On-the-fly Cone-of-Influence Reduction for Model Checking Concurrent Software

Csanád Telbisz, Levente Bajczi, Dániel Szekeres, and András Vörös

Department of Measurement and Information Systems
Budapest University of Technology and Economics, Budapest, Hungary
`csanadtelbisz@edu.bme.hu`
https://ftsrg.mit.bme.hu

Abstract. Calculating successor states in SMT-based software model checking is a costly task that often requires solving an SMT problem. However, in many cases, the evaluation of a program statement has no effect with respect to the verified property. Successor state calculation can be simplified in such cases. Several algorithms exist such as the cone-of-influence reduction that statically analyze the model and eliminate irrelevant variables from the model. In concurrent software, however, it is common for the result of a statement to be used in one interleaving of threads while unused in another. Algorithms that statically analyze the model cannot simplify such statements. Our on-the-fly approach detects whether a statement can be simplified during the state space exploration based on the current state of each process. Evaluation results show that our algorithm can simplify around 20% of all statements on average over a large set of benchmark programs while reducing the time of successor state calculation by more than 30% on average.

Keywords: cone-of-influence · concurrency · abstraction · data-flow.

Funding. This research was partially funded by the 2024-2.1.1-EKÖP-2024-00003 University Research Scholarship Programme under project numbers EKÖP-24-2-BME-118 and EKÖP-24-3-BME-{159,213}, and the Doctoral Excellence Fellowship Programme under project numbers 400434/2023 and 400443/2023; with the support provided by the Ministry of Culture and Innovation of Hungary from the NRDI Fund.

1 Introduction

Model checking has been a field of active research in recent decades as it is one of the most powerful software verification techniques. Model checking algorithms face the state space explosion problem: the state space of software systems grows exponentially with the number of variables [24]. Concurrency further increases the complexity due to the great number of possible thread interleavings. Various techniques have been developed to tackle this vast complexity. Partial order reduction avoids exploring parts of the state space when it is guaranteed that an equivalent thread interleaving is explored for each avoided trace [44]. Abstraction reduces the size of the state space by ignoring some details of the original problem [22,23]. Counterexample-guided abstraction refinement (CEGAR) iteratively refines the abstraction until the desired property can be verified [22]. Other abstraction-based techniques like the cone-of-influence (COI) reduction or program slicing eliminate model elements irrelevant to the verified property [10,33].

Existing cone-of-influence and slicing techniques choose eliminable variables or statements using static data-flow analysis based on the control-flow of the program [10,33]. In concurrent programs, this kind of elimination is often ineffective due to the communication between threads and the many possible thread interleavings. To address this, we propose an algorithm that can eliminate statements based on the current local states of concurrent threads. Whereas the COI reduction simplifies the model by eliminating completely redundant variables (redundant in all thread contexts) regarding the verified property [10], our approach identifies and simplifies statements on-the-fly that are redundant in the current state of concurrent threads with respect to the verified property. Thus, our method is more fine-grained: it can eliminate statements in certain contexts even if the statement cannot always be ignored. This is particularly useful when a statement is relevant in one interleaving of threads while it is redundant in another: we eliminate it in cases when it is redundant. As our algorithm takes its main advantage from the local states and interleaving of threads, we focus our presentation on concurrent programs. While our method could also be used for sequential programs, it loses its advantage over other techniques in that case.

As an example, take the program with two threads from Figure 1a. Let us take a state s from the state space of the program where process p_2 has executed the statement `y := x` previously (i.e., this statement can be found on the path from the initial state to s). Observe that the value of x cannot be read by any statement of any thread reachable from s in the state space. Thus, it is unnecessary to evaluate `x := 1` or `x := 0` after s. Our algorithm detects such situations and eliminates such statements. Note that a traditional COI algorithm could not eliminate the statement `x := 1` as its result may be used.

Our statement simplification method is motivated by the considerable runtime of calculating successor states in SMT-based state space exploration [12,32]. We strive to dynamically identify as many redundant statements as possible in the current exploration context. For this, we build a data-flow graph and update it based on the current thread interleaving during the state space exploration to reflect the individual states of each process. Before evaluating a statement (i.e., calculating the successor of the current state with respect to this statement), we check using the data-flow graph whether any other statement can use the result of the statement. We target reachability properties; thus, we are interested in whether the result of the statement is used transitively by a conditional statement, as only conditionals can directly influence whether some marked error locations in the model are reachable. This can be decided by a traversal of the data-flow graph. Redundant statements are eliminated, sparing the time of successor state calculation in such cases. We formulate our algorithm for abstract

Process p_1 | Process p_2

```
x := 1          | y := x
y := 1          | x := 0
assert(y=1)     |
```

(a) Simple example

initially: `x:=y:=z:=0`

Process p_0 | Processes $p_1 - p_N$; Processes $p_{N+1} - p_{2N}$

```
repeat N times:    |    y := y+1
  z := z+2*y       |
if z mod 2 = 0:    |
  x := 0           |    y := y*y
else:              |
  x := 1           |
```

finally: `assert(x*y=0)`

(b) Example with possible exponential gain

Fig. 1: Motivational examples for demonstration

state space exploration and exploit information about the current abstraction to reduce the number of edges in the data-flow graph.

To further motivate our approach, it is possible to achieve exponential gains in the number of evaluated statements by using our novel algorithm. Consider the example from Figure 1b with $2N+1$ processes. The safety of the program can be proven with abstraction by only tracking the predicates $z \bmod 2 = 0$ and $x = 0$ about our variables: $z \bmod 2 = 0$ is an invariant of the loop of p_0, so x gets 0 that satisfies the assertion. Processes $p_1 - p_{2N}$ have $2N!$ interleavings not considering the statements of the loop of p_0; together there are even more interleavings. This is indeed a difficult task for verifiers: we tried state-of-the-art tools (such as Ultimate [38] and Dartagnan [41]), and they cannot solve the problem for $N > 4$ within a reasonable time. However, our algorithm notices that we do not track any information about y, so the results of statements writing y are not used in this abstraction: so, our approach eliminates all of these statements ($2N! * 2N$ statements exactly). Our approach also enables existing partial order reduction algorithms [1,3,30] to reduce the number of explored interleavings exponentially that otherwise would have to explore all interleavings. Our algorithm achieves this by eliminating the source of dependency between statements.

Contributions. We take the base idea of the cone-of-influence reduction one step further by deciding on-the-fly during state space exploration whether the result of a statement can be used later. We present a novel algorithm for identifying redundant statements using an abstract dynamically updated data-flow graph. Furthermore, we discuss the necessary additions in an iterative abstraction-refinement verification scheme (namely, CEGAR). We have implemented and evaluated our algorithm in the abstraction-based model checking tool Theta [48].

2 Related Work

Several works aim to simplify the model by eliminating redundant model elements based on data-flow analysis [10, 28, 33, 36, 42, 43]. These techniques only statically analyze and simplify the input model which is limited compared to our on-the-fly data-flow analysis. These static approaches have the advantage that they have to be executed only once before the state space exploration while our algorithm is performed at each successor state calculation. On the other hand, our experiments in Section 5 show that our approach does not have a significant overhead, so it is worth running our algorithm several times (i.e., once for each state transition during the state space exploration) to eliminate further statements that static techniques cannot eliminate.

There are dynamic program slicing techniques as well [33, 39]. However, *dynamic*, in those contexts, means that these techniques use actual input values or already discovered error traces for slicing [5]. These techniques do not take advantage of the local states and interleaving of threads (most of them formulated for sequential programs [39]) which is the basis of our approach.

Many algorithms have been developed for model checking concurrent programs that reduce the number of explored thread interleavings such as partial order reduction or maximum causality reduction [1, 4, 37]. Some of these techniques leverage abstraction-related information to achieve further reduction [8, 29, 30], but they do not use a complex data-flow analysis to further reduce the number of dependent program statements. Some other works perform dynamic data-flow analysis in various ways to improve the reduction potential of these techniques [4,6,20,37,45], though they only use data-flow analysis to reduce the number of explored interleavings and not to simplify statements. These techniques take explored traces and discover redundant statements within these traces: they use this information to explore even less interleavings (e.g., by ignoring these statements when calculating a dependency relation [6]). The works [4, 37] discover causality connections and build causality constraints between statements (events) of a trace and simplify these formulae by eliminating irrelevant statements which is a similar concept to our approach. However, they only use this idea to simplify these constraints, but they still completely explore traces first. So our approach could achieve further reduction in these cases as well. In other words, these works aim to reduce the size of the explored state space whereas our purpose is to accelerate the exploration of a (reduced) state space by skipping the evaluation of certain program statements. Our algorithm is orthogonal to these techniques and could be applied on top of them to further improve the performance by eliminating further model elements.

The combination of abstraction-based and slicing techniques have already been investigated [18, 27]. Those approaches enhance the refinement step of the counterexample-guided abstraction refinement approach by slicing infeasible error traces found during state space exploration. On the other hand, our technique works in the state space exploration phase of CEGAR. Therefore, the two techniques are orthogonal and have a different purpose. We include some remarks about the consistent use of our technique with refinement methods in Section 4.3.

Dynamic pruning and a so-called *dynamic cone-of-influence* algorithm is also introduced in [21]. However, it is just a coincidental name collision: they use it in the context of fault tree analysis to lazily construct fault trees and compute minimal cut sets. It has nothing to do with concurrent programs.

3 Preliminaries

This section introduces the basic concepts and notions regarding abstract state space exploration necessary for the presentation of the proposed technique.

3.1 Computation Model

In this paper, we assume a computation model of concurrent programs where processes (threads) communicate via shared variables. We assume a sequential consistency memory model. Though it would be easy to incorporate extra features into the model (such as heap memory, dynamic thread creation or termination, and synchronization primitives), we strive to keep our presentation simple and skip these details. Our implementation for the evaluation naturally supports these features. We represent concurrent programs by control-flow automata (CFA) [15]: each process has its own (conventional) CFA representation.

Definition 1. *A multi-threaded CFA is a tuple (V, P), where*

- *V is a set of (global) variables,*
- *P is a set of processes. A process is a tuple $p = (L, l_0, A, E)$, where:*
  - *L is a set of control locations with $l_0 \in L$ as the initial location,*
  - *A is a set of statements,*
  - *$E \subseteq L \times A \times L$ is a set of transitions. A transition is a directed edge with a source control location, a target control location, and one statement.*

Each variable $v \in V$ has a domain D_v (the possible values for v), and possibly an initial value from its domain. A statement can be a deterministic assignment ($v = expr$), a non-deterministic assignment ($havoc\ v$) where the new value of v can be anything from its domain, or a guard condition ($[cond]$). For the verification of reachability properties, some CFA locations are marked as error locations: the program is safe if no error location can be reached by any of its processes. We define transition systems (state spaces) as follows:

Definition 2. *A transition system is a tuple (S, A, T, I), where S is a set of states, A is a set of actions, $T \subseteq S \times A \times S$ is a set of transitions, and $I \subseteq S$ is a non-empty set of initial states.*

In this paper, actions correspond to statements as introduced above. An action α is an *outgoing* action from a state s if there is a transition $(s, \alpha, s') \in T$ for some $s' \in S$. We use the following notations:

- $\alpha(s) = \{s' \in S : \quad \exists(s, \alpha, s') \in T\}$,
- $outgoing(s)$ denotes the set of outgoing actions from s,
- $vars(\alpha)$ denotes the set of variables referenced by α,
- $written(\alpha)$ and $read(\alpha)$ is the set of variables written/read by α, respectively.

The state space of a program is a transition system where a state stores the CFA locations of all processes and the values of all variables. An action of a transition corresponds to a statement of a single process (processes step asynchronously). We use the Greek alphabet for actions, and we write p_α for the process of action α. Note that $written(\alpha)$ has a single item for deterministic and non-deterministic assignments, and it is an empty set for a guard condition. We denote the control location of process p in state s by $s(p)$, and the value of variable v in state s by $s(v)$. We define an expression function for a state s based on the values of variables in s: $expr(s) := \bigwedge_{v \in V}(v = s(v))$. By $w = t_1 ... t_k$, we denote a transition sequence (or trace), and we use the following for the concatenation of transition sequences or transitions: $w.v$. We also refer to action sequences as traces. A state is an error state if any of the processes is in an error location in the state. A state s' is said to be reachable from a state s if there is a trace starting from s and ending in s'. If we have a trace from a state that leads to an error state, we call this trace an error trace.

3.2 Abstraction

An abstraction can be defined through an abstract domain, a precision, and a transfer function [14].

Definition 3. *An abstract domain is a tuple $(S, expr)$, where S is a set of abstract states*[1]*, and $expr : S \mapsto FOL$ is an expression function mapping an abstract state to a first-order logic formula describing the state.*

We assume that CFA locations of all processes are explicitly tracked in all abstract domains (and refer to the location of process p in a state s by $s(p)$ as introduced earlier). An abstract state s represents a concrete state c denoted by $c \models s$ if $c(p) = s(p)$ for each process p, and $expr(c)$ implies $expr(s)$. In our notation, we use s for abstract states and c for concrete states. An abstract state is an error state if any process is in an error location.

An abstract trace $w = \alpha_1 ... \alpha_k$ from the abstract state s_0 ($s_0 \xrightarrow{\alpha_1} ... \xrightarrow{\alpha_k} s_k$) is *feasible* if w is also a trace in the concrete state space ($c_0 \xrightarrow{\alpha_1} ... \xrightarrow{\alpha_k} c_k$) with $c_i \models s_i$; otherwise, w is spurious from s_0. The abstract state space overapproximates the behavior of the concrete state space: if there is a trace w from a concrete state c, then w is also a trace in the abstract state space from all abstract states s with $c \models s$ [14].

The precision (Π) describes which aspects the abstraction keeps, defined differently for each abstract domain. The *variables of a precision $vars(\Pi)$* are the variables that may appear in abstract state expression formulae. As a consequence, the abstraction tracks no information about variables in $V \setminus vars(\Pi)$. The transfer function calculates the successor states of an abstract state with respect to a statement and a precision.

[1] Abstract states are usually defined as a semi-lattice with a partial order [14], but we do not need those details for this paper, so we simplify.

Two frequently used abstract domains are explicit-value [17] and predicate abstraction [31]. In explicit-value abstraction, an abstract state is defined by the CFA locations of processes and an abstract variable assignment. The precision is the subset of variables $\Pi \subseteq V$ that are explicitly tracked in this abstraction, therefore $vars(\Pi) = \Pi$, here. Values of other variables are unknown in all abstract states. The expression function of an abstract state is defined similarly to concrete states in Section 3.1: variables whose values are unknown in a state are simply omitted. The result of the transfer function is based on the strongest post-operator under abstract variable assignment [17]. In predicate abstraction, an abstract state is defined by the CFA locations of processes and a combination of first-order logic (FOL) predicates [31]. The precision is a set of FOL predicates (e.g., `x > 0`, `y = z`) that are tracked in the abstraction; $vars(\Pi)$ is the set of variables appearing in the tracked predicates. The expression function of an abstract state is the combination of FOL predicates that describes the state [31].

4 Statement Reduction during Dynamic Analysis

This section presents a method for simplifying the statement of an action before calculating the successors of the current state with respect to the action. Basically, when there is no possible interleaving of threads from the current state where the value of a written variable is read by any relevant statement regarding the verified property, we do not evaluate the expression writing the variable.

4.1 Data-Flow Graph with Precision

First, let us formalize the connection between actions of the program when one action uses the result of another action.

Definition 4. *Let α, β be actions, and Π be the precision of the abstraction. We say that β* observes *α with precision Π if $written(\alpha) \cap read(\beta) \cap vars(\Pi) \neq \emptyset$.*

An action α is transitively observed *by an action β in a trace $w = w_1 ... w_n$ if there is a sequence of indices $i_1, ..., i_m$ $(1 \leq i_1 < ... < i_m \leq n)$ such that w_{i_j} is observed by $w_{i_{j+1}}$ for each $1 \leq j < m$, and $w_{i_1} = \alpha$, $w_{i_m} = \beta$.*

Note that the sequence of indices in the definition does not necessarily contain adjacent indices (i.e., i_{j+1} is not necessarily $i_j + 1$): for example, in the trace $x = 1, z = 1, y = x$, the last action transitively observes the first with indices $i_1 = 1$ and $i_2 = 3$ in the definition. Each action α transitively observes itself as the trace consisting of the single action α fulfills the conditions of the definition. Also note that this is an over-approximation of possible data-flow between α and β, since it is possible that a variable is rewritten before it is observed (e.g., actions $x = 1$, $x = 2$, $y = x$ in this order).

We build an abstract data flow graph whose nodes are statements of the program and a directed edge represents an observation between the connected nodes, i.e., the target action observes the source of the edge. There are two types of edges: in-process (**D**irect) and inter-process (**I**ndirect) observation.

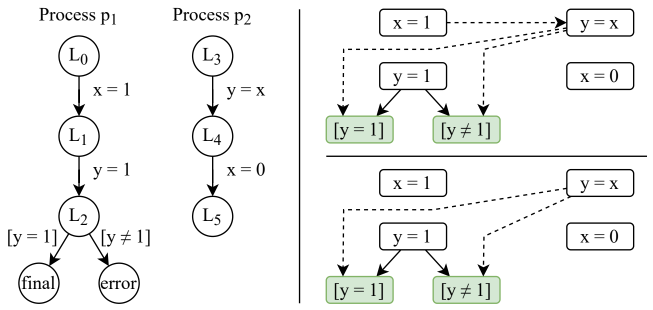

Fig. 2: CFA of two processes and data-flow graphs with different precisions

Definition 5. *An abstract data-flow graph is a tuple $G = (A, D, I, \Pi)$ where:*

- *A is the set of actions of the program (the nodes of the data flow graph),*
- *$D \subseteq A \times A$ is the set of direct observation edges: $(\alpha, \beta) \in D$ if β observes α with Π, $p_\alpha = p_\beta$, and β is reachable from α in the CFA of their process,*
- *$I \subseteq A \times A$ is the set of indirect observation edges: $(\alpha, \beta) \in I$ if β observes α with Π and $p_\alpha \neq p_\beta$.*[2]

The data-flow graph can be precomputed for the state space exploration and the same data-flow graph can be used for the full state space exploration without the need for updating it as long as the precision of the abstraction is the same. To collect direct observation edges, the CFA is traversed from each action α, and for each action β reachable from α, (α, β) is added to D if β observes α. For inter-process observation, we simply iterate over the actions of all other processes and add an indirect observation edge wherever needed. So the data-flow graph can be built in polynomial (quadratic) time in the number of CFA edges.

Example 1. Let us have the simple program from Figure 1a: its CFA is shown in Figure 2. The figure also shows two abstract data-flow graphs: the upper one with a precision where some information is tracked about both x and y ($vars(\Pi) = \{x, y\}$); below, we have no information about x ($vars(\Pi) = \{y\}$). Therefore, no edges start from actions assigning x in the second graph. Solid edges are direct observation edges, dashed edges represent inter-process observations.

4.2 Simplifying Statements On-the-Fly Based on Data-Flow

Let Π be the precision of the abstraction, and $G = (A, D, I, \Pi)$ the computed abstract data-flow graph. Let s be a state, $\alpha \in outgoing(s)$: our goal is to decide whether α can be transitively observed later during the program execution in a relevant way. We target reachability properties, so relevant actions are the actions

[2] On the implementation side, when threads can be created and terminated dynamically, several threads can have the same CFA process. In that case, inter-process observation edges may exist between actions of the same CFA process.

with guard conditions since reachability of error locations of the CFA can only be blocked by conditional statements. We will refer to these relevant actions as *real observers*. Real observers are colored in Figure 2. Thus, the evaluation of α can be skipped if there is no trace from the current state where a *real observer* transitively observes α[3]. This can be decided using the data-flow graph. To formalize this idea, we introduce the following definitions:

Definition 6. *Let s be an abstract state and p be a process. Let $reachable(s,p)$ denote the set of actions such that $\alpha \in reachable(s,p)$ if there is an abstract trace w in the abstract state space from s with $\alpha \in w$ and $p_\alpha = p$.*

Technically, $reachable(s,p)$ is the set of actions that could be executed by p at some point after s. Intuitively, if α is transitively observed by an action β in a trace starting from the current state s, then there is a path in the data-flow graph from α to β only passing through graph nodes (actions) which can still be reached from s by one of the processes. Formally, we define conditions for the enabledness of the data-flow graph edges:

Definition 7. *Let s be an abstract state, and $G = (A, D, I, \Pi)$ an abstract data-flow graph.*

- *An edge $(\alpha_1, \alpha_2) \in D$ is* enabled *in s if $\alpha_1, \alpha_2 \in reachable(s,p)$ for some process p.*
- *An edge $(\alpha_1, \alpha_2) \in I$ is* enabled *in s if $\alpha_1 \in reachable(s,p_1)$ and $\alpha_2 \in reachable(s,p_2)$ for some processes $p_1 \neq p_2$.*

Rephrasing the previous paragraph: if there is a trace from s where α is transitively observed by an action β, then there is a sequence $\alpha_1, ..., \alpha_n$ such that $\alpha_1 = \alpha$, $\alpha_n = \beta$, $(\alpha_i, \alpha_{i+1}) \in D \cup I$ for each $1 \leq i < n$, and (α_i, α_{i+1}) is enabled in s. Using the definition, deciding the enabledness of a data-flow graph edge amounts to answering reachability questions in the state space (see Definition 6) which is also the original purpose of the verification of reachability properties: seemingly, the problem has not become easier. However, $reachable(s,p)$ can be over-approximated by checking reachability in the CFA of the program[4].

Example 2. Let us continue our example from Figure 2 with a precision Π such that $vars(\Pi) = \{x, y\}$ (i.e., the upper data-flow graph in Figure 2). In the initial state where both processes are in their initial locations (L_0 and L_3), all actions may be reachable in the future by one of the processes since we over-approximate reachability in the state space by reachability in the CFA. Thus, all data-flow graph edges are enabled, so there is a path of enabled data-flow graph edges from both outgoing actions $x = 1$ and $y = x$ to a real observer (e.g., to $[y = 1]$).

[3] Note that based on the reflexivity of the transitive observation relation, conditional statements are never simplified.

[4] This over-approximation would be too coarse for the original reachability question of the verification in most cases. However, it can be effectively used for our purposes to answer reachability questions on a lower level.

However, if we have a state where the processes are in locations L_0 and L_4, then $y = x$ can never be reached from this state, so all data-flow graph edges leaving or targeting $y = x$ are disabled. That is, there is no path from $x = 1$ and $x = 0$ to a real observer in this state, so these actions are not transitively observed by a real observer, and thus, do not have to be evaluated from this state.

This example also shows a great advantage and novelty of our algorithm over existing cone-of-influence and program slicing techniques: some statements ($x = 1$ in our case) can be removed in certain states even though the same statement may be important and needs to be preserved in other states.

Using an adequate data structure, edge enabledness in the data flow graph can be over-approximated in constant time using CFA reachability information. For indirect edges, CFA reachability information can either be stored in a 2D array (with constant time indexing) or a more memory-efficient, but slightly more over-approximating approach based on strongly connected components can be used (by storing the CFA strongly connected component id number for each CFA edge and comparing these ids on-the-fly). All direct observation edges reachable in G from an action $\alpha \in \mathit{outgoing}(s)$ are enabled based on Definition 7.

For each action $\alpha \in \mathit{outgoing}(s)$, we traverse the data-flow graph from α in the way introduced above. If a real observer is reached, then the value produced by α is used (or at least may be used, c.f., the applied over-approximations), so we evaluate α properly to calculate the successor states $\alpha(s)$. However, if no real observer is reached, then the value is unused, making α unnecessary to evaluate. Instead, the successor state s' can be the state differing from the current state s only in the location of the process of α: $s'(p_\alpha)$ is the target location of α. Specifically, the following method is used to determine the successor states when there is no real observer of α: for the single variable $v \in \mathit{written}(\alpha)$, if $v \in \mathit{vars}(\Pi)$, the original statement assigning a new value to v is replaced by a *havoc* v statement; if $v \notin \mathit{vars}(\Pi)$, the original statement is removed (more precisely replaced with a *no operation* statement that has no effect). Using *havoc* on the variables tracked in the current abstraction is necessary for the refinement step of CEGAR (see Section 4.3 for more details).

Algorithm 1 summarizes the presented method of statement simplification based on dynamic data-flow analysis. Line 9 corresponds to the case when simplification is not possible (α has a real observer). Lines 12-13 are applied when α does not have a real observer but the written variable of the action is tracked in the abstraction: so α is replaced by a *havoc* statement. When we have track no information about the written variable in the current abstraction, α is completely eliminated (only the location is updated) in lines 15-17.

Theorem 1 proves that using Algorithm 1 for state space exploration yields correct results, that is, it reaches an error state whenever an error state is reachable with a feasible trace in the original state space. By original state space, we mean the abstract state space explored without the introduced statement simplification (i.e., for each $\alpha \in \mathit{outgoing}(s)$, the successor states $\alpha(s)$ are all explored).

Algorithm 1: State Space Exploration with Statement Simplification

```
   Input: $s_0, \Pi$                                  /* initial state, precision */
   Output: $verdict$                                  /* safe/unsafe */
1  $G \leftarrow$ construct abstract data-flow graph with $\Pi$
2  $waitlist \leftarrow \{s_0\}$
3  while $waitlist \neq \emptyset$ do
4      $s \leftarrow$ remove an item from $waitlist$
5      if $s$ is an error state then return unsafe
6      else
7          foreach $\alpha \in outgoing(s)$ do
8              if $\exists$ path in $G$ of enabled edges in $s$ from $\alpha$ to a real observer then
9                  $successors \leftarrow \alpha(s)$
10             else
11                 if $written(\alpha) = \{v\}$ and $v \in vars(\Pi)$ then
12                     $\alpha' \leftarrow havoc\ v$
13                     $successors \leftarrow \alpha'(s)$
14                 else
15                     $s' \leftarrow s$
16                     $s'(p_\alpha) \leftarrow$ target location of $\alpha$
17                     $successors \leftarrow \{s'\}$
18             $waitlist \leftarrow waitlist \cup successors$
19 return safe
```

Theorem 1. *Algorithm 1 returns an unsafe verdict whenever an error state is reachable in the concrete state space.*

Proof. A reachable error state in the concrete state space means that the original abstract state space contains a feasible abstract error trace. We prove that if we take *successors* instead of $\alpha(s)$ in a step of the algorithm, then if there is a feasible abstract error trace from s starting with α, there is also a feasible abstract error trace from some $s' \in successors$. We have the following cases:

1. α is transitively observed by a real observer.
   Then α is not simplified, so $successors = \alpha(s)$. Naturally, if there is a feasible abstract error trace from s in the form $\alpha.w$, then w is a feasible abstract error trace from at least one element of $successors = \alpha(s)$.
2. α is not observed transitively by a real observer, and $v \notin vars(\Pi)$ for the single item $v \in written(\alpha)$.[5]
   In this case, α practically has no effect since no information is tracked about v in the current abstraction. So an assignment of v only performs a location update for p_α. This is exactly how s' defined in lines 15-16, so $successors = \alpha(s)$ in this case, as well. Similarly to case 1, there is a feasible abstract error from at least one element of $successors = \alpha(s)$.

[5] Note that $written(\alpha)$ has exactly one item when α is not transitively observed by a real observer because α must be an assignment then.

3. α is not observed transitively by a real observer, and $v \in \mathit{vars}(\Pi)$: α is replaced by a *havoc* statement.
   A feasible abstract error trace $\alpha.w$ from s implies that there is a concrete state c with $c \models s$ such that $\alpha.w$ is a trace from c to a concrete error state. Note that an unobserved α can be a deterministic or non-deterministic assignment. If we have a non-deterministic assignment, we are back in the previous case since practically, α is not replaced (a *havoc* replaced with a *havoc* on the same variable). So we consider α as a deterministic assignment, that is $\alpha(c) = \{c'\}$. Thus, w is an error trace from c'. Now, if we take α' instead of α, then $c' \in \alpha'(c)$ since a *havoc* means that v can get any value from its domain including the value $c'(v)$ originally assigned by α. Based on the abstraction, for each concrete state $\hat{c} \in \alpha'(c)$ there is an abstract state $\hat{s} \in \alpha'(s)$ such that $\hat{c} \models \hat{s}$. Therefore, for $c' \in \alpha'(c)$, there is an abstract state $s' \in \alpha'(s)$ with $c' \models s'$. This way, w being an error trace from c' implies that w is a feasible abstract error trace from $s' \in \mathit{successors} = \alpha'(s)$.

As the property proven above is preserved in each exploration step, it follows by induction that if a feasible abstract error trace is available from the initial state, then there is a feasible abstract error trace in the state space explored by Algorithm 1, as well, which proves the theorem. □

4.3 Statement Simplification with CEGAR

Counterexample-Guided Abstraction Refinement (CEGAR) [22] is an abstraction-based model checking algorithm, starting from a coarse abstraction and iteratively refining it until it can prove or disprove the analyzed property. Its core is the *CEGAR-loop* consisting of the *abstractor* and the *refiner*. The abstractor builds the abstract state space over an abstract domain with a given precision. Since this is an over-approximation of the original concrete state space, if no abstract error state is reachable, the concrete model is also safe. On the other hand, when an abstract error is reachable, the refiner checks whether it is a *feasible* or a *spurious* abstract counterexample. The counterexample is an alternating sequence of abstract states and actions from the initial abstract state to an abstract error state. The refiner checks whether this trace is feasible or not, that is, whether there is a concrete variable assignment for each state of the trace that does not contradict the abstract state expressions and the actions of the trace. If the abstract trace is feasible, the program is unsafe. If the trace is spurious, the precision is refined (i.e., the refiner provides a new refined precision that could better establish the safety or reveal the faulty behavior of the program). The abstract state space is built with this refined precision in the next iteration.

Our proposed algorithm can be used by the abstractor of CEGAR for abstract state space exploration. However, it is important that a potential counterexample provided to the refiner must contain the original actions even if our algorithm simplified them during the state space exploration. For a reason, consider a program with a single process which is *Process* p_1 from Figure 2, but let the action from L_1 to L_2 be $y = x$. Let our precision only track information about

x: $vars(\Pi) = \{x\}$. An error state is reachable in the abstract state space with the clearly spurious trace ($x = 1$, $y = x$, $[y \neq 1]$). However, our algorithm simplifies $x = 1$ and $y = x$ since they are not observed by a conditional action with this precision. If the refiner only sees the simplified actions in the trace, i.e., (*havoc* x, *no operation*, $[y \neq 1]$), the contradiction cannot be spotted. Concluding that the counterexample is feasible, it would give a wrong *unsafe* verdict (instead of spotting the contradiction and providing a better refined precision).

Our algorithm presented previously can also be used in verification algorithms other than CEGAR. In such a case, it may be possible to drop lines 11-14 of Algorithm 1 and always use lines 15-17 to define the successor state when α is not observed transitively by a real observer. However, we focus on CEGAR as the base algorithm in this paper, making the *havoc* necessary when the assignment of a variable in the precision is simplified. If the successor state is defined as in lines 15-16 instead of using a *havoc*, the refiner may see a contradiction. For example, assume that the value of variable x is explicitly tracked in a CEGAR iteration, and the abstractor finds a counterexample trace. The trace contains a state s where $x = 0$, and the next action α assigns 1 to x. However, our algorithm noticed that α cannot be transitively observed by any real observer, so it skipped the evaluation of the statement of α. Based on lines 15-16 of the algorithm, the value of x would be the same (namely 0) in the state s' after α in the trace. Then, the refiner finds a contradiction here as the value of x cannot be 0 after an action that assigns 1 to x (we have seen in the previous paragraph that the counterexample must contain the original actions). On the other hand, the precision could not be refined based on this misleading contradiction, and the CEGAR algorithm could get stuck in endless iterations.

Applying a *havoc* statement instead of the unevaluated assignment expression overcomes this problem, as the *havoc* statement covers the behavior of the original assignment whatever value it would assign. Going back to our example, the *havoc* statement would erase the value of x from s', so it is not a contradicting state after α. Evaluating the *havoc* statement is still a simple task, so it is still worth replacing the original assignments with it.

It is worth mentioning that our algorithm cannot introduce new spurious counterexamples and degrade performance this way. Intuitively, guard conditions cannot get enabled as a side-effect of our algorithm since Algorithm 1 only simplifies statements that are not observed transitively by any conditional statement. Thus, the evaluation of guard conditions is not affected, so new (spurious) counterexamples cannot emerge. As for originally feasible counterexamples, they remain feasible with our algorithm as feasible traces are always available in the abstract state space explored by our algorithm based on the proof of Theorem 1.

5 Experimental Evaluation

In this section, we evaluate the efficiency of our algorithmic contributions. The goal of our experiment is to evaluate the performance of our proposed dynamic statement simplification algorithm. We refer to our novel algorithm as a dynamic cone-of-influence-based statement simplification algorithm (or DCOI for short) in our experiments. We compare our algorithm to a baseline using static cone-of-influence (SCOI) to see how much further reduction is achieved (that is when DCOI and SCOI are both enabled). Since DCOI can do the job of SCOI so to say (and thus, completely replace SCOI), it is also meaningful to investigate the performance with only DCOI while SCOI is disabled. We are interested in different abstract domains, therefore we investigate the effect of our proposed algorithm in two abstract domains frequently used in state-of-the-art verification tools [11]: explicit-value abstraction (later EXPL) [17] and (Cartesian) predicate abstraction (later PRED) [9].

We implemented our algorithm as an open-source extension of the Theta verification tool [48] which already had a built-in CEGAR algorithm and a static cone-of-influence preprocessing step, and had prior support for multi-threaded C programs including a partial order reduction algorithm [8]. We also compare our results to other state-of-the-art verifiers using abstract state space exploration.

In our experiments, we use a *static* partial order reduction (POR) algorithm [2, 8] during the state space exploration in a way that POR is applied first as a filter on the outgoing actions of states, and then the DCOI technique is used to simplify the subset of outgoing actions selected by POR. Since the applied static POR only takes actions into account that might be executed later, the soundness of the POR algorithm is not affected by our proposed algorithm (as the effect of DCOI is materialized in the already explored part of the state space).

Research Questions To evaluate the presented algorithm, we aim to answer the following research questions concerning metrics relevant to it:

RQ1 What proportion of statements can be simplified or completely eliminated using the proposed algorithm?
RQ2 How is the time of successor state calculation affected by our algorithm?
RQ3 How is the overall verification performance affected by the algorithm?
RQ4 What practical performance improvement can we observe on programs where theoretically exponential gain is expected?

Experimental Configuration In our experiments, we executed different configurations of Theta over a set of input programs written in C from the concurrency safety reachability category benchmark suite[6] of SV-COMP [11] (715 tasks) that is parsable by Theta (598) for RQ1-RQ3 and a direct implementation of Figure 1b for RQ4 with $N = 2^{0 \leq i \leq 7}$. We executed 6 configurations on the SV-COMP benchmarks: both abstract domains (EXPL, PRED) with the three different cone-of-influence methods (SCOI, SCOI+DCOI, DCOI). The benchmark tests were executed on virtual machines with Intel Core (Haswell) processors, 2 dedicated CPU cores were allocated to each task. Each verification task had a time

[6] https://gitlab.com/sosy-lab/benchmarking/sv-benchmarks

limit of 900 seconds (1800 seconds for RQ4) and a memory limit of 15GB. We used a sequence interpolation-based refinement strategy for the refinement step of CEGAR, and depth-first state space exploration with thread-safe large-block encoding and a static abstraction-based partial order reduction algorithm [8] in the abstraction phase. We used atoms as the basis of predicate splitting for the predicate domain; and we used a maximum number of enumerated successor states (maxenum) of 1 for the explicit domain [32]. Our backend SMT solver was Z3. For the exponential gain program, we used the predicate abstract domain with an initial precision obtained by extracting branching conditions from the program; and we also applied a simple static partial order reduction algorithm [2] after applying `DCOI` (the same POR algorithm is used when `DCOI` is disabled for a fair comparison).

5.1 Experiment Results

In the concurrency safety benchmark suite, Theta was able to parse 602 programs. No configuration provided incorrect results (where the verdict reported by Theta differs from the expected result). Table 1 shows the results for different metrics aggregated by configuration. For a fair comparison, the aggregated values are calculated over the common subset of correctly solved tasks by abstract domain: a common subset of 332 tasks was solved with the configurations using explicit-value abstraction, and 350 with predicate abstraction.

The *simplified by* `DCOI` column shows the average proportion of simplified statements (including statements replaced by *havoc* and completely eliminated statements) simplified by `DCOI` compared to all statements. The *successor calculation* and *CPU time* columns are the sum of successor state calculation and CPU times of commonly solved tasks.

The results confirm the reduction potential of our algorithm: configurations using `DCOI` greatly outperform (depending on the abstract domain) the baselines without `DCOI` in terms of both successor state calculation and overall verification performance. It is also in line with our expectations that using `DCOI` without `SCOI` leads to slightly better performance since `DCOI` can also eliminate the statements removed by `SCOI` with a minor overhead while the time of `SCOI` is completely spared. Let us interpret the results by answering the research questions:

RQ1 `DCOI` simplifies 19.6% of all statements on average (14% completely eliminated, while 5.6% replaced by *havoc* with explicit-value abstraction; 14.4% and 5.2% respectively with predicate abstraction). This confirms the relevance of our method: a significant subset of statements is unnecessary in certain thread interleavings for verifying the given property of the program.

RQ2 Our algorithm greatly reduces the time of successor state calculation as testified by the results in Table 1: by 29.9% with explicit abstraction and 31.8% with predicate abstraction. A significant part of successor state calculation is taken by SMT-solvers solving SMT problems (especially when using predicate abstraction). Thus, the overall system load is significantly decreased by reducing the SMT problem solving time.

domain	coi	simplified by DCOI	successor calculation	CPU time	solved tasks
EXPL	**SCOI**	0%	1254s	5581s	332
	SCOI+DCOI	19.5%	880s	5548s	332
	DCOI	19.6%	879s	5376s	334
PRED	**SCOI**	0%	22168s	40496s	352
	SCOI+DCOI	19.7%	15289s	35102s	358
	DCOI	19.6%	15118s	34582s	358

Table 1: Different metrics of the evaluation

RQ3 Overall performance is also improved (see CPU time in Table 1), especially for predicate abstraction: `DCOI` reduces the overall CPU time compared to the baseline by 3.7% using explicit abstraction, and by 14.6% using predicate abstraction. It was our expectation to have better improvement with predicate abstraction since it is more costly to compute which tracked predicates (or their negations) are entailed by the previous abstract state and the current action. Thus, successor state calculation takes a greater portion of the verification in predicate abstraction leading to a greater impact of our algorithm. The number of solved tasks is only slightly increased (i.e., the number of tasks where the baseline configuration hit the timeout, but our approach enabled Theta to verify the program within the time limit) probably because the complexity of input tasks is not linearly increasing. The overhead of our algorithm (the time of building and traversing the data-flow graph) is not huge though not completely negligible: 197 seconds and 206 seconds for `SCOI+DCOI` and `DCOI`, respectively, aggregated for all tasks with explicit-value analysis which is 3.6% and 3.8% of all CPU time. Similarly, our algorithm ran for a total of 342 and 345 seconds with predicate abstraction taking 1% of all CPU time in both cases.

RQ4 Even though the baseline used the same static COI and the same partial order reduction algorithm, it could only solve the three smallest tasks in the set (up to $N = 4$) within the time limit, whereas `DCOI` was able to verify 8 tasks, up to $N = 128$ as seen in Figure 3. Indeed, our algorithm scales very well in some cases. We tried to solve these tasks with state-of-the-art verifiers, as well. Even the best tools in SV-COMP 2024[7] (Dartagnan and Ultimate GemCutter, winner and second place in the concurrency category of SV-COMP'24 [11]) cannot solve these tasks for $N > 4$ within the time limit which highlights the potential of our algorithm.

Comparison with the state-of-the-art. We also compare the average performance of our solution to other state space exploration algorithms in state-of-the-art verifiers. We select the best performing verifiers from SV-COMP 2024 concurrency category [11] that use some kind of abstract state space exploration algorithm. Two such state-of-the-art tools are CPAchecker [7] and

[7] The official results of SV-COMP 2025 are not published at the time of writing.

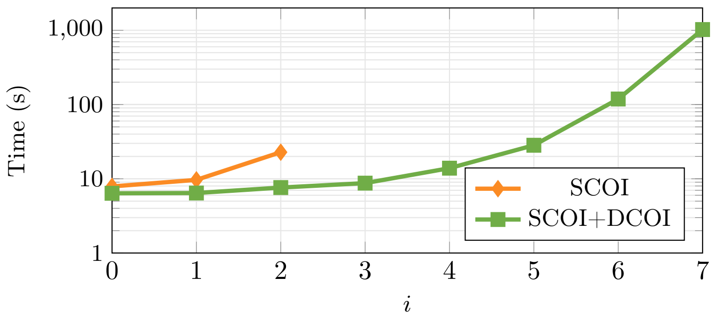

Fig. 3: Execution time given i for $N := 2^i$ in Figure 1b

PIchecker [47]. Other successful tools in SV-COMP are either bounded model checkers (such as Dartagnan [41], Deagle [34], and CSeq [25]) that would be unfair choices for comparison with a complete model checking algorithm; or use a conceptually different trace abstraction algorithm (such as Ultimate Automizer [35], GemCutter [38], and Taipan [26]); or use some advanced algorithm or tool selection strategies without implementing own analyses (such as PeSCo [46], and Graves [40]).

We executed the two verifiers on the same SV-COMP tasks, on the same hardware with the same limits. CPAchecker uses a standard state space exploration technique for multi-threaded programs combined with a BDD analysis [13] and achieved to solve 346 tasks correctly[8]. PIchecker is built on CPAchecker and has multiple analyses for concurrent software [47]. One that uses CEGAR and Craig interpolation could verify 246 tasks while its main method, a BDD analysis with an elaborate partial order reduction algorithm could verify 388 tasks. Our solution solving 358 tasks thus ranks second among these similar analyses. While our contribution does not bring Theta to the first place among these tools, it reduces the advantage of PIchecker. As a reference, the most solved tasks by a single tool was 452 in the SV-COMP 2024 concurrency reachability category [11], though that tool used bounded model checking.

5.2 Threats to Validity

Internal validity. We used BenchExec [19] to ensure the accuracy of our experiments executed on virtual machines in our university's cloud computing platform. External factors such as shared resources may have influenced the results.

External validity. The SV-COMP benchmark suite is considered a de facto standard for academic benchmarking in software verification. Theta can only parse a limited subset of SV-COMP concurrent benchmark programs which further reduces generalizability. However, there might be more redundant model el-

[8] CPAchecker also has a predicate analysis for concurrent software [16] but the algorithm is a bit dated and this analysis can only verify 159 tasks.

ements in real-world software than in the simplified programs of the SV-COMP benchmarks, making our technique disadvantaged on the benchmark set.

Construct validity. Evaluation metrics were carefully chosen to accurately describe the performance of our algorithm: both *end-user* statistics (such as CPU time, number of solved tasks) and *backend-related* information (such as the ratio of simplified statements, successor state calculation time) were used.

6 Conclusion

In this paper, we have presented a novel statement reduction algorithm based on dynamic data-flow analysis to aid abstract state space exploration of concurrent programs. Our method is based on a similar idea to cone-of-influence algorithms, however, our algorithm performs a more fine-grained analysis resulting in more extensive reduction of model elements. We have proven its correctness and discussed its integration into the abstraction-based verification algorithm CEGAR. The evaluation of the algorithm shows that our approach can simplify or completely eliminate a great proportion of statements which leads to a significant improvement in both successor state calculation time and overall verification time, especially in cases where successor state calculation takes a significant proportion of verification time, such as in the case of predicate abstraction. Therefore, the presented algorithm is worth implementing in a model checking tool that verifies concurrent software.

References

1. Abdulla, P.A., Aronis, S., Jonsson, B., Sagonas, K.: Optimal dynamic partial order reduction. pp. 373–384. ACM (2014). https://doi.org/10.1145/2535838.2535845
2. Abdulla, P.A., Aronis, S., Jonsson, B., Sagonas, K.: Comparing Source Sets and Persistent Sets for Partial Order Reduction. Lecture Notes in Computer Science, vol. 10460, pp. 516–536. Springer (2017). https://doi.org/10.1007/978-3-319-63121-9_26
3. Abdulla, P.A., Aronis, S., Jonsson, B., Sagonas, K.: Source Sets: A Foundation for Optimal Dynamic Partial Order Reduction. J. ACM **64**(4), 25:1–25:49 (2017). https://doi.org/10.1145/3073408
4. Agarwal, P., Chatterjee, K., Pathak, S., Pavlogiannis, A., Toman, V.: Stateless model checking under a reads-value-from equivalence. Lecture Notes in Computer Science, vol. 12759, pp. 341–366. Springer (2021). https://doi.org/10.1007/978-3-030-81685-8_16
5. Agrawal, H., Horgan, J.R.: Dynamic program slicing. pp. 246–256. ACM (1990). https://doi.org/10.1145/93542.93576
6. Aronis, S., Jonsson, B., Lång, M., Sagonas, K.: Optimal Dynamic Partial Order Reduction with Observers. Lecture Notes in Computer Science, vol. 10806, pp. 229–248. Springer (2018). https://doi.org/10.1007/978-3-319-89963-3_14
7. Baier, D., Beyer, D., Chien, P., Jankola, M., Kettl, M., Lee, N., Lemberger, T., Rosenfeld, M.L., Spiessl, M., Wachowitz, H., Wendler, P.: Cpachecker 2.3 with strategy selection - (competition contribution). Lecture Notes in Computer Science, vol. 14572, pp. 359–364. Springer (2024). https://doi.org/10.1007/978-3-031-57256-2_21

8. Bajczi, L., Telbisz, C., Somorjai, M., Ádám, Z., Dobos-Kovács, M., Szekeres, D., Mondok, M., Molnár, V.: Theta: Abstraction based techniques for verifying concurrency (competition contribution). Lecture Notes in Computer Science, vol. 14572, pp. 412–417. Springer (2024). https://doi.org/10.1007/978-3-031-57256-2_30
9. Ball, T., Podelski, A., Rajamani, S.K.: Boolean and Cartesian Abstraction for Model Checking C Programs. Lecture Notes in Computer Science, vol. 2031, pp. 268–283. Springer (2001). https://doi.org/10.1007/3-540-45319-9_19
10. Berezin, S., Campos, S.V.A., Clarke, E.M.: Compositional Reasoning in Model Checking. Lecture Notes in Computer Science, vol. 1536, pp. 81–102. Springer (1997). https://doi.org/10.1007/3-540-49213-5_4
11. Beyer, D.: State of the art in software verification and witness validation: SV-COMP 2024. Lecture Notes in Computer Science, vol. 14572, pp. 299–329. Springer (2024). https://doi.org/10.1007/978-3-031-57256-2_15
12. Beyer, D., Dangl, M., Wendler, P.: A Unifying View on SMT-Based Software Verification. J. Autom. Reason. **60**(3), 299–335 (2018). https://doi.org/10.1007/s10817-017-9432-6
13. Beyer, D., Friedberger, K.: A Light-Weight Approach for Verifying Multi-Threaded Programs with CPAchecker. EPTCS, vol. 233, pp. 61–71 (2016). https://doi.org/10.4204/EPTCS.233.6
14. Beyer, D., Henzinger, T.A., Théoduloz, G.: Configurable Software Verification: Concretizing the Convergence of Model Checking and Program Analysis. Lecture Notes in Computer Science, vol. 4590, pp. 504–518. Springer (2007). https://doi.org/10.1007/978-3-540-73368-3_51
15. Beyer, D., Keremoglu, M.E.: CPAchecker: A Tool for Configurable Software Verification. Lecture Notes in Computer Science, vol. 6806, pp. 184–190. Springer (2011). https://doi.org/10.1007/978-3-642-22110-1_16
16. Beyer, D., Keremoglu, M.E., Wendler, P.: Predicate abstraction with adjustable-block encoding. pp. 189–197. IEEE (2010)
17. Beyer, D., Löwe, S.: Explicit-Value Analysis Based on CEGAR and Interpolation. CoRR **abs/1212.6542** (2012)
18. Beyer, D., Löwe, S., Wendler, P.: Sliced path prefixes: An effective method to enable refinement selection. Lecture Notes in Computer Science, vol. 9039, pp. 228–243. Springer (2015). https://doi.org/10.1007/978-3-319-19195-9_15
19. Beyer, D., Löwe, S., Wendler, P.: Reliable benchmarking: requirements and solutions. Int. J. Softw. Tools Technol. Transf. **21**(1), 1–29 (2019). https://doi.org/10.1007/s10009-017-0469-y
20. Blanc, N., Kroening, D.: Race analysis for systemc using model checking. ACM Trans. Design Autom. Electr. Syst. **15**(3), 21:1–21:32 (2010). https://doi.org/10.1145/1754405.1754406
21. Bozzano, M., Cimatti, A., Tapparo, F.: Symbolic fault tree analysis for reactive systems. Lecture Notes in Computer Science, vol. 4762, pp. 162–176. Springer (2007). https://doi.org/10.1007/978-3-540-75596-8_13
22. Clarke, E.M., Grumberg, O., Jha, S., Lu, Y., Veith, H.: Counterexample-guided abstraction refinement for symbolic model checking. J. ACM **50**(5), 752–794 (2003). https://doi.org/10.1145/876638.876643
23. Clarke, E.M., Grumberg, O., Long, D.E.: Model Checking and Abstraction. ACM Trans. Program. Lang. Syst. **16**(5), 1512–1542 (1994). https://doi.org/10.1145/186025.186051
24. Clarke, E.M., Klieber, W., Novácek, M., Zuliani, P.: Model Checking and the State Explosion Problem. Lecture Notes in Computer Science, vol. 7682, pp. 1–30. Springer (2011). https://doi.org/10.1007/978-3-642-35746-6_1

25. Coto, A., Inverso, O., Sales, E., Tuosto, E.: A prototype for data race detection in cseq 3 - (competition contribution). Lecture Notes in Computer Science, vol. 13244, pp. 413–417. Springer (2022). https://doi.org/10.1007/978-3-030-99527-0_23
26. Dietsch, D., Heizmann, M., Klumpp, D., Schüssele, F., Podelski, A.: Ultimate taipan and race detection in ultimate - (competition contribution). Lecture Notes in Computer Science, vol. 13994, pp. 582–587. Springer (2023). https://doi.org/10.1007/978-3-031-30820-8_40
27. Dietsch, D., Heizmann, M., Musa, B., Nutz, A., Podelski, A.: Craig vs. newton in software model checking. pp. 487–497. ACM (2017). https://doi.org/10.1145/3106237.3106307
28. Dwyer, M.B., Clarke, L.A.: Data Flow Analysis for Verifying Properties of Concurrent Programs. pp. 62–75. ACM (1994). https://doi.org/10.1145/193173.195295
29. Eilers, M., Dardinier, T., Müller, P.: Commcsl: Proving information flow security for concurrent programs using abstract commutativity. Proc. ACM Program. Lang. **7**(PLDI), 1682–1707 (2023). https://doi.org/10.1145/3591289
30. Farzan, A., Klumpp, D., Podelski, A.: Stratified Commutativity in Verification Algorithms for Concurrent Programs. Proc. ACM Program. Lang. **7**(POPL), 1426–1453 (2023). https://doi.org/10.1145/3571242
31. Flanagan, C., Qadeer, S.: Predicate abstraction for software verification. pp. 191–202. ACM (2002). https://doi.org/10.1145/503272.503291
32. Hajdu, Á., Micskei, Z.: Efficient Strategies for CEGAR-based Model Checking. Journal of Automated Reasoning **64**(6), 1051–1091 (2020). https://doi.org/10.1007/s10817-019-09535-x
33. Harman, M., Hierons, R.M.: An overview of program slicing. Softw. Focus **2**(3), 85–92 (2001). https://doi.org/10.1002/swf.41
34. He, F., Sun, Z., Fan, H.: Deagle: An smt-based verifier for multi-threaded programs (competition contribution). In: International Conference on Tools and Algorithms for the Construction and Analysis of Systems. pp. 424–428. Springer (2022)
35. Heizmann, M., Barth, M., Dietsch, D., Fichtner, L., Hoenicke, J., Klumpp, D., Naouar, M., Schindler, T., Schüssele, F., Podelski, A.: Ultimate automizer and the commuhash normal form - (competition contribution). Lecture Notes in Computer Science, vol. 13994, pp. 577–581. Springer (2023). https://doi.org/10.1007/978-3-031-30820-8_39
36. Henzinger, T.A., Jhala, R., Majumdar, R., Sutre, G.: Software Verification with BLAST. Lecture Notes in Computer Science, vol. 2648, pp. 235–239. Springer (2003). https://doi.org/10.1007/3-540-44829-2_17
37. Huang, J.: Stateless model checking concurrent programs with maximal causality reduction. pp. 165–174. ACM (2015). https://doi.org/10.1145/2737924.2737975
38. Klumpp, D., Dietsch, D., Heizmann, M., Schüssele, F., Ebbinghaus, M., Farzan, A., Podelski, A.: Ultimate gemcutter and the axes of generalization - (competition contribution). Lecture Notes in Computer Science, vol. 13244, pp. 479–483. Springer (2022). https://doi.org/10.1007/978-3-030-99527-0_35
39. Korel, B., Rilling, J.: Dynamic program slicing methods. Information and Software Technology **40**(11-12), 647–659 (1998)
40. Leeson, W., Dwyer, M.B.: Graves-cpa: A graph-attention verifier selector (competition contribution). In: International Conference on Tools and Algorithms for the Construction and Analysis of Systems. pp. 440–445. Springer (2022)
41. de León, H.P., Haas, T., Meyer, R.: Dartagnan: Leveraging compiler optimizations and the price of precision (competition contribution). Lecture Notes in Computer Science, vol. 12652, pp. 428–432. Springer (2021). https://doi.org/10.1007/978-3-030-72013-1_26

42. Loiacono, C., Palena, M., Pasini, P., Patti, D., Quer, S., Ricossa, S., Vendraminetto, D., Baumgartner, J.: Fast cone-of-influence computation and estimation in problems with multiple properties. pp. 803–806. EDA Consortium San Jose, CA, USA / ACM DL (2013). https://doi.org/10.7873/DATE.2013.170
43. Nanda, M.G., Ramesh, S.: Slicing concurrent programs. pp. 180–190. ACM (2000). https://doi.org/10.1145/347324.349121
44. Peled, D.A.: Ten Years of Partial Order Reduction. Lecture Notes in Computer Science, vol. 1427, pp. 17–28. Springer (1998). https://doi.org/10.1007/BFb0028727
45. Peled, D.A., Valmari, A., Kokkarinen, I.: Relaxed visibility enhances partial order reduction. Formal Methods Syst. Des. **19**(3), 275–289 (2001). https://doi.org/10.1023/A:1011202615884, https://doi.org/10.1023/A:1011202615884
46. Richter, C., Hüllermeier, E., Jakobs, M., Wehrheim, H.: Algorithm selection for software validation based on graph kernels. Autom. Softw. Eng. **27**(1), 153–186 (2020). https://doi.org/10.1007/S10515-020-00270-X
47. Su, J., Yang, Z., Xing, H., Yang, J., Tian, C., Duan, Z.: Pichecker: A POR and interpolation based verifier for concurrent programs (competition contribution). Lecture Notes in Computer Science, vol. 13994, pp. 571–576. Springer (2023). https://doi.org/10.1007/978-3-031-30820-8_38
48. Tóth, T., Hajdu, A., Vörös, A., Micskei, Z., Majzik, I.: Theta: a Framework for Abstraction Refinement-Based Model Checking. pp. 176–179 (2017). https://doi.org/10.23919/FMCAD.2017.8102257