On Stability in a Happens-Before Propagator for Concurrent Programs (Reproducibility Study)

Levente Bajczi(✉), Csanád Telbisz, Dániel Szekeres, and András Vörös

Department of Artifical Intelligence and Systems Engineering, Budapest University of Technology and Economics, Budapest, Hungary
{bajczi,vori}@mit.bme.hu

Abstract. Analyzing concurrent programs often involves reasoning about happens-before relations, handled by dedicated SMT theory solvers. Recently, preventative propagation rules have been introduced for consistency models to avoid unnecessary computations. This paper analyses the reproducibility of a recently published paper regarding a conflict-avoiding happens-before propagator. We show that the underlying axioms are insufficient for supporting sequential consistency. We find that the algorithm can leave out constraints on event ordering (even considering the original axioms), impacting the accuracy of verification. We show a simple counterexample to the stability claim in the paper. Two revisions of the algorithm are presented, and a proof on the correctness of these approaches respective of the original axioms is shown. The tool implementing the original algorithm is examined to ascertain how it circumvents wrong results. It is found that it deviates from the published algorithm. We show that an unmodified algorithm (via a patch in the implementing tool) causes incorrect results. We also show that our revised algorithm can be implemented efficiently in an independent verification tool.

Keywords: verification · formal methods · software verification

1 Introduction

Deagle [1,2] won the error-reachability-based ConcurrencySafety category of the software verification competition SV-COMP for two consecutive years in 2022 [3] and 2023 [4]. This has motivated us to research the underlying algorithms in detail, in hopes of implementing (and hopefully, improving) the state-of-the-art techniques that led Deagle to victory. One aspect of the algorithm found in Deagle is the consistency-preserving propagation of happens-before orders [5].

This research was partially funded by the EKÖP-24-{2,3} New National Excellence Program under project numbers EKÖP-24-2-BME-118, EKÖP-24-3-BME-213 and EKÖP-24-3-BME-159, and the Doctoral Excellence Fellowship Programme under project numbers 400434/2023, 400443/2023; funded by the NRDI Fund of Hungary. This work was partially supported by project no. 2022-1.2.4-EUREKA-2023-00013 under the 2022-1.2.4 EUREKA funding scheme from the the National Research, Development and Innovation Fund of Hungary.

© The Author(s) 2025

However, we found our implementation of the published algorithm faulty, yet DEAGLE produced no wrong results in 2022, and only one incorrect result in 2023. Therefore, we constructed a *reproducibility study* in hopes of uncovering the reason for this discrepancy. Our contributions in this paper concerning the reproducibility of the published results are the following:

- We analyze the published algorithm theoretically, and provide a counterexample to its *stability* claim (see Section 3), and dispute its applicability for *sequential consistency (SC)* (see Section 3.4)
- We propose two ways of fixing the algorithm, and prove the *stability* claim for both (see Section 4)
- We formulate actionable research questions to validate the applicability of the proposed approach(es) compared to the previous state of the art (see Section 5)
- We devise and implement experiments to answer the research questions (see Section 5.1)
- We evaluate the experimental data and answer the research questions, thus also providing an answer to the *reproducibility* premise of our paper (see Section 5.2, while also discussing the threats to the validity of our results (see Section 5.3)

We hope that our contributions and insights in this paper will save potential tool developers from experiencing the same contrariety between theory and practice, thus being able to build better-performing competing verification tools.

2 Happens-Before Relations in Concurrent Software

The formal verification of concurrent software is often reduced to reasoning about *happens-before* relations [6,7] (defined as a partial order on program instructions), utilized by many Satisfiability Modulo Theories (SMT) based verification tools both for strictly sequential, as well as weak memory software-hardware systems [8,9,5]. In most cases, either a dedicated theory solver, or an encoding to a pre-existing theory (supported by the underlying SMT solver backend) is applied.

The idea behind these techniques is the following. Instead of applying the semantics of asynchronous concurrency directly (i.e., any of the threads may advance from a state of the program, therefore we must analyze all possible orders), we treat global memory accesses as *events*, for which we must find a (partially defined) sequence they can be ordered in. Suitable orders are *consistent* with the execution semantics of the platform. Then, we pair *read* events to *write* events, which we call the *read-from* (rf) relation, meaning the *read* event returned the value written by the *write* event. Furthermore, we employ conventional analyses to explore the state space of the individual threads, treating *reads* as reading values from their *rf*-paired *write* accesses.

Because values returned by these *read* accesses might be used in guards of conditional statements, and in turn, the accesses to encode as events depend on the execution of these conditionals, one cannot completely separate the event graph and local analyses. However, in the context of this work, we presume that the local analyses already rely on SMT solvers, that can be utilized to handle the events and their relations. Some approaches encode this in a theory supported by the SMT solver [10], and others rely on custom theory solvers integrated with the SMT engine [8,9,5].

2.1 Consistency-Preserving Propagation

Satisfiability Modulo Theories (SMT) is the problem of deciding whether a value assignment to symbols exists that satisfies a first-order formula within formal *theories* [11]. SMT solvers utilize *theory solver* backends to handle the different supported theories (such as real numbers, integers, or arrays), and recently, even user-defined theory solvers (*user propagators*) have become possible to implement [12]. Generally, a dedicated theory solver must implement three procedures [11]:

- *Propagation*: Given a set of facts, derive consequences and add them to the set of known facts
- *Consistency checking*: Decide whether a certain set of facts are consistent with the background theory
- *Conflict clause generation*: If a set of facts is *inconsistent* with the background theory, determine which (minimal) subset is responsible for the inconsistency.

However, *Sun et al.* recently developed a framework of *preventative* propagation rules for sequential consistency that always result in consistent models, and thus, there is no need for either *consistency checking* or *conflict clause generation* [5]. This novel approach greatly boosts verification performance due to the decreased need for backtracking in the solver.

Unfortunately, part of the published algorithm contains an oversight, which results in missing crucial constraints on the event ordering (in case of certain sequences of decisions in the solver backend). We aim to revise this algorithm, and discuss its influence on the overall performance of the algorithm.

3 Instability in Propagation

In this paper, program verification based on event propagation assumes a loop-free program in concurrent static single assignment (CSSA) form [13], and an error property ρ_{error} (e.g., a designated "bad" program location, or some variable valuation representing an erroneous state) [5].

In this section, we start by summarizing the work of Sun et al. [5]. This work is not our contribution to claim, and for a more contextualized explanation, we direct the Reader to the original paper [5]. Here we introduce the core concepts and notations, which are necessary for understanding our contributions in later sections.

3.1 Value Encoding in SMT

First, we encode the value assignments of the program in the formula ρ_{va} by taking each *write* access with its accompanying *guard* (i.e., its enabling condition; the conjunction of all decisions the program must have taken in order for the event to execute). Given a guard grd_{w_i}, and assuming w_i sets variable v_i with CSSA index k to $expr_i$, then $grd_{w_i} \Rightarrow v_i^k = expr_i$. Consequently, ρ_{va} is the conjunction of all such implications.

Similarly, we must encode the *read* accesses in the program. Because the value of the *read* operations depend on their *rf*-paired *write* operation, we introduce a boolean variable rf_{ij} for each same-address events w_i and r_j. Then, if rf_{ij} is *true*, we know that $v_i = v_j$, grd_{w_i}, and grd_{r_j} hold. To encode all *reads*, ρ_{rf-val} [9] is the conjunction of all $rf_{ij} \Rightarrow grd_{w_i} \wedge grd_{r_j} \wedge v_i = v_j$. Furthermore, because all *reads* must be paired with a *write*, we know that if a grd_{r_i} holds, then $\exists j. rf_{ij}$. We encode this in $\rho_{rf-some}$ [9].

3.2 Ordering Constraint Encoding

Furthermore, we must encode the *ordering constraints* of the program. Due to causality, all *read* operations paired with a *write* operation must happen after the associated *write* operation (this is the *rf*-ordering set $\prec_{rf}$, a subset of the happens-before ordering set $\prec$). For sequential consistency, all *program order* (i.e., the actual instruction order in the program source) constraints are preserved, and are thus added to $\prec$ as ρ_{ppo}. For both of these sets, we use the same notation as a predicate, meaning $i \prec j := (i, j) \in \prec$. We also encode the relationship that $\forall (i, j). i \prec_{rf} j \iff rf_{ij}$ as ρ_{rf-ord}.

3.3 Decision Procedure

We must decide if $\rho_{va} \wedge \rho_{error} \wedge \rho_{rf-val} \wedge \rho_{rf-some} \wedge \rho_{rf-ord} \wedge \rho_{ppo}$ is satisfiable. If yes, we return the resulting model as the *counterexample* to the original problem. If not, we have proven the safety of the program.

This satisfiability check will need to check whether any candidate model would result in a cycle being formed in the transitive closure of the $\prec$ relation. Because all elements of $\prec$ denote a *happens-before* relation, this would mean that an element precedes itself, which is impossible, thus rendering the candidate inconsistent. To achieve this, we can implement a *propagation* module for the SMT solver, which will be called every time the decision procedure fixes a value for a given expression that influences the ordering sets. For us, these expressions will either be the rf_{ij} variables, or the guard expressions grd_k. We then perform *derivation* on the current elements of $\prec$ and $\prec_{rf}$ by applying the following axioms:

1. *transitivity* axiom: $(e_1 \prec e_2) \wedge (e_2 \prec e_3) \Rightarrow (e_1 \prec e_3)$
2. *from-read* axiom [9] for same-address events:
   $(w_1 \prec_{rf} r) \wedge (w_1 \prec w_2) \wedge grd_{w_2} \Rightarrow (r \prec w_2)$

$x_1 := 0$	$y_2 := 3$
$y_1 := 2$	$\mathrm{b} := y_3$
$\mathrm{a} := x_3$	$x_2 := 1$

Fig. 1: CSSA program

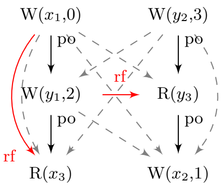

Fig. 2: Final $\prec$, $\prec_{rf}$

```
propagate(rf_x1_x3)              ≺_rf ,≺ += (x_1, x_3)
   transitivity(x_1, x_3)
   derive(x_1, x_2, x_3)

propagate(rf_y1_y3)              ≺_rf ,≺ += (y_1, y_3)
   transitivity(y_1, y_3)                ≺ += (y_1, x_2)
      transitivity(y_1, x_2)             ≺ += (x_1, x_2)
         transitivity(x_1, x_2)
                                         ≺ += (x_1, y_3)
      transitivity(x_1, y_3)
   derive(y_1, y_2, y_3)                 ≺ += (y_2, y_1)
      transitivity(y_2, y_1)             ≺ += (y_2, x_3)
         transitivity(y_2, x_3)
```

Fig. 3: A trace of Algorithm 1 over Figure 1

3. *write-serialization* axiom [5] for same-address events:
   $(w_1 \prec_{rf} r) \wedge (w_2 \prec r) \wedge grd_{w_2} \Rightarrow (w_2 \prec w_1)$

The algorithm for propagation is shown in Algorithm 1, taken from *Sun et al.*'s *Algorithm 2: Theory Propagation*[1] [5]. Then, either conventional consistency checking and conflict clause generation follows, or the *preventative reasoning* step introduced by *Sun et al.* [5]. The claim these solutions build upon is that after propagation, $\prec$ is *stable*, i.e., it contains all elements derivable by applying theory axioms. Using the example CSSA program in Figure 1 and the accompanying trace in Figure 3, we show a counterexample to this claim: x_3 and x_2 are not ordered in $\prec$ as shown in Figure 2, therefore any order may be taken. However, because $x_1 \prec_{rf} x_3$ and $x_1 \prec x_2$, Axiom 2 would order x_2 after x_3, yet this is not encoded in $\prec$. Therefore, we conclude that the state of $\prec$ is *not* always stable after propagation. This may (and will) lead to incorrect results.

3.4 Happens-Before Orders for Sequential Consistency

In this paper, we use the same axioms as *Sun et al.* [5] for the ordering constraints, and thus, for the happens-before order propagation. However, we must discuss the applicability of said axioms on *Sequential Consistency*, the most strict memory model targeted in the original paper (besides its relaxed versions *Total Store Ordering (TSO)* and *Partial Store Ordering (PSO)*).

Consider the happens-before graph (including *rf*, *po*, and generic $\prec$ relations) in Figure 4:

– We cannot apply the *transitivity* axiom any further, because all such relations have already been discovered (shown in gray).

[1] There are some typos in the original paper's Algorithm 2: in lines 31-33, e should be e'. These are minor mistakes and are fixed here.

Algorithm 1 Original Theory Propagation Algorithm [5], with typos fixed

```
Input: $l$: Positive literal
proc PROPAGATE($l$)
    if $l$ is $rf_{ij}$ then
        $w_i$, $r_j$ := write and read of $rf_{ij}$
        $\prec_{rf} \leftarrow \prec_{rf} \cup \{(w_i, r_j)\}$
        $\prec \leftarrow \prec \cup \{(w_i, r_j)\}$
        TRANSITIVITY($w_i$, $r_j$)
        foreach $w_k$ s.t. $grd_k$ do
            if $(w_i, w_k, r_j)$ same-addr. then
                DERIVE($w_i$, $w_k$, $r_j$)
            end if
        end foreach
    else if $l$ is $grd_k$ then
        $w_k$ := write of $grd_k$
        foreach $(w_i, r_i) \in \prec_{rf}$ do
            if $(w_i, w_k, r_j)$ same-addr. then
                DERIVE($w_i$, $w_k$, $r_j$)
            end if
        end foreach
    end if
end proc
```

```
Input: $w_i$, $w_k$, $r_j$: $w_i \prec_{rf} r_j$, $grd_k$
proc DERIVE($w_i$, $w_k$, $r_j$)
    if $w_k \prec r_j$ then
        $\prec \leftarrow \prec \cup \{(w_k, w_i)\}$
        TRANSITIVITY($w_k$, $w_i$)
    else if $w_i \prec w_k$ then
        $\prec \leftarrow \prec \cup \{(r_j, w_k)\}$
        TRANSITIVITY($r_j$, $w_k$)
    end if
end proc

Input: $e_1$, $e_2$: $e_1 \prec e_2$
proc TRANSITIVITY($e_1$, $e_2$)
    foreach $(e_2, e_3) \in \prec$ do
        if $(e_1, e_3) \notin \prec$ then
            $\prec \leftarrow \prec \cup \{(e_1, e_3)\}$
            TRANSITIVITY($e_1$, $e_3$)
        end if
    end foreach
    foreach $(e_0, e_1) \in \prec$ do
        if $(e_0, e_2) \notin \prec$ then
            $\prec \leftarrow \prec \cup \{(e_0, e_2)\}$
            TRANSITIVITY($e_0$, $e_2$)
        end if
    end foreach
end proc
```

- We cannot apply the *from-read* axiom, because that would require an *rf*-edge beginning with a *write* that has a successor *write* with the same address, and no such pair exists.
- We cannot apply the *write-serialization* axiom either, because that would require an *rf*-edge ending with a *read* that has a predecessor *write* with the same address, and no such pair exists.

Therefore, according to the axioms, we are finished with deriving happens-before relations, and as we found no cycles, the execution is deemed *allowed*. However, adding *any* order between the writes to z (shown in **bold**) makes a cycle (via the *reads* to z in the last two threads, when all axioms are applied again after adding either order between the writes), and therefore, this execution is not actually allowed over SC. Therefore, we conclude that the axioms are not

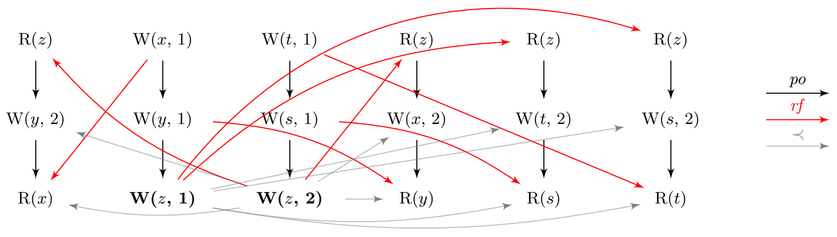

Fig. 4: Counterexample for the applicability of *Sun et al.*'s axioms [5] for Sequential Consistency (SC). Example taken from [14], Fig. 1c.

suitable for determining SC-consistency. Therefore, in this paper, we cannot claim we verify SC concurrency, but rather, a somewhat weaker memory model, called *Weak Sequential Consistency* [14].

Fixing this issue can be done post-propagation, by trying to serialize all writes in the program, and reporting the execution *inconsistent* with SC if this cannot be done. However, this scales exponentially with the number of same-variable writes, which we expect, as the problem (VSC-Read) is NP-complete [15][2]. Therefore, we exclude this fix from our solution proposals in Section 4, and focus on satisfying the axioms as they are written by *Sun et al.* [5], and show they are still not fulfilled.

4 Fixing Stability of the Propagation

To fix the stability problems of $\prec$ after the propagation step, we propose two solutions:

Retrospective approach We repeat Propagate in Algorithm 1 until its applications of all axioms no longer produce new relations.

Prospective approach We patch Algorithm 1 to disallow non-stable $\prec$ outputs by analyzing and fixing its procedures, thus achieving stability by construction.

4.1 Fixing Stability Retrospectively

In order to achieve *stability*, i.e., a state of $\prec$ where the theory axioms can no longer add new relations, the simplest method is to apply this definition directly by wrapping Propagate in the procedure in Algorithm 2. Because this will not return until $\prec$ stops changing, we can be sure that all axioms are fully applied and therefore, stability is achieved. However, as evident from Algorithm 2, this comes at the price of complexity: we need to iterate over $\prec$ and $\prec_{rf}$ repeatedly.

4.2 Fixing Stability Prospectively

To avoid the expensive option of fixing stability retrospectively, the alternative is to fix it *prospectively*, i.e., by applying the axioms only on recent additions to the orders. This was also the goal of *Sun et al.* [5], and hence Algorithm 1 needs only some minor modifications to realize this.

The main problem with Algorithm 1 is that Derive is not re-called even though the conditions $w_k \prec r_j$ and $w_i \prec w_k$ may change during later calls to any of the procedures when new elements are added to $\prec$. Therefore, as shown in Figure 3, when Derive is first called with $(w_i, w_k, r_j) = (x_1, x_2, x_3)$, it checks whether $w_k \prec r_j$ (which is false with $x_2 \prec x_3$), and $w_i \prec w_k$ (which is also false

[2] Note: the originally published algorithm is polynomial (every edge is added at most once to the event graph) [5], and therefore, cannot be the solution to this problem.

Algorithm 2 Retrospective algorithm

```
Input: $l$: Positive literal
1: proc PropagateWrapper($l$)
2:     Propagate($l$)
3:     while $\prec$ not fix do
4:         foreach $(e_1, e_2) \in \prec$ do
5:             Transitivity($e_1$, $e_2$) // Axiom 1
6:         end foreach
7:         foreach $(w_i, r_j) \in \prec_{rf}$ do
8:             foreach $w_k$ s.t. $grd_k$ do
9:                 if $(w_i, w_k, r_j)$ same-addr. then
10:                    Derive($w_i$, $w_k$, $r_j$) // Axiom 2, 3
11:                end if
12:            end foreach
13:        end foreach
14:    end while
15: end proc
```

with $x_1 \prec x_2$), then returns without adding anything to the order. Later, when propagating $rf_{y_1 y_3}$, during the call to Transitivity(y_1, x_2), the pair (x_1, x_2) is added to $\prec$, making the `else if` condition retroactively true for the Derive call above, but it is never checked again. Thus, $x_3 \prec x_2$ is missed, $\prec$ is unstable, and we mistakenly allow some executions that would never be observable on the execution platform.

To fix this issue, one solution is to re-call Derive every time a new relation is added to $\prec$ that could influence the conditions therein. This means that all same-address (w_k, r_j) and (w_i, w_k) relations are subject to this rule, for all previously checked triples (w_i, w_k, r_j). Because we know that Derive is only ever called with same-address events for which $w_i \prec_{rf} r_j$ also holds, we already have access to the set of already checked triples. Therefore we introduce a helper procedure, AddToOrder, and change Algorithm 1 to always call this procedure instead, when any other procedure adds a new element to $\prec$. The updated algorithm can be seen in Algorithm 3. Notice that checking the nonexistence of the new pair in $\prec$ is also included in AddToOrder, leading to it no longer being necessary in other procedures.

Theorem 1. *After executing Algorithm 3 with any l positive literal, then given $\prec$ is stable, $\prec$ remains stable.*

To prove stability, we must show that all three axioms are fully applied to $\prec$ when Propagate returns.

Lemma 1. *Axiom 1 is fully applied (may no longer be used to derive new relations) when* Propagate *returns.*

Proof. The full application of Axiom 1 is always given, as after every addition to $\prec$ (line 3), the procedure Transitivity is called, which adds all events that

Algorithm 3 Revised Theory Propagation Algorithm

```
Input: $l$: Positive literal
 1: proc PROPAGATE($l$)
 2:     if $l$ is $rf_{ij}$ then
 3:         $w_i$, $r_j$ := write and read of $rf_{ij}$
 4:         $\prec_{rf} \leftarrow \prec_{rf} \cup \{(w_i, r_j)\}$
 5:         ADDTOORDER($w_i, r_j$)
 6:         foreach $w_k$ $s.t.$ $grd_k$ do
 7:             if $(w_i, w_k, r_j)$ same-addr. then
 8:                 DERIVE($w_i$, $w_k$, $r_j$)
 9:             end if
10:         end foreach
11:     else if $l$ is $grd_k$ then
12:         $w_k$ := write of $grd_k$
13:         foreach $(w_i, r_i) \in \prec_{rf}$ do
14:             if $(w_i, w_k, r_j)$ same-addr. then
15:                 DERIVE($w_i$, $w_k$, $r_j$)
16:             end if
17:         end foreach
18:     end if
19: end proc
```

```
Input: $e_1, e_2$: Events
 1: proc ADDTOORDER($e_1, e_2$)
 2:     if $(e_1, e_2) \notin \prec$ then
 3:         $\prec \leftarrow \prec \cup \{(e_1, e_2)\}$
 4:         TRANSITIVITY($e_1$, $e_2$)
 5:         if $(e_1, e_2)$ same-addr. then
 6:             if $e_1$ is write, $e_2$ is read then
 7:                 foreach $(w, e_2) \in \prec_{rf}$ do
 8:                     DERIVE($w, e_1, e_2$)
 9:                 end foreach
10:             else if $e_1$ is write, $e_2$ is write then
11:                 foreach $(e_1, r) \in \prec_{rf}$ do
12:                     DERIVE($e_1, e_2, r$)
13:                 end foreach
14:             end if
15:         end if
16:     end if
17: end proc
```

```
Input: $w_i$, $w_k$, $r_j$: $w_i \prec_{rf} r_j$, $grd_k$
 1: proc DERIVE($w_i$, $w_k$, $r_j$)
 2:     if $w_k \prec r_j$ then
 3:         ADDTOORDER($w_k, w_i$)
 4:     end if
 5:     if $w_i \prec w_k$ then
 6:         ADDTOORDER($r_j, w_k$)
 7:     end if
 8: end proc
```

```
Input: $e_1$, $e_2$: $e_1 \prec e_2$
 1: proc TRANSITIVITY($e_1$, $e_2$)
 2:     foreach $(e_2, e_3) \in \prec$ do
 3:         ADDTOORDER($e_1, e_3$)
 4:     end foreach
 5:     foreach $(e_0, e_1) \in \prec$ do
 6:         ADDTOORDER($e_0, e_2$)
 7:     end foreach
 8: end proc
```

are $\prec$-before the first event as being $\prec$-before the second element as well; and events $\prec$-after the second event as being $\prec$-after the first element as well. This is the definition of Axiom 1.

Lemma 2. *Axiom 2 and Axiom 3 are fully applied (may no longer be used to derive new relations) when* PROPAGATE *returns.*

Proof. Axiom 2 as per its definition can derive new elements of $\prec$ when either grd_k becomes *true*, or a new same-address *write-write* pair is added to $\prec$, or a new pair is added to $\prec_{rf}$. In addition, Axiom 3 can also derive new elements when a new same-address *write-read* pair is added to $\prec$. Therefore, when:

1. grd_k becomes *true*, we must examine all $w \prec_{rf} r$ elements for which the premise of Axiom 2 may hold with w_k, i.e., $w \prec w_k$. Therefore, DERIVE(w, w_k, r) is called with all $(w, r) \in \prec_{rf}$ (line 15 in PROPAGATE), and checks whether $w \prec w_k$ holds (line 5), after which it adds the new derived element (r, w_k) to $\prec$;
2. a new same-address *write-write* pair (w_1, w_2) is added to $\prec$, we must examine all $w_1 \prec_{rf} r$ pairs. We know grd_{w_2} holds because it could not have been added to $\prec$ otherwise, so we call DERIVE(w_1, w_2, r) (in line 12 of ADDTOORDER), which, because $w_1 \prec w_2$ (in line 5), adds (r, w_2) to $\prec$;
3. a new same-address *write-read* pair (w, r) is added to $\prec$, we must examine all such $w_2 \prec_{rf} r_2$ pairs where r is r_2. We know grd_{w_2} holds because it could

not have been added to $\prec_{rf}$ otherwise, so we call Derive(w_2, w, r) (in line 8 of AddToOrder), which, because $w \prec r$ (in line 2), adds (w, w_2) to $\prec$;
4. a new pair (w, r) is added to $\prec_{rf}$, we must examine all same-address *write-write* (w_1, w_2) (for Axiom 2, where w_1 is w) and *write-read* (w_2, r_2) (for Axiom 3, where r_2 is r) pairs in $\prec$. In both cases, grd_{w_2} must hold. Therefore, we must look at all guard-enabled same-address *writes*, and call Derive(w, w_2, r) (in line 8 of Propagate), which will add both the consequence of Axiom 2 (in line 6), and the consequence of Axiom 3 (in line 3), given their premises are fulfilled (lines 5 and 2, respectively)[3].

Because for every possible change in the terms of the premises of Axiom 2 and Axiom 3 the premises are checked and the consequences are applied, both axioms are fully applied.

5 Empirical Evaluation

In order to determine the performance impact and efficacy of the two proposed solutions, we formulated the following experimental research questions:

ERQ1 How does the performance change among the original, the retrospective, and the prospective algorithm in an isolated, *clean* implementation?
ERQ2 How does the practical implementation in the tool corresponding to the original *Sun et al.* publication [5] (Deagle) circumvent *false* results, given the theoretical issue with Algorithm 1?
ERQ3 How does the performance and efficacy of the revised algorithms (both the prospective and the retrospective) transfer to another verification tool (Theta [16])?
ERQ4 How does the performance change among the retrospective and the prospective algorithm when integrated in a model checking tool?

Additionally, we formulate the premise of our paper as a research question:

RQ Is the stability claim by *Sun et al.* [5] supported theoretically, and reproducible in practice?

5.1 Experimental Setup

We used the ConcurrencySafety-Main category of SV-COMP [4] to measure verification performance throughout the experiments. We relied on Benchexec [17] to provide accurate and reproducible performance measurements.

[3] In the original algorithm presented in Algorithm 1, the check for Axiom 2 and Axiom 3 were mutually exclusive in Derive. We believe this is faulty, and have therefore changed it here.

Isolated Implementation We transformed a simple (i.e., pointerless and arrayless) subset of the SV-COMP ConcurrencySafety-Main category's tasks into the CSSA form suitable for the algorithms above, with an unrolling bound of 2. Then, we used the original algorithm (in Algorithm 1) to generate a consistent rf-assignment for each task. We then replayed the order of rf-assignments in the two algorithms proposed in Section 4. We measured time, and the size of $\prec$ after propagation. There were 22 tasks where there was a discrepancy in the size of $\prec$ among the original and the two revised algorithms (on average, 6.9%), in which cases more traces are thought to be possible than in reality, possibly leading to false unsafe verdicts (see Figure 1). The two revised algorithms always produced the same size $\prec$, as expected, empirically supporting the soundness proof in Section 4.2. In these 22 instances, the performance of the three algorithms is visible in Figure 5. We can see that the retrospective solution is much slower than the original (on average, by 240%), and that the prospective approach is consistently a bit slower (on average, by 16%).

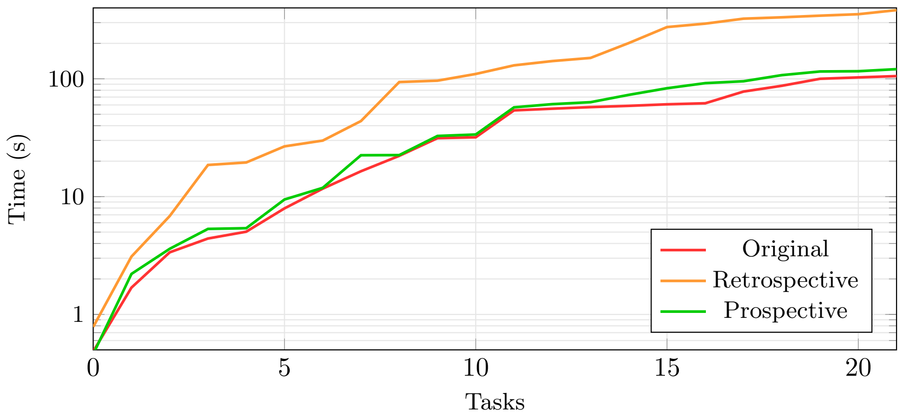

Fig. 5: Quantile plot of execution times for the isolated implementation

Implementation in Deagle We used the publicly available source code of Deagle[4] for this part of the experimental evaluation. We also ran our experiments with the SV-COMP'24 binary release [2]. We used 900 seconds of timeout in the experiments.

We used the entire ConcurrencySafety-Main category of SV-COMP with 713 tasks. The results can be seen in Figure 6, showing a verdict-based comparison.

We first ran our experiments with the binary release [2] and an unmodified version of the source code. We found that the binary release solved marginally

[4] https://github.com/thufv/Deagle/commit/19e267151cca620cb1d24bc109a451b8a0e617f8

fewer tasks, but produced *no incorrect verdicts*. However, the source release solved marginally more tasks, but produced 14 incorrect verdicts. We compared the logs of these two configurations, and found the lines "Use SV-COMP unwinding strategy: X" differ between the two configurations, varying between 2 and 100. This leads us to believe that there is a discrepancy in the way the unwinding strategy is selected between the two versions, which leads to the slight difference in solved tasks and can cause false results, because Deagle accepts a bounded proof of safety as an overall safety proof. There is one notable exception, pthread-race-challenges/thread-local-value.yml, which is correctly determined to be *safe* by the binary version with 5 unwindings, but incorrectly classified as *unsafe* by the source version with 2 unwindings. This means that there may be further differences between the versions.

We examined the source code of Deagle to find out how it is possible that with a faulty implementation, there can still exist a version (the binary release) which circumvents all incorrect results. We found that in the ClosureSolver.cc file there is a one_more_time flag in the propagate() function[5] that checks for newly added edges, and recursively calls propagate again, until a fixpoint (no new edges) is reached. This closely resembles what we call the *retrospective* approach.

Because this is not aligned with the algorithm of the original publication, we removed this flag. Besides this flag, we found no other meaningful difference between the published algorithm and the source. We ran the experiments with this version as well, and its results are included in Figure 6. A lot of tasks run afoul of an assertion in this configuration, causing an overall dip in solved tasks, but this is not (directly) the problem of the originally published algorithm, just a side effect. However, even with this smaller sample size, the nominal number of incorrect verdicts still grew: there were 20 tasks that this version solved incorrectly, out of which 14 could still be solved by the source code, unpatched version.

	binary release (retrospective)	source code (retrospective)	patched (original)
correct	**618**	**628**	**236**
true	320	332	165
false	298	296	71
incorrect	**0**	**14**	**20**
true	0	10	4
false	0	4	16

Fig. 6: Verification efficacy of Deagle's binary release, source code release, and patched version

[5] https://github.com/thufv/Deagle/blob/19e267151cca620cb1d24bc109a451b8a0e617f8/minisat-2.2.1/minisat/core/ClosureSolver.cc#L452

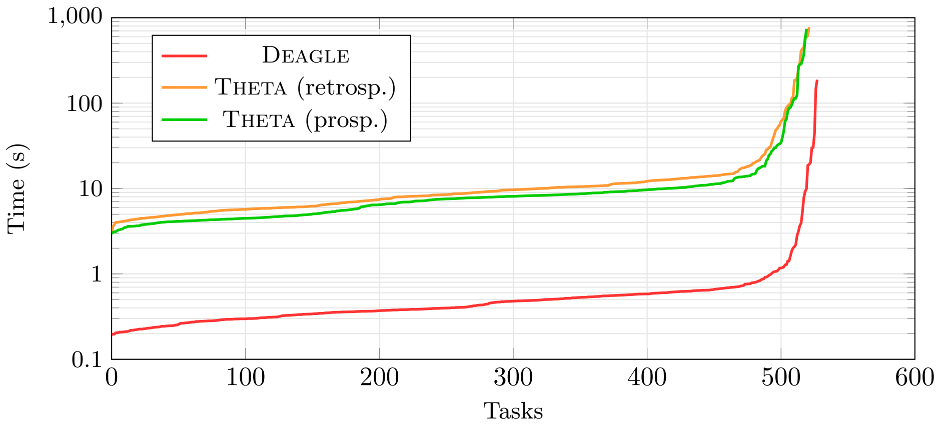

Fig. 7: Quantile plot of execution times for THETA vs. DEAGLE

Implementation in Theta We implemented a version of the prospective, as well as the retrospective algorithm in THETA [16] (as an independent model checker tool). We compare its experimental results with the best performing, source-based, unpatched version of DEAGLE. The results can be seen as a quantile plot in Figure 7. THETA produced 2 wrong results due to insufficient loop unrollings, and DEAGLE produced 14 wrong results, presumably due to similar issues. These verdicts are not included in the quantile plot. Furthermore, THETA does not support certain elements of the C language in its frontend, and therefore we excluded these tasks to preserve a fair comparison.

We can see (in Figure 7) that the two tools produce almost the same number of correct verdicts (528 for DEAGLE and 522/520 for the retrospective/prospective version of THETA), with similar performance characteristics. The offset in performance (in our opinion) is down to THETA being a non-native, JVM-based application, while Deagle is a compiled, native program. We can further see that while the prospective algorithm consistently outperformed the retrospective one, it did so only marginally, and even ended up solving two tasks fewer.

5.2 Evaluation of Experiments

Based on our results reported in Section 5.1, we can answer the three experimental research questions.

ERQ1 As shown in Section 5.1, the original (faulty) algorithm is the fastest, the prospective algorithm is slightly slower (by around 16%), and the retrospective algorithm is much slower (by around 240%).

ERQ2 As shown in Section 5.1, the published algorithm [5] and the implementation in the published source code differ in containing a flag to re-run PROPAGATE() when necessary (akin to our proposed *retrospective* solution). Without this patch, the original algorithm produces (more) wrong results.

ERQ3 The performance and efficacy of the algorithm transfers suitably to independent tools, as shown with our implementation in the THETA model checking framework.

ERQ4 As shown in Figure 7, the prospective algorithm is slightly faster than the retrospective algorithm. The difference is far less pronounced than with the clean, isolated implementation (Section 5.1).

Additionally, we can answer **RQ** as well. The claim that after propagation, the state of the execution graph is *stable*, is not supported theoretically, as shown with our counterexample in Figure 3. In practice, the stability *is* reproducible, but only with a modified algorithm, containing a supporting flag, as uncovered in Section 5.1. Without this patch, the stability after running the algorithm is not guaranteed, and can cause real-life problems, as the newly incorrect verdicts in Section 5.1 showcase.

5.3 Threats to Validity

As one of the main contributions of this paper is the experiment design and its analysis, the factors that threaten the validity of this experiment are presented in this section.

Internal Validity. Consistency and accuracy of the experiments were ensured by using the BenchExec framework [17]. Memory consumption statistics may deviate between executions due to the managed nature of some languages used in developing the tested tools, therefore, such metrics are not used. CPU time and, therefore, solved tasks may be influenced by external factors such as other processes or environmental temperature fluctuations, therefore, minute differences are disregarded.

External Validity. The results of the experiments are at risk of not being generalizable due to the relatively low number of benchmarks used throughout the experiments. Furthermore, we as authors do not have access to the most up-to-date source code of DEAGLE which was used to compile the competition binary. Additionally, we are not familiar with the whole DEAGLE code base, which may mean we missed crucial implementation details in the tool. Also, we were not able to meaningfully circumvent the assertion violation using our patched version when trying to achieve an implementation close to the published one, and therefore, the majority of test cases were excluded in that analysis.

Construct Validity. To justify the type of metrics used in the evaluation of the experiments, we considered the main use cases these tools would face should they be used in the approach described in this paper. Academic competitions such as SV-COMP [4] reflect the performance of tools after careful tuning of them while constantly re-testing on the same benchmark set. Therefore, our results may not be easily reproduced on a different benchmark set.

6 Conclusion and Future Work

In this paper, we have shown that the *Basic algorithm* published by *Sun et al.* [5] contains a problem where some $\prec$ elements are not discovered given a set of axioms for weak sequential consistency. Their algorithm is the basis of the software verification tool Deagle [5] that was the winner of the concurrency category of SV-COMP 2022 [3] and 2023 [4] which highlights the importance of the approach and the need for its improvement. We have proposed two solutions to this problem, a retrospective and a prospective algorithm, and have proven the post-propagation stability property of both. Using empirical data, we can conclude that in some circumstances the issue materializes in real-life problems as well, but there is a performance impact of using the revised algorithms. We found the prospective algorithm does not impact performance as much as the retrospective, but we could only show a significant difference in an isolated environment. We further found that the original implementing tool, Deagle, already contains a way to circumvent the problem, by utilizing a solution resembling our retrospective approach. We show that without this modification of the algorithm, Deagle does produce wrong results on verification tasks. We also showed that our proposal transfers to other model checkers as well, by showing that an implementation in the Theta [16] model checking framework achieved similar results to that of Deagle. Finally we have analyzed the empirical results and answered the premise of our reproducibility study: *the claims of the paper are not substantiated either theoretically, or in practice.*

References

1. F. He, Z. Sun, and H. Fan, "Deagle: An SMT-based Verifier for Multi-threaded Programs (Competition Contribution)," in *Tools and Algorithms for the Construction and Analysis of Systems*, D. Fisman and G. Rosu, Eds. Cham: Springer International Publishing, 2022, pp. 424–428.
2. Z. Sun, "Deagle for SV-COMP 2024," Nov. 2023. [Online]. Available: https://doi.org/10.5281/zenodo.10207348
3. D. Beyer, "Progress on Software Verification: SV-COMP 2022," ser. Lecture Notes in Computer Science, D. Fisman and G. Rosu, Eds., vol. 13244. Springer, 2022, pp. 375–402. [Online]. Available: https://doi.org/10.1007/978-3-030-99527-0_20
4. D. Beyer, "Competition on Software Verification and Witness Validation: SV-COMP 2023," ser. Lecture Notes in Computer Science, S. Sankaranarayanan and N. Sharygina, Eds., vol. 13994. Springer, 2023, pp. 495–522. [Online]. Available: https://doi.org/10.1007/978-3-031-30820-8_29

5. Z. Sun, H. Fan, and F. He, "Consistency-preserving propagation for SMT solving of concurrent program verification," *Proc. ACM Program. Lang.*, vol. 6, no. OOPSLA2, oct 2022. [Online]. Available: https://doi.org/10.1145/3563321
6. J. Alglave, L. Maranget, S. Sarkar, and P. Sewell, "Fences in weak memory models (extended version)," *Formal Methods Syst. Des.*, vol. 40, no. 2, pp. 170–205, 2012. [Online]. Available: https://doi.org/10.1007/s10703-011-0135-z
7. J. Alglave, D. Kroening, and M. Tautschnig, "Partial Orders for Efficient Bounded Model Checking of Concurrent Software," in *Computer Aided Verification*, N. Sharygina and H. Veith, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2013, pp. 141–157.
8. T. Haas, R. Meyer, and H. Ponce de León, "CAAT: consistency as a theory," *Proc. ACM Program. Lang.*, vol. 6, no. OOPSLA2, oct 2022. [Online]. Available: https://doi.org/10.1145/3563292
9. F. He, Z. Sun, and H. Fan, "Satisfiability modulo ordering consistency theory for multi-threaded program verification," in *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation*, ser. PLDI 2021. New York, NY, USA: Association for Computing Machinery, 2021, p. 1264–1279. [Online]. Available: https://doi.org/10.1145/3453483.3454108
10. H. Ponce-de León, F. Furbach, K. Heljanko, and R. Meyer, "Dartagnan: Bounded Model Checking for Weak Memory Models (Competition Contribution)," in *Tools and Algorithms for the Construction and Analysis of Systems*, A. Biere and D. Parker, Eds. Cham: Springer International Publishing, 2020, pp. 378–382.
11. C. Barrett and C. Tinelli, *Satisfiability Modulo Theories*. Cham: Springer International Publishing, 2018, pp. 305–343. [Online]. Available: https://doi.org/10.1007/978-3-319-10575-8_11
12. N. Bjørner, C. Eisenhofer, and L. Kovács, "Satisfiability Modulo Custom Theories in Z3," in *Verification, Model Checking, and Abstract Interpretation*, C. Dragoi, M. Emmi, and J. Wang, Eds. Cham: Springer Nature Switzerland, 2023, pp. 91–105.
13. C. Wang, S. Kundu, M. Ganai, and A. Gupta, "Symbolic Predictive Analysis for Concurrent Programs," in *FM 2009: Formal Methods*, A. Cavalcanti and D. R. Dams, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2009, pp. 256–272.
14. R. Zennou, M. F. Atig, R. Biswas, A. Bouajjani, C. Enea, and M. Erradi, "Boosting Sequential Consistency Checking Using Saturation," in *Automated Technology for Verification and Analysis - 18th International Symposium, ATVA 2020, Hanoi, Vietnam, October 19-23, 2020, Proceedings*, ser. Lecture Notes in Computer Science, D. V. Hung and O. Sokolsky, Eds., vol. 12302. Springer, 2020, pp. 360–376. [Online]. Available: https://doi.org/10.1007/978-3-030-59152-6_20
15. P. B. Gibbons and E. Korach, "Testing shared memories," *SIAM Journal on Computing*, vol. 26, no. 4, pp. 1208–1244, 1997.
16. T. Tóth, A. Hajdu, A. Vörös *et al.*, "Theta: a Framework for Abstraction Refinement-Based Model Checking," in *17th Conference on Formal Methods in Computer-Aided Design*, D. Stewart and G. Weissenbacher, Eds., 2017, pp. 176–179.
17. D. Beyer, S. Löwe, and P. Wendler, "Reliable benchmarking: requirements and solutions," *Int. J. Softw. Tools Technol. Transf.*, vol. 21, no. 1, pp. 1–29, 2019.

Open Access. This chapter is licensed under the terms of the Creative Commons Attribution 4.0 International License (http://creativecommons.org/licenses/by/4.0/), which permits use, sharing, adaptation, distribution, and reproduction in any medium or format, as long as you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons license and indicate if changes were made.

The images or other third party material in this chapter are included in the chapter's Creative Commons license, unless indicated otherwise in a credit line to the material. If material is not included in the chapter's Creative Commons license and your intended use is not permitted by statutory regulation or exceeds the permitted use, you will need to obtain permission directly from the copyright holder.