighted digraph with the Bellman-Ford algorithm [24].

The complexity of this decision procedure is $O(nm)$, where $n$ is the number of variables, and $m$ is the number of constraints. In [4] Cotton and Maler proposed an incremental complete difference constraint propagation algorithm with complexity $O(m + nlogn + |U|)$, where $|U|$ is the number of constraints which are candidates for being deduced. However, to check the consistency of conjunctions of constraints, the incremental algorithm has to be called for each constraint. Therefore, the complexity of the whole procedure is even higher. In contrast, the complexity of our decision procedure for ordering constraints is only $O(n + m)$.

Besides, there are some works consider extending a SAT solver with acyclicity detection. [21] deals with a conjunction of theory predicates, while our work is concerned with arbitrary Boolean combinations of ordering constraints. Due to the existence of the logical connectives (OR, NOT) of SMT(OC) formulas, the equality and disequality relations can be represented by inequality relations. We only have to consider two types of edges ($e^{>=}$ edge and $e^{>}$ edge) in our graph, which is more simple than four types of edges in [21]. Moreover, our theory propagation exploits the information from Tarjans algorithm. [14,15], and recent versions of MonoSAT [2] all rely on similar theory propagation and clause learning techniques. [2], for example, also uses Tarjan's SCC during clause learning in a similar way as this paper. However, they don't have a notion of $e^{<}$ edges versus $e^{<=}$ edges, and they couldn't support distinction of $e^{<}$ edges versus $e^{<=}$ edges without significant modifications.

## 8 Conclusion

Satisfiability Modulo Theories (SMT) is an important research topic in automated reasoning. In this paper, we identified and studied a useful *theory*, i.e., the theory of ordering constraints. We demonstrated its *applications* in symbolic analysis of concurrent programs. We also presented *methods* for solving the related satisfiability problems. In particular, we gave a decision procedure that has a lower complexity than that for the difference logic. We have also implemented a prototype tool for our algorithm and compared its performance with two state-of-the-art SMT solvers, Z3 and OpenSMT. Although our current implementation is not optimized, it achieves comparable performance as that of Z3 and OpenSMT which have been developed for years and are highly optimized. We explained why a particular tool is more efficient on certain problem instances. In our future work, we plan to further improve the performance of our approach by developing incremental and backtrackable decision procedures with more efficient theory propagation.

## References

1. Barrett, C., Conway, C.L., Deters, M., Hadarean, L., Jovanović, D., King, T., Reynolds, A., Tinelli, C.: CVC4. In: Gopalakrishnan, G., Qadeer, S. (eds.) CAV 2011. LNCS, vol. 6806, pp. 171–177. Springer, Heidelberg (2011)

2. Bayless, S., Bayless, N., Hoos, H.H., Hu, A.J.: SAT modulo monotonic theories. In: AAAI (2015)
3. Bruttomesso, R., Pek, E., Sharygina, N., Tsitovich, A.: The OpenSMT solver. In: Esparza, J., Majumdar, R. (eds.) TACAS 2010. LNCS, vol. 6015, pp. 150–153. Springer, Heidelberg (2010)
4. Cotton, S., Maler, O.: Fast and flexiable difference constraint propagation for DPLL(T). In: SAT (2006)
5. de Moura, L., Bjørner, N.S.: Z3: An efficient SMT solver. In: Ramakrishnan, C.R., Rehof, J. (eds.) TACAS 2008. LNCS, vol. 4963, pp. 337–340. Springer, Heidelberg (2008)
6. Dutertre, B., De Moura, L.: The Yices SMT solver. Technical report (2006)
7. Eslamimehr, M., Palsberg, J. Sherlock: Scalable deadlock detection for concurrent programs. In: FSE (2014)
8. Farzan, A., Holzer, A., Razavi, N., Veith, H.: Con2colic testing. In: ESEC/FSE (2013)
9. Farzan, A., Madhusudan, P., Razavi, N., Sorrentino, F.: Predicting null-pointer dereferences in concurrent programs. In: FSE (2012)
10. Flanagan, C., Freund, S.N.: Fasttrack: efficient and precise dynamic race detection. In: PLDI (2009)
11. Ganesh, V., Dill, D.L.: A decision procedure for bit-vectors and arrays. In: Damm, W., Hermanns, H. (eds.) CAV 2007. LNCS, vol. 4590, pp. 519–531. Springer, Heidelberg (2007)
12. Gange, G., Søndergaard, H., Stuckey, P.J., Schachte, P.: Solving difference constraints over modular arithmetic. In: Bonacina, M.P. (ed.) CADE 2013. LNCS, vol. 7898, pp. 215–230. Springer, Heidelberg (2013)
13. Ganzinger, H., Hagen, G., Nieuwenhuis, R., Oliveras, A., Tinelli, C.: DPLL($T$): fast decision procedures. In: Alur, R., Peled, D.A. (eds.) CAV 2004. LNCS, vol. 3114, pp. 175–188. Springer, Heidelberg (2004)
14. Gebser, M., Janhunen, T., Rintanen, J.: SAT modulo graphs: acyclicity. In: Fermé, E., Leite, J. (eds.) JELIA 2014. LNCS, vol. 8761, pp. 137–151. Springer, Heidelberg (2014)
15. Gebser, M., Janhunen, T., Rintanen, J.: Answer set programming as SAT modulo acyclicity. In: ECAI (2014)
16. Huang, J.: Stateless model checking concurrent programs with maximal causality reduction. In: PLDI (2015)
17. Huang, J., Luo, Q., Rosu, G.: Gpredict: Generic predictive concurrency analysis. In: ICSE (2015)
18. Huang, J., Meredith, P.O., Rosu, G.: Maximal sound predictive race detection with control flow abstraction. In: PLDI (2014)
19. Huang, J., Zhang, C.: PECAN: persuasive prediction of concurrency access anomalies. In: ISSTA (2011)
20. Huang, J., Zhang, C., Dolby, J.: CLAP: Recording local executions to reproduce concurrency failures. In: PLDI (2013)
21. Kroening, D., Weissenbacher, G.: An interpolating decision procedure for transitive relations with uninterpreted functions. In: Namjoshi, K., Zeller, A., Ziv, A. (eds.) HVC 2009. LNCS, vol. 6405, pp. 150–168. Springer, Heidelberg (2011)
22. Kim, H., Somenzi, F.: Finite instantiations for integer difference logic. In: FMCAD (2006)
23. Lee, D., Said, M., Narayanasamy, S., Yang, Z., Pereira, C.: Offline symbolic analysis for multi-processor execution replay. In: MICRO (2009)

24. Nieuwenhuis, R., Oliveras, A.: DPLL(T) with exhaustive theory propagation and its application to difference logic. In: Etessami, K., Rajamani, S.K. (eds.) CAV 2005. LNCS, vol. 3576, pp. 321–334. Springer, Heidelberg (2005)
25. Said, M., Wang, C., Yang, Z., Sakallah, K.: Generating data race witnesses by an SMT-based analysis. In: Bobaru, M., Havelund, K., Holzmann, G.J., Joshi, R. (eds.) NFM 2011. LNCS, vol. 6617, pp. 313–327. Springer, Heidelberg (2011)
26. Savage, S., Burrows, M., Nelson, G., Sobalvarro, P., Anderson, T.: Eraser: A dynamic data race detector for multi-threaded programs. In: TOCS (1997)
27. Sinha, N., Wang, C.: On interference abstraction. In: POPL (2011)
28. Smaragdakis, Y., Evans, J., Sadowski, C., Yi, J., Flanagan, C.: Sound predictive race detection in polynomial time. In: POPL (2012)
29. Tarjan, R.: Depth-first search and linear graph algorithms. SIAM J. Comput. **1**(2), 146–160 (1972)
30. Wang, C., Limaye, R., Ganai, M., Gupta, A.: Trace-based symbolic analysis for atomicity violations. In: Esparza, J., Majumdar, R. (eds.) TACAS 2010. LNCS, vol. 6015, pp. 328–342. Springer, Heidelberg (2010)