MŰEGYETEM 1782

Budapest University of Technology and Economics
Faculty of Electrical Engineering and Informatics
Department of Measurement and Information Systems

Abstract Data-Flow-Based Statement Reduction for Model Checking Concurrent Software

Scientific Students' Association Report

Author:

Csanád Telbisz

Advisors:

Levente Bajczi
Dániel Szekeres

2023

Contents

Kivonat

Az utóbbi évtizedekben a többmagos processzorok és a többszálú programok egyre nagyobb mértékű térhódítását figyelhettük meg ipari és biztonságkritikus rendszerekben a technológia ugrásszerű fejlődésének köszönhetően. A párhuzamos szoftverek verifikációja ugyanakkor jelentős kihívást jelent a szálak nagyszámú lehetséges átlapolódása miatt. A kielégítő tesztlefedettség elérése így nagy kihívást jelent, naiv modellellenőrzési technikák pedig rosszul skálázódnak, e nagyfokú bonyolultság miatt gyakorlatilag alkalmazhatatlanná válnak.

Az absztrakciós-finomítási algoritmusok hatékony technikák az állapottérben való elérhetőség vizsgálatára. Azonban az állapottér bejárása során a követő állapotok kiszámítása, vagyis az utasítások kiértékelése költséges feladat, amely gyakran SMT-probléma megoldását igényli. Másrészről viszont sok esetben egy utasítás kiértékelése nincs hatással az ellenőrzött tulajdonságra. Ilyen esetekben egyszerűsíthető az követő állapotok kiszámítása. Számos algoritmus létezik, amely statikusan elemzi a modellt és eltávolítja a modellből a nem releváns változókat vagy utasításokat. Párhuzamos szoftverekben azonban gyakori, hogy egy utasítás eredményét a szálak egy bizonyos átlapolódásában használják másik utasítások, míg egy másik ütemezés mellett nem használják. A modellt statikusan elemző algoritmusok nem tudják kiegyszerűsíteni az ilyen utasításokat.

Munkámban egy új, dinamikus megközelítést javaslok, amely absztrakt adatfolyamelemzés segítségével, illetve az egyes szálak aktuális állapota alapján felismeri, hogy egy utasítás egyszerűsíthető-e az állapottér feltárása során. Módszerem jelentősen képes csökkenteni a követő állapot számítására fordított időt, ezáltal pedig a verifikáció teljes idejét is. Végezetül kiértékelem a bemutatott algoritmus teljesítményét egy nagyszámú benchmark programhalmazon. A mérési eredmények azt mutatják, hogy az algoritmusom képes a nagy benchmark programhalmazon vett átlagban az utasítások több, mint 20%-ának egyszerűsítésére, ezáltal átlagosan akár 60%-os javulást is elérve a követő állapotok kiszámítására fordított idő terén.

Abstract

Rapid development in technology led to the increasing popularity of multi-core processors and multi-threaded programs in industrial, safety-critical systems. The verification of concurrent software poses significant challenges due to the inherent complexity arising from the multitude of potential thread interleavings. Achieving satisfying test coverage is a highly challenging task, and naive model checking techniques become practically infeasible as a result of this complexity.

Abstraction-refinement algorithms are efficient techniques for checking reachability in a state space. However, evaluating statements for calculating successor states during state space exploration is a costly task that often requires solving an SMT problem. On the other hand, in many cases, the evaluation of a statement has no effect with regard to the verified property. Successor state calculation can be simplified in such cases. Several algorithms exist that statically analyze the model and eliminate irrelevant variables or statements from the model. In concurrent software, however, it is common that the result of a statement is used in one interleaving of threads while unused in another. Algorithms that statically analyze the model cannot simplify such statements.

In this work, I propose a novel dynamic approach using abstract data-flow analysis that detects whether a statement can be simplified during state space exploration based on the actual state of each thread. My method can considerably reduce the time spent on successor state calculation and, thus, the overall time of verification. Finally, I evaluate the performance of the proposed algorithm on a large set of benchmark programs. Evaluation results show that my algorithm can simplify more than 20% of all statements on average over a large set of benchmark programs while reducing the time of successor state calculation by up to 60% on average.

Chapter 1

Introduction

Rapid development in technology led to huge advancements in microprocessor systems. Today, multi-core processors are available for various targets, from personal computers and smartphones to safety-critical systems. In a critical system, the increased computing capacity of a multi-core processor may add extra resources to the critical functionalities. This reason has lead to the increasing popularity of multi-core processors and multi-threaded programs in critical systems. Nonetheless, functionally correct behavior is still crucial in safety-critical systems: the need for safe operation and safety requirements remain a central element of critical systems.

Testing can efficiently find programming errors. However, even in a single-threaded application, testing is insufficient to prove correctness due to the large number of possible inputs. In a multi-threaded program, the number of possible executions (thread interleavings) can be exponential in the number of operations and threads. Thorough testing becomes practically infeasible when dealing with concurrency.

Formal verification can mathematically prove safety guarantees for a system. Verification is a challenging task in itself, as the number of possible behaviors can be huge. The verification task is often to determine whether an error location can be reached in the program. Basically, this question can be answered by searching the program's state space for an error state. Unfortunately, the number of states grows exponentially with the number of variables. This phenomenon is called the state space explosion problem [21].

Model checking has been a field of active research in recent decades as it is one of the most powerful software verification techniques. All model checking algorithms have to face the state space explosion problem. Furthermore, concurrency 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 algorithms avoid exploring parts of the state space when it is guaranteed that an equivalent thread interleaving is explored for each avoided trace [35]. Abstraction-based techniques reduce the size of the state space by ignoring some details of the original problem [19, 20]. Focusing on some parts of the problem while ignoring other details gives us a smaller representation of the problem. We may be able to solve the original problem by analyzing the abstract representation. If we fail to solve the problem using this representation, we can refine our abstraction by considering more details. Counterexample-guided abstraction refinement (CEGAR) is a model checking algorithm that iteratively refines the abstraction until the desired property can be verified [20]. Other abstraction-based techniques, such as the cone-of-influence reduction or program slicing, eliminate some model elements that are irrelevant with respect to the verified property [8, 29].

My approach presented in this report is also an abstraction-based technique that aims to reduce the runtime of state space exploration by simplifying certain model elements. My algorithm is based on a similar idea to cone-of-influence (COI) reduction and program slicing algorithms. Whereas COI reduction simplifies the model by eliminating completely redundant variables with respect to the verified property [8], my approach identifies and simplifies statements that are redundant in the current state of concurrent threads with respect to the verified property. Thus, my algorithm is more fine-grained in the sense that it can eliminate statements in certain contexts even if the variables used by these statements cannot be ignored entirely. My algorithm is particularly advantageous when a statement is relevant in one interleaving of concurrent threads while it is redundant in another: we can still eliminate it in the interleaving where it is redundant.

My statement simplification method is motivated by the considerable runtime of calculating successor states in SMT-based state space exploration algorithms [14, 28]. Satisfiability modulo theories (SMT) is the problem of deciding whether a given mathematical formula is satisfiable or not [7]. It is a generalization of the SAT problem, which aims to answer the same question for Boolean formulae. In SMT-based model checking, program states and statements are represented as first-order formulae. When exploring a transition from a state in the state space of a program, we have to decide whether the state formula combined with the transition's statement formula is satisfiable. If it is satisfiable, a solution of this SMT problem also provides the formula of the successor state. Many SMT solvers have been developed since the 1970s in industrial and academic projects to solve such problems efficiently [22, 6, 18]. However, it is still an expensive task to solve an SMT problem. In this work, I aim to reduce the number of SMT problems that have to be solved during state space exploration.

My goal is to identify dynamically as many redundant statements as possible in the current state space exploration context. For this, I 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 (that is, properly calculating the successor of the current state with respect to this statement), it is checked using the data-flow graph whether any other statement can use the result of the statement. In the scope of this work, I target reachability properties; thus, we are interested in whether the result of the statement is used transitively by a conditional statement of the program since conditionals influence whether some marked error locations are reachable in the model or not. This can be decided with a simple traversal of the data-flow graph. Redundant statements are eliminated, and the time of solving an SMT problem for successor state calculation is spared in such cases. I formulate my algorithm for abstract state space exploration and exploit information about the current abstraction to reduce the number of edges (data dependencies) in the data-flow graph. I discuss the combination of my algorithm with existing concurrent software verification approaches such as CEGAR and partial order reduction.

Summarizing my contributions: I take the base idea of the cone-of-influence reduction one step further by dynamically deciding whether the result of a statement can be used later. I present a novel algorithm for identifying redundant statements using an abstract dynamically updated data-flow graph. Furthermore, I discuss the necessary additions in an iterative abstraction-refinement verification scheme as well as the possibilities of using my algorithm together with a partial order reduction algorithm. I have implemented and evaluated my algorithm in the abstraction-based model checking tool Theta [37].

Chapter 2

Preliminaries

This report assumes that the reader is familiar with the basic concepts of concurrent software design and formal software verification. Nevertheless, to avoid the misunderstanding of used concepts and notions, definitions are introduced in this chapter. Furthermore, I assume that the reader is well acquainted with graph theory: I omit such definitions (e.g., the definition of strongly connected components) from this report.

2.1 Formal Verification and Model Checking

Formal software verification aims to prove certain properties of a program mathematically [16]. Verified properties can be reachability criteria (whether a certain error state is reachable with any execution of the program), memory-safety (no memory leak or other memory handling issue), or the problem of termination (whether all executions of the program terminate). In the scope of this work, reachability criteria are considered exclusively.

Model checking is a formal verification technique where properties are verified by analyzing the state space of the program [26]. Generally, the input of a model checking algorithm is a *model* and a *formal requirement*. The output of such algorithms is a verdict: the model is either *safe* (mathematically proven to be safe) or *unsafe* (a counterexample is provided where the requirement is violated). As for the formal requirement, specific points of the verified program are marked as unsafe in reachability analysis. If any possible program executions reach such a point, the reachability criterion is said to be violated.

The mathematical problem of model checking is undecidable in general. Consider any program with an error location at its exit point. To prove that this error location is unreachable is equivalent to answering whether this program always terminates. The termination problem is undecidable [38]. Verification techniques have to face this problem and provide algorithms that can be used in practical applications.

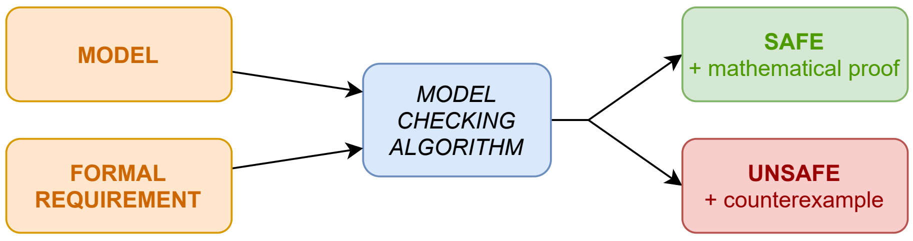


Figure 2.1: Model checking in general.

2.2 Computation Model

Though high-level languages (such as C) are convenient for developers, their verification would require a formal model of the language semantics, which can be quite complicated [4]. Thus, for verifying a program written in a high-level language, its source code is transformed into a low-level formalism that is easier to verify. For the representation of concurrent C programs, I use an extended form of control-flow automata (CFA) [10] where there can be multiple procedures (separate traditional CFAs). To keep the presentation simple, I assume that processes cannot be created or terminated dynamically and that each process has its own CFA representation (a CFA *procedure*).

Definition 1 (Multi-Threaded Control Flow Automaton). A multi-threaded CFA is a tuple $CFA = (V, P)$, where:

- V is a set of (global) variables,
- P is a set of procedures. A procedure is a tuple $p = (L, l_0, A, E)$, where:
  - L is a set of control locations,
  - l_0 is the initial location,
  - A is a set of statements. A statement can be:
    - a deterministic assignment ($v = expr$),
    - a non-deterministic assignment ($havoc\ v$), where the new value of variable $v \in V$ can be anything from its domain, or
    - a guard condition ($[cond]$).
  - $E \subseteq L \times A \times L$ is a set of transitions. A transition is a directed edge with a source control location, a target control location, and one statement. ▪

Processes communicate through shared variables. Each variable $v \in V$ has a domain D_v: the possible values for v. For the verification of reachability properties, locations can be marked as error locations: a concurrent program is safe if none of its processes can reach any error location in any possible thread interleaving.

Let us illustrate control flow automata of a single-threaded and a multi-threaded C program with the following simple examples.

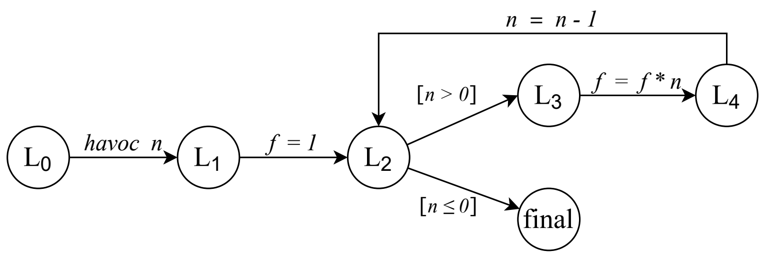
Example 1. *The program in Figure 2.2a calculates the factorial of the given number: the value of variable* `f` *is* $n!$ *at the end of the execution of this program. Figure 2.2b depicts the CFA of this program. The edges of the CFA correspond to the statements of the program (including condition checks);* l_0 *is the initial location. Note that a value from user input is assigned to* `n`*, which translates to the non-deterministic assignment* `havoc n`*.*

The source code in Figure 2.3a shows two C functions that are the functions of two different threads[1]*. The program has two global variables,* `x` *and* `y`*, which both threads can write. The* `reach_error` *function means a safety violation: I could have written* `assert(y==1)` *instead of using the* `if` *structure. However, this way, it can be easily seen how a reachability property works. Figure 2.3b depicts the CFA procedures of the two processes.* L_e *is an error location of the CFA.*

[1]In practice, in a real concurrent C program, the functions of different threads are provided to the `pthread_create` function that can start new threads. On the other hand, as I have previously noted, I assume that processes cannot be created dynamically, so I omit these details from this example as well.

```c
void main() {
  int n;
  scanf("%d", &n);
  int f = 1;
  while(n > 0) {
    f *= n;
    n--;
  }
}
```

(a) C source code



(b) CFA of the program

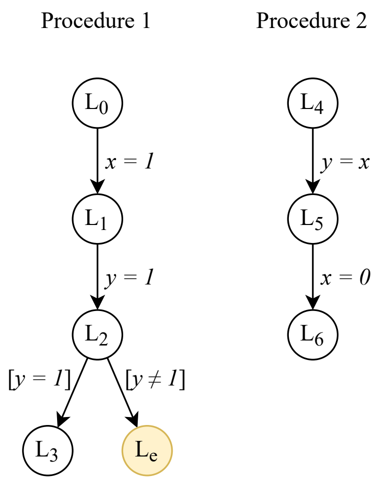
Figure 2.2: CFA of a single-threaded program

```c
int x, y;

void thread1() {
  x = 1;
  y = 1;
  if (y != 1) {
    reach_error();
  }
}

void thread2() {
  y = x;
  x = 0;
}
```

(a) C source code



(b) CFA of the program

Figure 2.3: CFA of a multi-threaded program

2.3 State Space of a Program

Before introducing the state space of a (multi-threaded) program, a general definition is given for transition systems (or state spaces).

Definition 2 (Transition System). 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. ▪

An action α is *enabled* in a state s if there is a transition $(s, \alpha, s') \in T$ for some $s' \in S$. I use the notation $s \xrightarrow{\alpha} s'$ for such a transition, and I refer to s' as a successor state of s with respect to α. Note that the definition of transition systems allows non-determinism, i.e., there can be multiple transitions from a state with the same action (c.f., non-deterministic assignments).

The state space of a program is a transition system. A *state* of a multi-threaded CFA (V, P) represents the control locations of all processes and the values of all variables at a certain point during the operation of the program: $s = (l_1, l_2, ..., l_p, d_1, d_2, ..., d_n)$, where:

- $l_j \in L_{p_j}$ is the current location of process p_j, for $1 \leq j \leq p = |P|$
  (where $P_j = (L_{p_j}, l_{p_j 0}, A_{p_j}, E_{p_j})$ is the CFA procedure of process p_j),
- $v_i = d_i$, the current value of variable v_i, for $1 \leq i \leq n = |V|$
  (where $v_i \in V$, $d_i \in D_{v_i}$).

I denote the control location of process p in state s by $s(p)$, and the value of variable v in state s by $s(v)$. I define an expression function for a state s based on the values of variables in s: $expr(s) := \bigwedge_{v \in V}(v = s(v))$. A state is an error state if any of the processes is in an error location in the state.

An initial state of a program is a state where all processes are in the initial location of their main procedure. The values of the variables in an initial state can vary based on the language the program is written in. Uninitialized variables either contain memory garbage (as local variables in C [31]), resulting in several initial states per process, or are initialized automatically to a default value (as in Java [32]), resulting in one initial state per process.

An *action* of a *transition* is a statement that the program executes. An action is enabled in a state if that statement can be performed in that state of the program. An action of a transition corresponds to a statement of a single process (processes step asynchronously). I mainly use the Greek alphabet for actions, and I write p_α for the process of action α. A transition with action α leads to a possible new state of the program after executing the statement represented by α. The location of the process of α is the source CFA location of α in the source state and the target location of α in the target state.

The statements of a CFA manifest in different ways in the state space:

- For an assignment $s \xrightarrow{v=expr} s'$, the value of v in s' is the value of expression $expr$ evaluated in s. The location of the statement's process is the source location of the statement in s and the target location in s'.
- For a *havoc v* statement, there are several transitions, $|D_v|$ exactly, leading to different states. The location of the statement's process changes with each transition as usual (the target CFA location of the statement appears in the target states of the new transitions). The value of v differs in each target state: the values range over the domain of v.
- An action with a guard condition $[cond]$ is enabled in each state s where the location of the action's process is the source location of the action, and the expression *cond* evaluates to true in s.

I also use the following notations:

- $\alpha(s) = \{s' \in S : \quad \exists(s, \alpha, s') \in T\}$, i.e., the successor states of s with respect to α,
- $enabled(s)$ denotes the set of enabled actions in s,
- $vars(\alpha)$ denotes the set of variables referenced by α,

- *written*(α) is the set of variables written by α, and
- *read*(α) is the set of variables referenced but not written by α.

Note that *written*(α) has a single item for deterministic and non-deterministic assignments, and it is an empty set for a guard condition. By $w = t_1 ... t_k$, I denote a transition sequence (or trace), and I use the following for the concatenation of transition sequences or transitions: $w.v$. I also refer to action sequences as traces. If there is a trace from a state that leads to an error state, I call this trace an error trace.

Since model checking includes searching the state space, the efficiency of a verification algorithm largely depends on the size of the state space, that is, on the number of control locations and variables in the program and the size of their domains. To represent even a single 32-bit integer variable, 2^{32} states would be necessary. With more variables, it would grow exponentially: this is called the *state space explosion problem* [21]. Thus, efficient algorithms are essential to overcome this problem. One such approach is abstraction.

2.4 Abstraction-Based Verification

Using abstraction, the size of the state space of a program can be greatly reduced. First, I define abstraction on a general level. Then I introduce an abstraction-based model checking algorithm.

2.4.1 Abstraction

An abstraction can be defined with an abstract domain, a precision, and a transfer function [13].

Definition 3 (Abstract domain). An abstract domain is a tuple $Dom = (S, expr)$ where:

- S is a lattice of abstract states, *and*
- $expr : S \mapsto FOL$ is an expression function that maps an abstract state to a first-order logic formula describing the state. ▪

I assume that CFA locations of all processes are explicitly tracked in all abstract domains, that is, each abstract state stores the locations of processes. I refer to the location of process p in the abstract state s by $s(p)$. 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 $expr(s)$. An abstract state is an error state if any process is in an error location in the state. 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 over-approximates 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 $c \models s$ [13].

The precision is the information defining which aspects are kept in the abstraction. It is defined variously in different abstract domains. The variables of a precision $vars(\Pi)$ are the variables that appear in the abstract state expression formulae. That is, no information is tracked about variables $V \setminus vars(\Pi)$ in an abstraction with precision Π. The transfer function calculates the successor states of an abstract state with respect to a statement and a precision.

Two frequently used abstract domains are explicit-value abstraction [11] and predicate abstraction [24]. 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; $vars(\Pi) = \Pi$. The 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 2.3: variables whose values are unknown in a state are simply omitted from the formula. The result of the transfer function is based on the strongest post-operator under abstract variable assignment [11]. In predicate abstraction, an abstract state is defined by the CFA locations of processes and a combination of first-order logic (FOL) predicates [24]. The precision is a set of FOL predicates (e.g., `x > 0`, `y = z`) that are tracked in this 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 [24].

2.4.2 Counterexample-Guided Abstraction Refinement

Counterexample-Guided Abstraction Refinement (CEGAR) is an abstraction-based model checking algorithm [20]. It uses abstraction to handle the problem of state space explosion. CEGAR starts from a coarse abstraction of the problem and iteratively refines the abstraction until the abstraction can prove or disprove the analyzed property. The more coarse the abstraction is, the more details are ignored. This way, there is a chance to answer the original problem by solving a much simpler abstract problem. If the abstract problem is too generic to provide an answer, the abstraction must be refined.

The core of the algorithm is the *CEGAR-loop*, which consists of two main parts: the *abstractor* and the *refiner* (see Figure 2.4).

The abstractor builds the abstract state space (or *abstract reachability graph*, ARG [12]) over an abstract domain with a precision. The abstractor tries to prove that no abstract error state is reachable in the abstract state space. Since the abstract state space is an over-approximation of the original concrete state space, if no abstract error state is reachable, the concrete model is safe as well: the algorithm terminates with a safe verdict. On the other hand, when an abstract error state is reachable, the abstractor provides an abstract counterexample to the refiner.

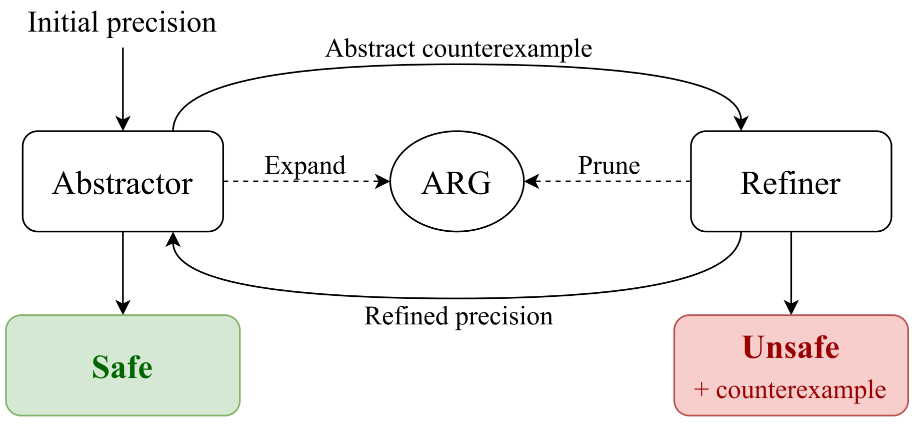
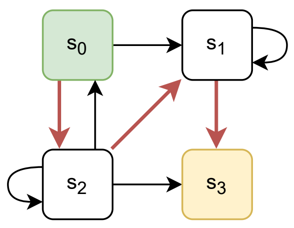
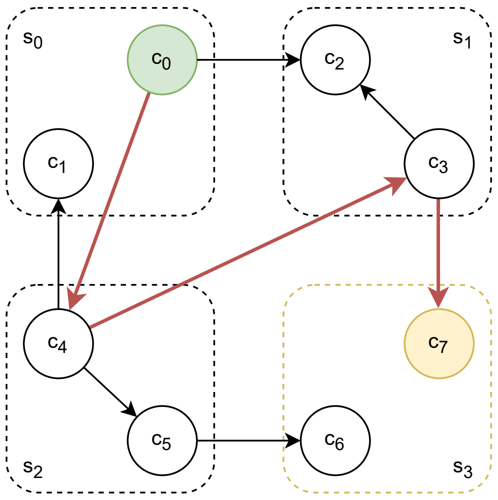


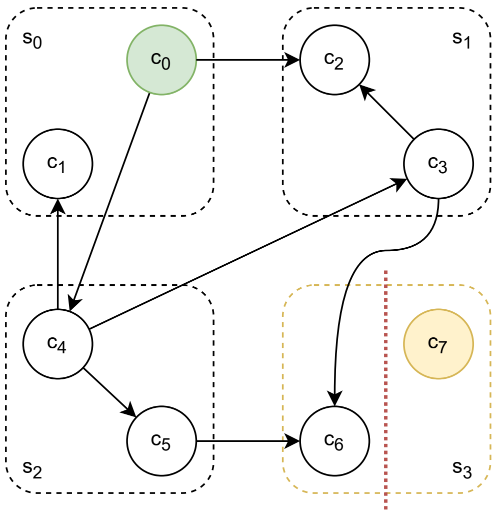
Figure 2.4: The CEGAR-loop.



(a) Abstract state space S with an abstract counterexample



(b) Feasible counterexample in S_1



(c) Spurious counterexample in S_2

Figure 2.5: CEGAR counterexamples

The refiner checks whether the given counterexample is *feasible* (a concrete error state is reachable, indeed) or *spurious* (a concrete error state is not reachable, and the abstract counterexample was the result of the abstraction) [27]. In the first case, the algorithm terminates with an unsafe verdict and the found counterexample. In contrast, in the latter case, the abstraction (the precision) is refined, and the unnecessary abstract states are removed (*pruned*) from the abstract state space. The abstract state space is built with the refined precision in the next iteration.

Example 2. *Consider the example from Figure 2.5. We have the abstract state space S from Figure 2.5a with the abstract error state s_3. The abstractor finds the abstract counterexample highlighted in Figure 2.5a. This counterexample leads from the abstract initial state s_0 to the abstract error state s_3 in the abstract state space S: $s_0 \rightarrow s_2 \rightarrow s_1 \rightarrow s_3$. The abstract state space is an over-approximation of the concrete state space. So the refiner has to decide whether the abstract counterexample is feasible or spurious.*

First, let us assume that the concrete state space abstracted by S is S_1 from Figure 2.5b. In this case, the counterexample is feasible since we can find a transition sequence for the abstract counterexample in the concrete state space starting from the initial concrete state c_0 leading to the error state c_7: $c_0 \rightarrow c_4 \rightarrow c_3 \rightarrow c_7$ with $c_0 \models s_0$, $c_4 \models s_2$, $c_3 \models s_1$, and $c_7 \models s_3$.

However, C_2 from Figure 2.5c can also be the concrete state space whose abstraction is S. The counterexample is spurious now, as there is no trace from c_0 to c_7 in S_2.

2.5 Partial Order Reduction

Generally, the execution order of operations from different threads is unspecified in a multi-threaded program. Thus, when such a program is verified, it is obviously insufficient to check only a single randomly chosen thread interleaving since the verified property may be violated in one interleaving of threads while it is not violated in another one.

2.5.1 Basic Idea of Partial Order Reduction

Unfortunately, checking every trace is too expensive as the number of possible thread interleavings can be huge. Partial Order Reduction (POR) is a well-known technique for avoiding the exploration of redundant thread interleavings during the verification of a multi-threaded program [25]. Its key idea is to define an equivalence relation on traces and explore a single representative (or as few as possible) from each equivalence class. Traces are defined to be equivalent if they can be obtained from each other by successively swapping adjacent *independent* actions. An equivalence class is called a *Mazurkiewicz trace* [34]. Intuitively, if adjacent independent actions are swapped, the outcome will remain the same: by exploring a single trace from each equivalence class, we still cover all behaviors of the system. For the above interpretation of equivalence, we need a definition of independence.

Dependency plays a key role in partial order reduction. Generally, we have two conditions for the independence of two actions [25]. The first condition is that independent actions can neither disable nor enable each other. The second property is that independent actions commute. Since the goal of POR is to avoid exploring multiple traces leading to the same state, these conditions cannot be used directly for determining dependency (two actions should be explored in both orders to decide whether they commute). Instead, practically, we can say that two actions are independent if they are from different processes, and they do not access the same shared variable. This practical interpretation satisfies the original conditions [25].

2.5.2 Partial Order Reduction Approaches

Partial order reduction methods construct a reduced transition system and explore only this smaller reduced state space instead of the original one. For the correctness of such an algorithm, it has to be guaranteed that at least one thread interleaving from each equivalence class is completely included in the reduced transition system. In practice, the reduced state space is "constructed" by calculating a sufficient subset of outgoing transitions for exploration from a state. When exploring the state space, we only proceed through transitions in the calculated subset. This way, only part of the state space is explored: the reduced state space.

There are two main approaches to partial order reduction: *static* and *dynamic* POR [3]. In the static version, the model (i.e., the CFA of the program) is analyzed, and the sufficient subset of outgoing transitions is precomputed before any transition is explored from the state. The dynamic approaches add transitions to these sufficient subsets on the fly. In this work, I will deal with a static POR algorithm similar to the one presented in [1].

2.6 Related Work

Model checking has been a field of active research since the early 1990s to this day [26, 19, 3]. Various techniques have been developed that are practically applicable even though these techniques have to face the state space explosion problem [21], such as iterative abstraction-refinement-based techniques [20] or partial order reduction [35]. These approaches reduce the size of the state space of the program in different ways.

Several approaches, such as the cone-of-influence reduction or program slicing, aim to simplify the model by eliminating redundant model elements [8, 33, 30, 29]. Some of them also use data-flow analysis for model checking [23]. I briefly introduce cone-of-influence and program slicing techniques below. These techniques typically only statically analyze and simplify the input model, which is limited compared to my dynamic data-flow analysis. These static algorithms have the advantage of being executed only once before the state space exploration, while my algorithm is performed at each successor state calculation. On the other hand, my experiments in Chapter 4 show that my algorithm does not have a significant runtime overhead.

Cone-of-influence reduction. Cone-of-influence algorithms statically analyze the data-flow of the program. Data dependency is determined for each variable. That is, for each variable v of the program, the set of variables is collected such that these variables are calculated from the value of v [8, 33]. This way, a data-flow graph is constructed where variables depend on other variables. Variables of interest that directly affect the verified property are marked first. Then, each variable v is marked if an already marked variable depends on v. This is repeated until no more variables can be marked. The unmarked (irrelevant) variables are removed from the model as they cannot affect the verified property, even indirectly. More precisely, each statement using an irrelevant variable is removed.

Program slicing. Program slicing techniques identify statements or blocks of code that can potentially affect certain variables of interest and focus on these selected aspects of semantics during the analysis [29]. In fact, the cone-of-influence reduction can be considered as a special case of program slicing. Program slicing is widely used beyond the scope of model checking: essentially, in any software engineering process where it is useful to extract parts of the program based on arbitrary semantic criteria (e.g., static code analysis or fault localization).

There are dynamic program slicing techniques; however, this means that concrete valuations of variables of interest are used for slicing [29]. None of these techniques take advantage of the interleaving of threads for further reduction, which is the base of my approach presented in this report.

Some partial order reduction works perform dynamic data-flow analysis in different ways to reduce the dependency between actions of different processes [17, 2], though they only use the results of data-flow analysis to reduce the number of explored thread interleavings and not to simplify the statements.

Chapter 3

Statement Reduction by Dynamic Data-Flow 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 accessed by any other statement relevant regarding the verified property, the expression writing the variable is not evaluated.

3.1 Data-Flow Graph with Precision

First, I present how a data-flow graph is built with a given precision. Intuitively, a data-flow graph represents dependencies between actions of a program: a directed edge points from one action to another if the latter uses the values produced by the first action. Let us formalize the connection between actions when one action uses the result of another action.

Definition 4 (Observation relation). Let α, β be actions, and Π be the precision of the abstraction. The action β *observes* α with precision Π if the following condition holds: $written(\alpha) \cap read(\beta) \cap vars(\Pi) \neq \emptyset$.

An action α is *transitively observed* by an action β in a trace $w = w_1 ... w_n$ if there are 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 each action α transitively observes itself as the trace $w = \alpha$ fulfills the conditions of the definition, so the transitive observation relation is reflexive; this relation is naturally transitive but not symmetric. 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 by another action before it is observed (e.g., actions $x = 1$, $x = 2$, $y = x$ in this order). However, these situations are relatively rare, so I will refrain from further refining the observation relation.

My algorithm builds an abstract data flow graph whose nodes are actions (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. Formally:

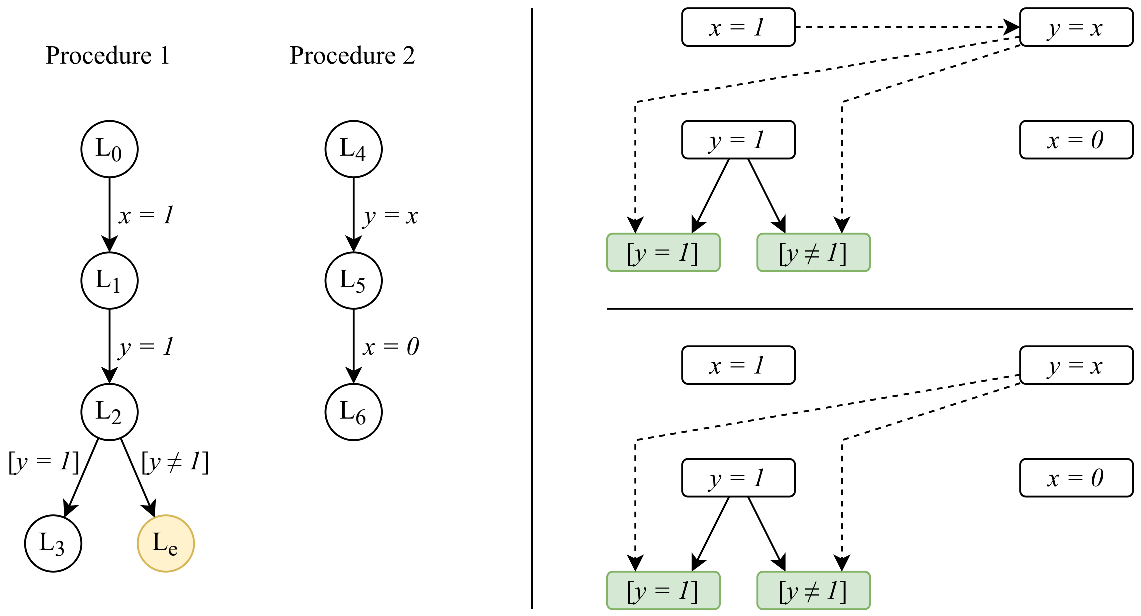


Figure 3.1: CFA of two procedures and data-flow graphs with $vars(\Pi) = \{x, y\}$ (above), and $vars(\Pi) = \{y\}$ (below)

Definition 5 (Abstract Data-Flow Graph). 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 procedure of their process, and
- $I \subseteq A \times A$ is the set of indirect observation edges: $(\alpha, \beta) \in I$ if β observes α with Π and $p_\alpha \neq p_\beta$.[1] ▪

The data-flow graph can be precomputed for the abstract state space exploration. For collecting 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, the algorithm simply iterates over the actions of all other procedures and adds an indirect observation edge wherever necessary. So, the data-flow graph can be built in polynomial (quadratic) time in the number of CFA edges.

Example 3. *Let us have two processes with the CFA procedures from Figure 3.1. The figure shows two abstract data-flow graphs: one with a precision where some information is tracked about both x and y ($vars(\Pi) = \{x, y\}$) and another where 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, while dashed edges represent inter-process observations.*

[1] On the implementation side, where processes can be created and terminated dynamically, several processes can have the same CFA procedure. In that case, inter-process observation edges can exist between actions of the same CFA procedure.

3.2 Statement Simplification

This section describes how the abstract data-flow graph can be used to simplify or completely eliminate statements. After presenting the algorithm in Section 3.2.1, I prove its correctness in Section 3.2.2.

3.2.1 Using the Data-Flow Graph to Simplify Statements

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, $\alpha \in \mathit{enabled}(s)$: our goal is to decide whether α can be transitively observed later during the program execution in a relevant way. In this work, I target reachability properties, so relevant actions are the actions with guard conditions since the reachability of error locations of the CFA can only be blocked by conditional statements. I will refer to these relevant actions as *real observers*. Real observers are colored in Figure 3.1. Thus, the evaluation of the action α can be skipped if there is no trace from the current state where a *real observer* transitively observes α. Based on the reflexivity of the transitive observation relation, conditional statements are never simplified. Whether an action is transitively observed by a real observer can be decided using the data-flow graph. For this, I introduce the following definition:

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

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 a process. In fact, formally, we have conditions for the enabledness of the data-flow graph edges (different conditions for direct and indirect observation edges):

Definition 7 (Enabled Edges of an Abstract Data-Flow Graph). Let s be an abstract state, and $G = (A, D, I, \Pi)$ an abstract data-flow graph.

- An edge $(a_1, a_2) \in D$ is enabled in s if $a_1, a_2 \in \mathit{reachable}(s, p)$ for some process p.
- An edge $(a_1, a_2) \in I$ is enabled in s if $a_1 \in \mathit{reachable}(s, p_1)$ and $a_2 \in \mathit{reachable}(s, p_2)$ for some processes $p_1 \neq p_2$. ▪

Rephrasing my statement from the previous paragraph: if there is a trace from s where α is transitively observed by an action β, then there is a sequence of actions $a_1, ..., a_n$ such that $a_1 = \alpha$, $a_n = \beta$, $(a_i, a_{i+1}) \in D \cup I$ for each $1 \leq i < n$, and (a_i, a_{i+1}) is enabled in s. That is, this sequence of actions $a_1, ..., a_n$ represents the data-flow between α and β.

Therefore, the data-flow graph G is traversed from s for each action $\alpha \in \mathit{enabled}(s)$ to check if there is a path from α to a real observer. During the traversal, we only use enabled edges of the data-flow graph. All direct observation edges reachable in G from an action $\alpha \in \mathit{enabled}(s)$ are enabled based on Definition 7. However, deciding whether an inter-process observation edge is enabled is not a trivial task. Fortunately, using an adequate data structure, it can be decided in constant time during the verification. Section 3.4 presents the details of an efficient method for deciding the enabledness of data-flow graph edges.

If a real observer is reached from α in the data-flow graph, 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 of α is unused, so it is unnecessary to evaluate α. Instead, the successor state s' can be the state that only differs from the current state s in the location of the process of α: $s'(p_\alpha)$ is the target location of α.

In fact, the following method is used to determine the successor states. Let v be the single variable $v \in \mathit{written}(\alpha)$. Note that $\mathit{written}(\alpha)$ has exactly one item when α is not transitively observed by a real observer because α must be an assignment then (conditional statements are real observers, hence, they are transitively observed).

- 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 simply removed (more exactly replaced with a *no operation* statement that has no effect).

Using the *havoc* statement on the variables tracked in the current abstraction is necessary for the refinement step of CEGAR (see more details in Section 3.3).

Example 4. *Let us take the example of two processes from Figure 3.1 again with a precision such that $\mathit{vars}(\Pi) = \{x, y\}$. Let us have a state s where the processes are in locations L_0 and L_5. Since the statement $y = x$ cannot be reached by any process from s, the inter-process observation edge $(x = 1, y = x)$ of the data-flow graph is disabled (as well as some other edges of the data-flow graph): Figure 3.2 shows the data-flow graph with disabled edges. Thus, there is no path from the action $x = 1$ to a real observer, so the evaluation of this statement can be skipped in s. Since the written variable x is in the precision, the assignment will be replaced by a* havoc *statement, in fact.*

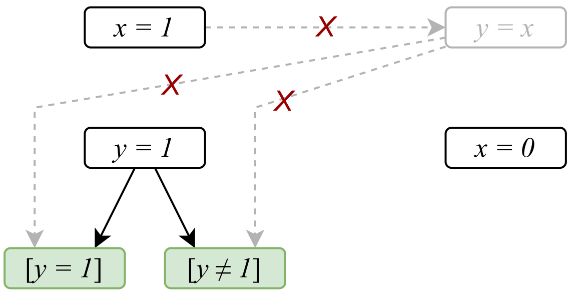


Figure 3.2: Abstract data-flow graph with disabled edges

Algorithm 1 summarizes the presented method of statement simplification based on dynamic data-flow analysis. The initial state and the precision of the current abstraction are the inputs of the algorithm. The output is a verdict: unsafe if an error state is reachable, safe otherwise. The algorithm is a modified state space exploration. The *waitlist* variable is used as an abstraction of the state space traversal strategy (e.g., BFS, DFS, or some heuristic A* search); that is, the search strategy is independent of my algorithm, and it can be customized. The body of the *foreach* loop formulates how the statement simplification presented in this section works.

Algorithm 1: State Space Exploration with Statement Simplification

```
Input: $s_0, \Pi$                          /* $s_0$: initial state, $\Pi$: 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
6          return unsafe
7      end
8      foreach $\alpha \in enabled(s)$ do
9          if $\exists$ path in $G$ of enabled edges in $s$ from $\alpha$ to a real observer then
10             $successors \leftarrow \alpha(s)$
11         else
12             if $written(\alpha) = \{v\}$ and $v \in vars(\Pi)$ then
13                 $\alpha' \leftarrow havoc\ v$
14                 $successors \leftarrow \alpha'(s)$
15             else
16                 $s' \leftarrow s$
17                 $s'(p_\alpha) \leftarrow$ target location of $\alpha$
18                 $successors \leftarrow \{s'\}$
19             end
20         end
21         $waitlist \leftarrow waitlist \cup successors$
22     end
23 end
24 return safe
```

3.2.2 Correctness of the Presented Algorithm

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 the original state space, I mean the abstract state space explored without the introduced statement simplification (i.e., for each $\alpha \in enabled(s)$, the successor states $\alpha(s)$ are added to the state space).

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. I 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 a feasible abstract error trace from at least one $s' \in successors$, as well. 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 $v \in written(\alpha)$.[2]

   In this case, α practically has no effect since no information is tracked about v in the current abstraction. So, an assignment of v does not modify anything in the state except for the location update for p_α. This is exactly how s' is defined in lines 16-17, 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 $successors = \alpha(s)$.

3. α is not observed transitively by a real observer, and $v \in vars(\Pi)$.

   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. However, in the latter case, we are back in the previous case since, practically, α is not replaced (a *havoc* statement replaced with a *havoc* statement 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* statement 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 $s' \in successors = \alpha'(s)$.

As the property proven above is preserved in each step when an action is explored from an abstract state, 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. $\square$

3.3 Statement Simplification in CEGAR

My dynamic data-flow analysis-based state space exploration algorithm presented in the previous section could be used in other formal verification algorithms than CEGAR, as well. In such a case, it may be possible to forget lines 12-15 of Algorithm 1 and simply use lines 16-18 to define the successor state when the action α is not observed transitively by a real observer. However, I focus on CEGAR as the base verification algorithm in this work.

CEGAR is an iterative algorithm, and it refines the abstraction when a spurious counterexample is found during the exploration of the abstract state space [20]. The counterexample provided by the abstractor is a trace: an alternating sequence of abstract states and (original, not simplified) 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.

It is important that the counterexample provided to the refiner must contain the original actions, even if they were simplified by my algorithm during the state space exploration. Consider a program with a single process whose CFA is the one from Figure 3.3 where L_e is an error location. Let us have a precision where we only track information about x:

[2]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.

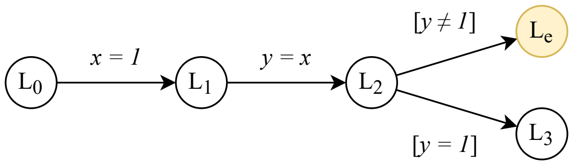


Figure 3.3: CFA of a single process

$vars(\Pi) = \{x\}$. An abstract error state is reachable in the abstract state space with the trace ($x = 1$, $y = x$, $[y \neq 1]$), which is clearly spurious (the value of y must be 1 after the first two statements, even if the current abstraction does not realize it). However, my algorithm simplifies $x = 1$ as well as $y = x$ 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*, $[y \neq 1]$), then the contradiction cannot be spotted. The refiner would conclude that the counterexample is feasible. Thus, a wrong *unsafe* verdict would be given as the result of the verification.

It is also necessary to use the *havoc* statement when the assignment of a variable in the precision is simplified. If the successor state is simply defined as in lines 16-17 instead of using a *havoc* statement, then the refiner may see a contradiction. As an example, let us assume that the value of variable x is explicitly tracked in an iteration of CEGAR, and the abstractor has found a counterexample trace. The trace contains a state s where the value of x is 0, and the next action α assigns 1 to x. However, my algorithm noticed that α cannot be transitively observed by any real observer, so it skipped the evaluation of the statement of α for calculating the successor states $\alpha(s)$. Based on lines 16-17 of the algorithm, the value of x would be the same (namely 0) in the state s' after α in the trace (we have seen in the previous paragraph that the counterexample must contain the original actions). The refiner finds a contradiction here as the value of x cannot be 0 after an action that assigns 1 to x. On the other hand, the precision could not be refined based on this misleading contradiction, and the CEGAR algorithm would probably get stuck in endless iterations.

This problem is overcome by applying a *havoc* statement instead of the unevaluated assignment expression since the *havoc* statement covers the behavior of the original assignment, whatever value it would assign. Going back to the previous example, the *havoc* statement would erase the value of x from s', so it is not a contradicting state after α. Furthermore, the evaluation of the *havoc* statement is still a simple task, so it is still worth replacing the original assignments with a *havoc* statement.

It is also worth mentioning that my algorithm cannot introduce new spurious counterexamples in certain abstract domains and degrade the performance of the verification by increasing the number of CEGAR iterations due to new spurious counterexamples. For example, in explicit-value abstraction, intuitively, guard conditions cannot get enabled as a result of applying my algorithm since Algorithm 1 only simplifies statements that are not observed transitively by any real observer, i.e., by any conditional statement. Thus, the evaluation of guard conditions is not affected, so completely new (spurious) counterexamples cannot emerge as a result of my algorithm. As for originally feasible counterexamples, they remain feasible with my algorithm as well since feasible traces are always available in the abstract state space explored by my algorithm based on the proof of Theorem 1. On the other hand, new spurious counterexamples may emerge in certain abstract domains, such as the predicate abstract domain where there can be a predicate about a variable of a simplified statement and another variable appearing in a guard condition: then, my previous reasoning does not work.

3.4 Deciding Enabledness of Data-Flow Graph Edges

When the abstract data-flow graph is traversed from a state, only enabled data-flow graph edges can be used. Using an adequate data structure, the enabledness of an edge in the data-flow graph can be decided (or at least over-approximated) in constant time during the verification. Finding the right data structure and algorithm is not trivial, though.

3.4.1 Problem Statement

To formulate the traversal of the data-flow graph, let us have a state s. Based on Definition 7 of enabled data-flow graph edges, we need the set $\mathit{reachable}(s, p)$ to decide the enabledness of a data-flow graph edge. First, note that $\mathit{reachable}(s, p)$ cannot be determined without exploring the state space from s. However, we can easily compute an over-approximation of it based on the CFA procedure of p: the actions reachable in the CFA of p from the location $s(p)$. Naturally, this way, we may get redundant actions, since an action with a guard condition may not be enabled in the state space, and this fact is ignored if we simply over-approximate $\mathit{reachable}(s, p)$ based on the control-flow graph. In this section, I use $\mathit{may\_reachable}(s, p)$ to refer to this over-approximated form of the $\mathit{reachable}(s, p)$ set; similarly, I say that a data-flow graph edge is *may_enabled* if we use *may_reachable* in Definition 7 instead of *reachable*. The over-approximation means that:

- $\alpha \in \mathit{reachable}(s, p)$ implies $\alpha \in \mathit{may\_reachable}(s, p)$, and
- if the data-flow graph edge (a_1, a_2) is *enabled* in a state s, then (a_1, a_2) is *may_enabled* in s.

Let us have a path $a_1, ..., a_n$ in the abstract data-flow graph G that we have already explored during the traversal of G from s. For this path, $a_1 \in \mathit{enabled}(s)$ and the data-flow graph edge (a_i, a_{i+1}) is *may_enabled* in s for each $1 \leq i < n$. Note that since we only use *may_enabled* edges for the traversal of the data-flow graph, these edges must be *may_enabled*, indeed. We have to check whether a potential new edge (a_n, a_{n+1}) of the data-flow graph from the final node a_n of the path is *may_enabled* or not.

Now, if (a_n, a_{n+1}) is a direct observation edge, it is always *may_enabled*. Since (a_{n-1}, a_n) is *may_enabled*, $a_n \in \mathit{may\_reachable}(s, p)$ for some process p (based on Definition 7 using the over-approximations). Then, (a_n, a_{n+1}) being a direct observation edge implies that a_{n+1} is reachable from a_n in the CFA procedure of process p based on Definition 5. Therefore, $a_{n+1} \in \mathit{may\_reachable}(s, p)$ as well, so (a_n, a_{n+1}) is *may_enabled*, indeed.

The case of indirect observation edges is more complex. To decide their *may_enabled*ness, we have to check whether $a_{n+1} \in \mathit{may\_reachable}(s, p)$ for any process $p \neq p_{a_n}$. A trivial solution would be to traverse the CFA from $s(p)$ for each process p and collect the set $\mathit{may\_reachable}(s, p)$. However, this would mean that we have to traverse the CFA from each state during the state space exploration which would be a huge overhead that we certainly do not want to have.

A better approach is to cache the set of actions reachable from each CFA location: thus we have to traverse the CFA only once from each location. On the other hand, deciding whether $a_{n+1} \in \mathit{may\_reachable}(s, p)$ would still require checking the existence of an item in a set. On the implementation side, this means that either we iterate over the elements of the collection or we use a hashed set[3] to "quickly" get based on the hash of the item

[3]`https://docs.oracle.com/javase/8/docs/api/java/util/HashSet.html`

whether it is in the set. In the first case, nothing is gained: the iteration is linear in the number of CFA edges. In the latter case, seemingly, we can get the answer to our question in constant time. However, hash calculation for classes with complex data structures also has a computational overhead: if we calculate hashes for each data-flow graph node in each state of the abstract state space, we end up with an overhead that is not negligible.

Therefore, I have chosen another approach based on strongly connected components of the control-flow graph. The strongly connected components are calculated for the CFA procedure of each process once prior to the model checking using a linear time algorithm such as Tarjan's algorithm [36]. Then, as we will see in the next section, we simply have to compare integers to decide the *may_enabled*ness of a data-flow graph edge.

3.4.2 Tarjan's Algorithm

Tarjan's algorithm partitions the vertices of a directed graph in a way that for each pair of vertices u and v in the same partition, there is a directed path from u to v and from v to u. These partitions are called *strongly connected components* (SCC). Each SCC contains at least one directed cycle covering all of its vertices. Each vertex belongs to exactly one SCC: Tarjan's algorithm assigns a number $scc(v)$ to each vertex v, which is a unique identifier of the SCC of v. If a vertex v does not belong to any directed cycle of the graph, then v forms a SCC alone.

Tarjan's algorithm is an extended DFS search that produces a DFS spanning tree (or forest) [36]. Each SCC is a subtree of the DFS tree. The algorithm aims to find the *head* of each SCC, that is, the vertex from which all other vertices of the SCC are reachable in the DFS tree. The basic idea of the algorithm is the following: if we find a *back edge* during the DFS search (i.e., an edge that points to a visited but unfinished vertex), then it forms a cycle with the edges of the DFS tree. This cycle is either a cycle on all vertices of an SCC, or at least all vertices of the cycle are part of the same SCC. The head of the SCC can only be the target of the found back edge or one of its ancestors.

In this report, I do not aim to give a detailed description of Tarjan's algorithm, as it can be found on many websites. However, I briefly explain how the algorithm works. The algorithm defines the attributes $discover(v)$ and $lowest(v)$ for each vertex v. Tarjan's algorithm works as follows: any unvisited vertex v is selected, and the extended DFS is run from v; this is repeated until there are no unvisited vertices.

The extended DFS performs the following steps on a vertex v: it initializes $discover(v)$ to a new value (based on an incremental counter) and sets $lowest(v)$ to the same value, and puts v on a stack. Then, it iterates over the neighbors of v, and invokes DFS recursively for each unvisited neighbor u. After the recursive call has returned, it updates $lowest(v)$ to $lowest(u)$ if $lowest(u) < lowest(v)$. For each visited but unfinished neighbor u, we have a back edge, and we update $lowest(v)$ to $discover(u)$ if $discover(u) < lowest(v)$ (that is, u may be the head of the SCC of v). After all neighbors of v have been processed, if $lowest(v) < discover(v)$, then v is not the head of its SCC, so the DFS simply returns (but keeps v on the stack). However, if $lowest(v) = discover(v)$, then v is the head of its SCC: the items currently on the stack are the vertices of this SCC. The found SCC gets an incremental ID number, and it is assigned to each vertex on the stack. Then, the stack is emptied, and the DFS returns from processing v.

Tarjan's algorithm finds the strongly connected components in a *reversed* topological order of the condensation[4] of the original graph. So, the associated incremental IDs of SCCs are in a reversed order compared to the direction of edges: if $scc(u) < scc(v)$ for two vertices u and v, then v cannot be reached from u.

Getting back to control flow automata, Tarjan's algorithm is executed for each procedure of a multi-threaded CFA, which assigns a value $scc(l)$ to each location l of the procedure. If $scc(l_1) < scc(l_2)$ for locations l_1 and l_2 of the same procedure, then l_2 cannot be reached from l_1; otherwise, l_2 is considered as being reachable from l_1. Naturally, this is an over-approximation of location reachability: locations of parallel branches (without a loop) get different *scc* numbers, and none can be reached from the other.

Example 5. *Figure 3.4 shows the scc numbers associated to each graph vertices (CFA locations). CFA locations with the same number belong to the same SCC. That is, any location of an SCC can be reached from all other locations of the same SCC. The head of each SCC is the vertex of the SCC that is closest to the initial CFA location. It is clearly visible that if we have a path from l_1 to l_2, then $scc(l_1) \geq scc(l_2)$. Note that we have an over-approximation indeed, as the location whose number is 4 cannot be reached from the location with 5.*

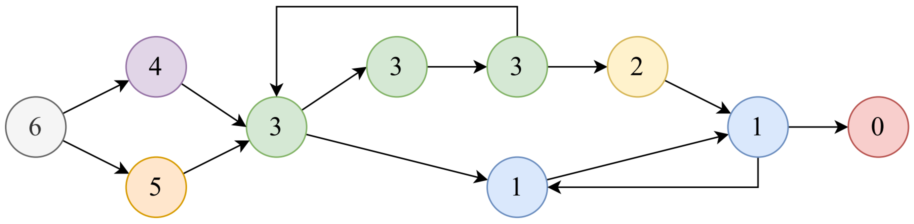


Figure 3.4: Strongly connected components of a CFA procedure

This way, when the data-flow graph G is traversed from a state s, we can easily check whether the edges of the data-flow graph are *may_enabled* or not. An action β may be reached in the future if the source location of β can be reached from the current location of p_β in s. This can now be easily decided by comparing the *scc* values assigned to the CFA locations. Formally, a data-flow graph edge $(a_1, a_2) \in I$ is *may_enabled* in s if $scc(s(p_{a_2})) \geq scc(l_2)$ where l_2 is the source location of a_2; that is, if l_2 is reachable from $s(p_{a_2})$ in the CFA.

For performance reasons and to avoid stack overflow in the case of large models, I have implemented an iterative form of Tarjan's algorithm instead of using the recursive DFS presented in this section. On the other hand, it would be more complicated to present the iterative form of the algorithm in an easily understandable way, so I opted for the presentation of the recursive form.

As we have seen, using the concept of strongly connected components, it is possible to decide the *may_enabled*ness of a data-flow graph edge by simply comparing integers instead of often traversing the CFA or computing complicated hashes.

[4]The condensation of a directed graph is a directed acyclic graph where each strongly connected component of the original graph is represented by a single vertex. If there is a directed path from one SCC of the original graph to another, then the condensation contains a directed edge between the respective vertices.

3.5 Effect on Partial Order Reduction

For the practical applicability of model checking for concurrent software, it is inevitable to use some reduction techniques to reduce the number of explored thread interleavings. One of the most common reduction techniques is partial order reduction (POR) introduced in Section 2.5. Therefore, when developing an algorithm for the verification of concurrent systems, as I do in this work, it is worth considering the interaction of the developed algorithm with partial order reduction.

Partial order reduction algorithms identify equivalent thread interleavings based on the interaction of threads: two actions are dependent if they belong to the same process, or they access the same shared variable (see Section 2.5 for more details). Since dependency plays a key role in partial order reduction algorithms, its performance heavily depends on the size of the dependency relation of actions. An important side effect of my statement simplifying algorithm presented in this paper is that it reduces dependency between the actions of the program. By simplifying or completely eliminating statements, variable accesses are also reduced, so the size of the dependency relation of actions is decreased by my method. Therefore, it is motivated to examine how my novel algorithm affects the performance of POR algorithms. In this work, I investigate a *static* POR algorithm, where the explored actions from a state are precomputed when the state is reached during the exploration; for dynamic POR algorithms, there is already a method in the literature to reduce dependency based on an observation relation between actions [2].

By default, I reduce the abstract state space first with POR, and then I apply my statement simplification algorithm. This way, POR is not affected by my algorithm as POR is performed first.

To observe the effect of my algorithm on POR, we could compose the two algorithms in the reverse order, i.e., first simplifying statements with my novel approach, then applying POR. This way, my algorithm eliminates some variable accesses from the statements which reduces the size of the dependency relation for POR. However, my algorithm is applied on a larger state space, which means that the abstract data-flow graph must be traversed more often, imposing a more significant computational overhead on the verification. To have our cake and eat it too, I propose the composition of the algorithms in a way that first POR is applied, then my algorithm is used to simplify statements, and then POR is applied again. This way, my algorithm is performed on the reduced state space, and POR is also applied with the smaller dependency relation. Naturally, this way, POR has to be applied two times, which also means a computational overhead compared to the first way of composition. However, it is reapplied on a smaller state space that has already been reduced by POR. Furthermore, this way, we can analyze the pure effect of my algorithm on POR, that is, what further reduction can be achieved when using my algorithm.

The correctness of these different compositions is not trivial (that we never reduce the state space in a way where we remove all reachable error states), though it is not too complicated either. However, I will not get into the formal details in this report. As for applying my algorithm after POR, based on the proof of Theorem 1, for each abstract error trace present in the POR-reduced abstract state space, my algorithm will also explore an abstract error trace, so correctness is preserved. When performing POR after my algorithm, intuitively, we only reduce parts of the state space that are irrelevant with respect to the verified property anyway (further POR reduction can happen when my algorithm simplifies statements, which means that such statements are irrelevant).

In the evaluation chapter (Chapter 4), I evaluate the extra reducing effect of a static partial order reduction applied after my statement simplification algorithm.

```
int x, y;

void thread1() {  // process p1
  x = 1;
  y = 1;
  if (y != 1) {
    reach_error();
  }
}

void thread2() {  // process p2
  y = x;
  x = 0;
}
```

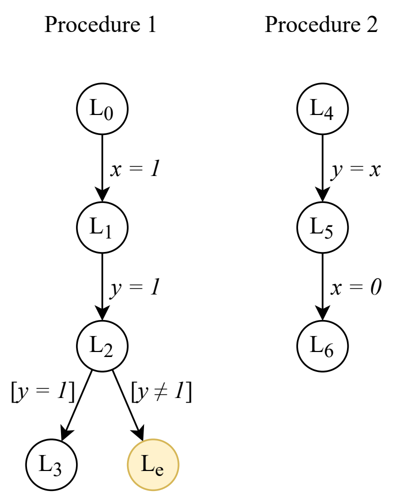


Figure 3.5: Example program for the case study

3.6 Case Study

In this section, the presented algorithm is illustrated on a small multi-threaded program. Let us follow the verification of our previous example step-by-step. Figure 3.5 shows the program with two concurrent threads and the CFA of the program with the two CFA procedures for the two threads, respectively. We have to check whether the error location L_e can be reached by process p_1.

Let us use explicit-value abstraction in CEGAR and let the precision be $\Pi = \{y\}$ in the first iteration, so only the value of variable y is tracked. Figure 3.7 depicts the abstract state space of the first iteration. Rectangles are states, and arrows are transitions. The locations of the active processes are shown in a state along with the value of the tracked variables (which is only y in this iteration). The labels of transitions indicate the simplified action, i.e., the action used for successor state calculation. Where the original statement is replaced by another statement (a *havoc* statement or an empty action denoted by `NOP`), the transition is highlighted with green; an action that is not enabled in a state only because its guard condition is false in the state is displayed with grey, and an "X" indicates that it is not enabled. For visual consistency, transitions going downwards belong to process p_1, while transitions going right belong to process p_2. To be concise, I will use $s_{i,j}$ as a reference to the state where the locations are L_i and L_j for processes p_1 and p_2, respectively. Unknown values are represented by the $\top$ (top) symbol.

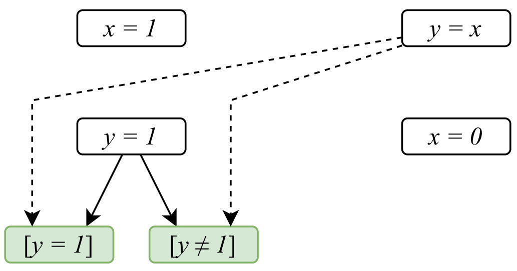


Figure 3.6: Abstract data-flow graph with $vars(\Pi) = \{y\}$

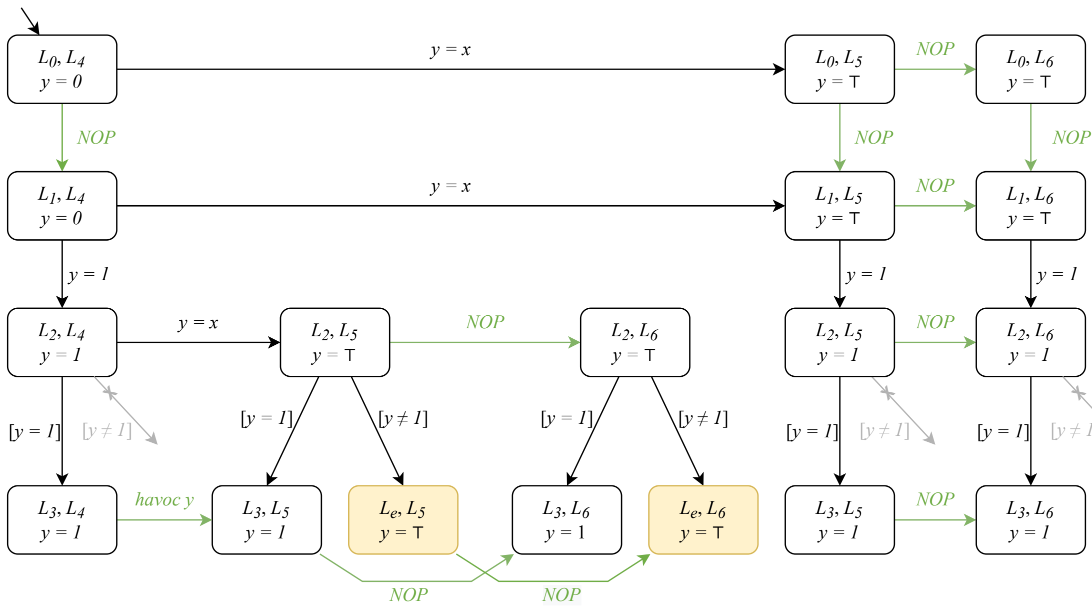


Figure 3.7: State space of the program

Based on our precision, the abstract data-flow graph used in this iteration is the one visible in Figure 3.6, that is, actions assigning x are not observed by any other action. The initial state is $s_{0,4}$ as all processes are in their initial locations, and the value of the tracked variable y is the default 0. There are two enabled actions in $s_{0,4}$: $x = 1$ and $y = x$. We apply the introduced statement simplification method for each enabled action. There are no observation edges starting from the action $x = 1$, and it is not a conditional statement, so it can be simplified: since x is not in the precision, this statement is completely reduced (replaced by an empty action). On the other hand, there is a path of enabled edges from the action $y = x$ in the data flow graph to a conditional statement, so it is transitively observed by a relevant action: it cannot be simplified. The successor state of $s_{0,4}$ with respect to $y = x$ is $s_{0,5}$, where the value of y is unknown ($\top$) since an unknown value (x) is assigned to y.

Proceeding further with the state space exploration, we can always eliminate the assignments of x since $x \notin vars(\Pi)$; see the NOP labels in Figure 3.7. In most cases, statements using y cannot be simplified. However, from $s_{3,4}$, the assignment of y can also be simplified. In $s_{3,4}$, the abstract data-flow graph has no enabled observation edges based on the locations of processes; thus, $y = x$ is not observed transitively by any conditional statement (in fact, no conditional action can be reached from $s_{3,4}$), so it can be simplified. Since $y \in vars(\Pi)$, we cannot completely eliminate this statement: we have to use the *havoc* statement on y.

Using this abstraction, there is an abstract error trace from the abstract initial state since $[y \neq 1]$ is enabled in states $s_{2,5}$ and $s_{2,6}$ due to the unknown value of y in these states. Let us assume that we have found the following abstract counterexample: $s_{0,4} \xrightarrow{x=1} s_{1,4} \xrightarrow{y=1} s_{2,4} \xrightarrow{y=x} s_{2,5} \xrightarrow{[y\neq 1]} s_{e,5}$. Naturally, this is a spurious counterexample since the value of y is always 1 after the first three actions, so $[y \neq 1]$ cannot be executed after them. Note that if we had used the simplified statements (i.e., NOP instead of $x = 1$) for checking the feasibility of the counterexample, we could not have spotted the contradiction (since the initial value of x is 0 which is assigned to y, and $0 \neq 1$).

Also, note that my algorithm may be able to simplify a statement in one interleaving of threads even if it cannot simplify that statement in another interleaving: see for example the statement $y = x$ from $s_{3,4}$ versus from $s_{0,4}$. Existing cone-of-influence algorithms would say that the variables x and y are both important if we want to prove the safety of this program, so those algorithms would not be able to eliminate any statement from this program.

In the next iteration of CEGAR, we can use a precision where both variables are explicitly tracked. This way, we are able to prove that the program is safe since p_1 can never reach its error location L_e. In this iteration as well, my algorithm could simplify some statements though less than in the previous iteration: e.g., $x = 1$ is observed from the initial state, so it cannot be simplified.

This case study demonstrates that my proposed algorithm can simplify a significant proportion of program statements. Evaluation results in Chapter 4 confirm that my algorithm is capable of a considerable reduction over a large set of benchmark programs as well.

Chapter 4

Experiments and Evaluation

In this section, I evaluate the efficiency of my algorithmic contributions. First, I introduce my plans for the experiment with the research questions in Section 4.1. Then, I present and evaluate the results in Section 4.2. Finally, I discuss the threats to the validity of my experiments in Section 4.3.

4.1 Experiment Design

The goal of my experiment is to evaluate the performance of my novel dynamic data-flow-based statement simplification algorithm presented in the previous section. Though I have explained the differences between my method and cone-of-influence techniques in Chapter 1, I still refer to my algorithm as cone-of-influence-based statement simplification algorithm (or `COI` for short) in this evaluation section.

The presented algorithm has been defined independently of concrete abstract domains, therefore I investigate the effect of my proposed algorithm in two frequently used abstract domains: explicit-value abstraction (later `EXPL`) [11] and (Cartesian) predicate abstraction (later `PRED`) [5].

I also investigate the effect of my proposed algorithm on partial order reduction as described in Section 3.5. The default implementation (where my algorithm is applied after POR and POR is not affected by my algorithm) is referred simply as `COI`. The configuration where POR is reapplied after my algorithm is `POR-COI`.

I have implemented my algorithm as an open-source extension of the Theta verification framework [37], which already had a built-in CEGAR algorithm and had prior support for multi-threaded C programs, including a partial order reduction algorithm (partial order reduction was my work at last year's TDK conference). I will refer to the existing Theta algorithm as `NO COI`, that is, the baseline configuration where my new algorithm is not enabled.

4.1.1 Research Questions

To evaluate the presented algorithm, I aim to answer the following research questions concerning important metrics with respect to my novel algorithm and the above-listed configurations.

RQ1 What is the proportion of statements that can be simplified or completely reduced using the presented algorithm?

RQ2 How is the time of successor state calculation affected by the introduced algorithm?

RQ3 How is the overall performance of the verification affected by my algorithm?

RQ4 How does the proposed algorithm affect the performance of partial order reduction?

4.1.2 Experimental Configuration

In my experiments, I executed different configurations of THETA over a set of input programs written in C from the concurrency safety benchmark suite[1] of SV-COMP [9] that is parsable by THETA. I executed 4 configurations: both abstract domains (`EXPL`, `PRED`) with my algorithm enabled and disabled. The benchmark tests were executed on virtual machines with Intel Core (Haswell) processors; 5 dedicated CPU cores were allocated to each task. Each verification task had a time limit of 900 seconds and a memory limit of 15GB. I used a sequence interpolation-based refinement strategy for the refinement step of CEGAR and a depth-first state space exploration strategy with thread-safe large-block encoding [28] in the abstraction phase. I used atoms as the basis of predicate splitting for the predicate domain; I used a maximum number of enumerated successor states (maxenum) of 1 for the explicit domain [28].

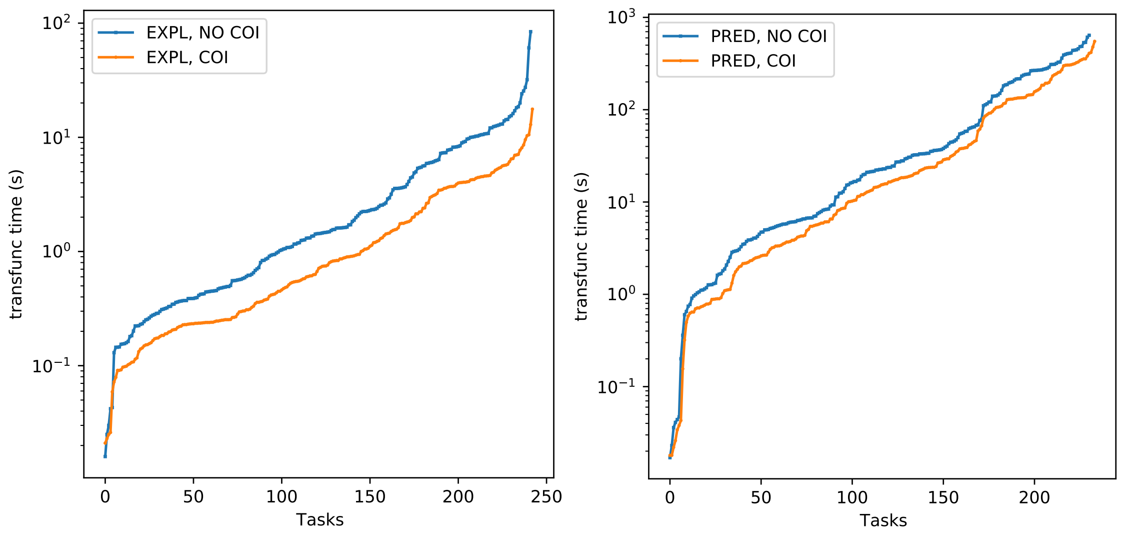
4.2 Experiment Results

In the concurrency safety benchmark suite, THETA was able to parse 266 programs. No configuration provided any wrong results. Table 4.1 shows the results of 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: out of the 266 tasks, a common subset of 242 tasks was solved with the configurations using explicit-value abstraction, and 224 with predicate abstraction. The *simplified ratio* column stands for the average proportion of simplified statements (including completely eliminated statements) compared to all statements. The *successor state calculation* and *CPU time* columns are the sum of successor state calculation times and CPU times of the commonly solved tasks, respectively.

domain	coi	simplified ratio	successor state calculation (s)	CPU time (s)	solved tasks
EXPL	NO COI	0	1057	4770	242
	COI	0.218	435	4095	243
	POR-COI	0.221	660	4734	243
PRED	NO COI	0	17338	27506	231
	COI	0.224	12802	25192	234
	POR-COI	0.227	11388	23497	234

Table 4.1: Different metrics for the evaluation of benchmark results

[1]`https://gitlab.com/sosy-lab/benchmarking/sv-benchmarks/-/tree/2fa025c8cb683e5991b2bbdb057e4cb328700dc0`



(a) Explicit-value abstraction **(b)** Predicate abstraction

Figure 4.1: Successor state calculation on quantile plots

The results visibly confirm the reduction potential of my proposed algorithm: COI configurations greatly outperform NO COI configurations in terms of both successor state calculation and overall verification performance. Let us have the answers to the research questions:

RQ1 The proportion of simplified statements is more than 20% of all statements on average, see Table 4.1. In more detail, 17.0% of all statements are completely eliminated, while 4.8% of all statements are replaced with *havoc* statements with explicit-value abstraction; 17.3% and 5.1% for the same metrics when predicate abstraction is used. This confirms the relevance of my algorithm, i.e., a significant subset of statements is unnecessary in the sense that their result is unused in certain thread interleavings for the verification of the given property.

RQ2 The time of successor state calculation is greatly reduced by my algorithm. It is reduced by 58.9% with explicit abstraction and 26.2% with predicate abstraction. Figure 4.1 shows the successor state calculation time on quantile plots for the two abstract domains: the horizontal axes represent the tasks sorted by successor state calculation time, while the vertical axes show the overall successor state calculation time used to verify the corresponding problem. 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.

RQ3 The overall verification performance is also considerably improved by the proposed algorithm: `COI` reduces the overall CPU time by 14.2% using explicit-value abstraction and by 8.4% using predicate abstraction. However, the number of solved tasks is only slightly increased, probably because the complexity of input tasks is not linearly increasing. The computational overhead of my algorithm is not huge, though not completely negligible: 164 seconds and 241 seconds aggregated for the tasks of explicit and predicate abstraction, respectively, which is a mere 4.0% and 1.0% of all CPU time.

RQ4 Interestingly, applying POR after my algorithm behaves differently in different abstract domains. Though the proportion of simplified statements is slightly increased compared to the `COI` configurations, successor state calculation and overall CPU time have been negatively affected with the explicit domain while further improved with predicate abstraction (see Table 4.1). The following factors may lead to a decrease in performance of the `POR-COI` configuration:

- the POR reduction has to be executed twice for each state (though on a smaller input for the second time);
- the implemented POR algorithm aims to explore the smallest sufficient subset of enabled actions which may often be a simplified statement (as they are independent with most of the other actions). However, from another perspective, this means that the POR algorithm prefers the exploration of irrelevant model elements in some cases: this way, finding a counterexample is hindered. Indeed, for *safe* tasks in the benchmark set (i.e., where there is no feasible counterexample), `POR-COI` is able to achieve similar performance to the `COI` configuration (the overall CPU time of `COI` being only 2.8% better than `POR-COI`), whereas the performance of `POR-COI` on *unsafe* tasks suffers a considerable overhead (`COI` overall CPU time is 14.9% less in this case).

As a summary of my findings, my algorithm greatly improved the verification performance of concurrent software in all measured metrics. Applying partial order reduction again after the proposed simplification algorithm yields various results in different abstract domains: it can further improve the performance in certain cases.

4.3 Threats to Validity

The validity of my experiments may be influenced by the following factors.

Internal validity. Consistency and accuracy of the experiments were ensured by using the BenchExec framework [15]. I executed my experiments on virtual machines in the cloud computing platform of our university. Therefore, external factors such as loads on other virtual machines of the host environment and shared resources (such as disks) may have slightly influenced the results. On the other hand, to moderate the cloud performance interference, I performed the experiments in low utilization periods.

External validity. As my experiments were performed on benchmark programs of SV-COMP, results might not be generalizable to real-life industrial programs. However, the SV-COMP benchmark suite is considered to be a de facto standard for academic benchmarking of software verification tools. Furthermore, THETA can only parse a limited subset of SV-COMP concurrent benchmark programs, which further reduces generalizability. On the other hand, there are typically more redundant model elements in a real-world verification task than in the stripped, simplified programs of the SV-COMP benchmark suite that THETA is currently able to parse. Thus, my algorithm may achieve even greater reduction in industrial applications.

Construct validity. The metrics of the evaluation were carefully chosen to accurately describe the performance of my algorithm: both *end-user* statistics (such as overall CPU time or the number of solved tasks) and *backend-related* information (such as the proportion of simplified statements or successor state calculation time) were used. Therefore, these metrics accurately represent the expected outcomes of the executions.

4.4 Conclusion

In this report, I have presented a novel statement reduction algorithm based on dynamic data-flow analysis to aid abstract state space exploration of concurrent programs. My method is based on a similar idea to cone-of-influence algorithms. However, my algorithm performs a more fine-grained analysis, resulting in a more extensive reduction of model elements. I have proven its correctness and discussed its integration into the abstraction-based verification algorithm CEGAR as well as some low-level algorithmic optimizations, and the effects of my algorithm on partial order reduction.

With my work, I contributed to an open-source verification tool, Theta. My contributions enable Theta to verify a wider range of concurrent programs.

The evaluation of the algorithm shows that my 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. Therefore, the presented algorithm is worth implementing in a state-of-the-art model checking tool that verifies concurrent software.

4.5 Future Work

Benchmark tests have raised several questions that still have to be properly investigated. Most notably, why is the performance affected differently in different abstract domains? I plan to carry out a more profound benchmark test and analyze the results to get a deeper insight into the effects of my proposed algorithm. Based on what I may learn from a thorough analysis, I may be able to specialize my algorithm on some specific abstract domains to further improve the performance.

Though the proposed algorithm takes its advantage of considering the current interleaving of concurrent threads, theoretically, the algorithm can also be used to simplify the statements of a single-threaded program. I plan to perform benchmark tests on single-threaded programs as well, to see what improvements can be achieved by the presented algorithm on such tasks.

Software verification, especially the verification of concurrent software, remains a hard problem. I hope to find new solutions and optimize the algorithms presented in this work to make concurrent software verification feasible for safety-critical systems of larger scale.

Acknowledgements

Project no. 2019-1.3.1-KK-2019-00004 has been implemented with the partial support provided from the National Research, Development and Innovation Fund of Hungary, financed under the 2019-1.3.1-KK funding scheme.

Bibliography

[1] Parosh Aziz Abdulla, Stavros Aronis, Bengt Jonsson, and Konstantinos Sagonas. Comparing Source Sets and Persistent Sets for Partial Order Reduction. volume 10460 of *Lecture Notes in Computer Science*, pages 516–536. Springer, 2017. DOI: `10.1007/978-3-319-63121-9\_26`.

[2] Stavros Aronis, Bengt Jonsson, Magnus Lång, and Konstantinos Sagonas. Optimal Dynamic Partial Order Reduction with Observers. volume 10806 of *Lecture Notes in Computer Science*, pages 229–248. Springer, 2018. DOI: `10.1007/978-3-319-89963-3\_14`.

[3] Christel Baier and Joost-Pieter Katoen. *Principles of Model Checking.* MIT Press, 2008. ISBN 978-0-262-02649-9.

[4] Levente Bajczi, Zsófia Ádám, and Vince Molnár. C for Yourself: Comparison of Front-End Techniques for Formal Verification. IEEE, 2022. DOI: `10.1145/3524482.3527646`.

[5] Thomas Ball, Andreas Podelski, and Sriram K. Rajamani. Boolean and Cartesian Abstraction for Model Checking C Programs. volume 2031 of *Lecture Notes in Computer Science*, pages 268–283. Springer, 2001. DOI: `10.1007/3-540-45319-9\_19`.

[6] Haniel Barbosa, Clark W. Barrett, Martin Brain, Gereon Kremer, Hanna Lachnitt, Makai Mann, Abdalrhman Mohamed, Mudathir Mohamed, Aina Niemetz, Andres Nötzli, Alex Ozdemir, Mathias Preiner, Andrew Reynolds, Ying Sheng, Cesare Tinelli, and Yoni Zohar. cvc5: A versatile and industrial-strength SMT solver. volume 13243 of *Lecture Notes in Computer Science*, pages 415–442. Springer, 2022. DOI: `10.1007/978-3-030-99524-9\_24`.

[7] Clark W. Barrett and Cesare Tinelli. Satisfiability modulo theories. pages 305–343. Springer, 2018. DOI: `10.1007/978-3-319-10575-8\_11`.

[8] Sergey Berezin, Sérgio Vale Aguiar Campos, and Edmund M. Clarke. Compositional Reasoning in Model Checking. volume 1536 of *Lecture Notes in Computer Science*, pages 81–102. Springer, 1997. DOI: `10.1007/3-540-49213-5\_4`.

[9] Dirk Beyer. Competition on Software Verification and Witness Validation: SV-COMP 2023. volume 13994 of *Lecture Notes in Computer Science*, pages 495–522. Springer, 2023. DOI: `10.1007/978-3-031-30820-8\_29`.

[10] Dirk Beyer and M. Erkan Keremoglu. CPAchecker: A Tool for Configurable Software Verification. volume 6806 of *Lecture Notes in Computer Science*, pages 184–190. Springer, 2011. DOI: `10.1007/978-3-642-22110-1\_16`.

[11] Dirk Beyer and Stefan Löwe. Explicit-Value Analysis Based on CEGAR and Interpolation. *CoRR*, abs/1212.6542, 2012.

[12] Dirk Beyer, Thomas A. Henzinger, Ranjit Jhala, and Rupak Majumdar. The software model checker Blast. *Int. J. Softw. Tools Technol. Transf.*, 9(5-6):505–525, 2007. DOI: `10.1007/s10009-007-0044-z`.

[13] Dirk Beyer, Thomas A. Henzinger, and Grégory Théoduloz. Configurable Software Verification: Concretizing the Convergence of Model Checking and Program Analysis. volume 4590 of *Lecture Notes in Computer Science*, pages 504–518. Springer, 2007. DOI: `10.1007/978-3-540-73368-3\_51`.

[14] Dirk Beyer, Matthias Dangl, and Philipp Wendler. A Unifying View on SMT-Based Software Verification. *J. Autom. Reason.*, 60(3):299–335, 2018. DOI: `10.1007/s10817-017-9432-6`.

[15] Dirk Beyer, Stefan Löwe, and Philipp Wendler. Reliable benchmarking: requirements and solutions. *Int. J. Softw. Tools Technol. Transf.*, 21(1):1–29, 2019. DOI: `10.1007/s10009-017-0469-y`.

[16] Per Bjesse. What is Formal Verification? *SIGDA Newsl.*, 35(24):1–es, dec 2005. ISSN 0163-5743. DOI: `10.1145/1113792.1113794`.

[17] Nicolas Blanc and Daniel Kroening. Race analysis for systemc using model checking. *ACM Trans. Design Autom. Electr. Syst.*, 15(3):21:1–21:32, 2010. DOI: `10.1145/1754405.1754406`.

[18] Alessandro Cimatti, Alberto Griggio, Bastiaan Joost Schaafsma, and Roberto Sebastiani. The mathsat5 SMT solver. volume 7795 of *Lecture Notes in Computer Science*, pages 93–107. Springer, 2013. DOI: `10.1007/978-3-642-36742-7\_7`.

[19] Edmund M. Clarke, Orna Grumberg, and David E. Long. Model Checking and Abstraction. *ACM Trans. Program. Lang. Syst.*, 16(5):1512–1542, 1994. DOI: `10.1145/186025.186051`.

[20] Edmund M. Clarke, Orna Grumberg, Somesh Jha, Yuan Lu, and Helmut Veith. Counterexample-guided abstraction refinement for symbolic model checking. *J. ACM*, 50(5):752–794, 2003. DOI: `10.1145/876638.876643`.

[21] Edmund M. Clarke, William Klieber, Milos Novácek, and Paolo Zuliani. Model Checking and the State Explosion Problem. volume 7682 of *Lecture Notes in Computer Science*, pages 1–30. Springer, 2011. DOI: `10.1007/978-3-642-35746-6\_1`.

[22] Leonardo Mendonça de Moura and Nikolaj S. Bjørner. Z3: an efficient SMT solver. volume 4963 of *Lecture Notes in Computer Science*, pages 337–340. Springer, 2008. DOI: `10.1007/978-3-540-78800-3\_24`.

[23] Matthew B. Dwyer and Lori A. Clarke. Data Flow Analysis for Verifying Properties of Concurrent Programs. pages 62–75. ACM, 1994. DOI: `10.1145/193173.195295`.

[24] Cormac Flanagan and Shaz Qadeer. Predicate abstraction for software verification. pages 191–202. ACM, 2002. DOI: `10.1145/503272.503291`.

[25] Patrice Godefroid. *Partial-Order Methods for the Verification of Concurrent Systems - An Approach to the State-Explosion Problem*, volume 1032 of *Lecture Notes in Computer Science*. Springer, 1996. ISBN 3-540-60761-7. DOI: `10.1007/3-540-60761-7`.

[26] Orna Grumberg, Edmund M. Clarke, and Doron A. Peled. Model checking. 1999.

[27] Ákos Hajdu. *Effective Domain-Specific Formal Verification Techniques.* Thesis, Budapest University of Technology and Economics, 2020.

[28] Ákos Hajdu and Zoltán Micskei. Efficient Strategies for CEGAR-based Model Checking. *Journal of Automated Reasoning*, 64(6):1051–1091, 2020. DOI: `10.1007/s10817-019-09535-x`.

[29] Mark Harman and Robert M. Hierons. An overview of program slicing. *Softw. Focus*, 2(3):85–92, 2001. DOI: `10.1002/swf.41`.

[30] Thomas A. Henzinger, Ranjit Jhala, Rupak Majumdar, and Grégoire Sutre. Software Verification with BLAST. volume 2648 of *Lecture Notes in Computer Science*, pages 235–239. Springer, 2003. DOI: `10.1007/3-540-44829-2\_17`.

[31] ISO/IEC 9899:201x. Programming languages — C. International standard, International Organization for Standardization, International Electrotechnical Commission, December 2010.

[32] Java SE 8 Edition. The Java Language Specification. Language specification, Sun Microsystems, May 2015.

[33] Carmelo Loiacono, Marco Palena, Paolo Pasini, Denis Patti, Stefano Quer, Stefano Ricossa, Danilo Vendraminetto, and Jason Baumgartner. Fast cone-of-influence computation and estimation in problems with multiple properties. pages 803–806. EDA Consortium San Jose, CA, USA / ACM DL, 2013. DOI: `10.7873/DATE.2013.170`.

[34] Antoni W. Mazurkiewicz. Trace Theory. volume 255 of *Lecture Notes in Computer Science*, pages 279–324. Springer, 1986. DOI: `10.1007/3-540-17906-2\_30`.

[35] Doron A. Peled. Ten Years of Partial Order Reduction. volume 1427 of *Lecture Notes in Computer Science*, pages 17–28. Springer, 1998. DOI: `10.1007/BFb0028727`.

[36] Robert Endre Tarjan. Depth-First Search and Linear Graph Algorithms. *SIAM J. Comput.*, 1(2):146–160, 1972. DOI: `10.1137/0201010`.

[37] Tamás Tóth, Ákos Hajdu, András Vörös, Zoltán Micskei, and István Majzik. Theta: a Framework for Abstraction Refinement-Based Model Checking. pages 176–179, 2017. ISBN 978-0-9835678-7-5. DOI: `10.23919/FMCAD.2017.8102257`.

[38] Alan M. Turing. On computable numbers, with an application to the Entscheidungsproblem. *Proc. London Math. Soc.*, s2-42(1):230–265, 1937. DOI: `10.1112/plms/s2-42.1.230`.