MŰEGYETEM 1782

Budapest University of Technology and Economics
Faculty of Electrical Engineering and Informatics
Department of Artificial Intelligence and Systems Engineering

Efficient Automatic Verification of Concurrent Programs

Master's Thesis

Author
Csanád Telbisz

Advisor
Levente Bajczi

December 6, 2024

Contents

MŰEGYETEM 1782

DIPLOMATERVEZÉSI FELADAT

Telbisz Csanád Ferenc
szigorló mérnökinformatikus hallgató részére

Többszálú programok hatékony automatikus verifikációja

A kritikus beágyazott rendszerek világában mind a mai napig nehézséget jelent a többmagos processzorok hatékony kihasználása, főként a komplexitás biztonsági implikációi miatt. Hagyományos szoftververifikációs módszerek, mint például a tesztelés, nem tudják megfelelő biztonsággal kiértékelni a több szálon futó programok viselkedését.

Egy megoldást nyújthat erre a problémára a modellellenőrzés, mely egy formális megközelítéssel bizonyíthatja a programok biztonságát, illetve adhat ellenpéldát. Azonban a tipikusan nagyon nagy (bizonyos esetekben végtelen) állapotterek gátolhatják a modellellenőrzés praktikus felhasználását. Többféle módszer létezik ezen akadály leküzdésére. Egyik gyakran alkalmazott megközelítés a korlátos modellellenőrzés, amikor eleve korlátozzuk a verifikáció hatókörét azáltal, hogy a program viselkedését csak egy megszabott mélységig értékeljük ki. Egy másik lehetőség az absztrakció alkalmazása, melynek segítségével csoportosíthatóak a program állapotai és ezzel lényegesen kisebb állapottér fölött működhet a modellellenőrző.

Többszálú programok esetén még rosszabbul skálázódik a modellellenőrzés, ezért specializált módszerek szükségesek a komplexitás leküzdéséhez. A hallgató feladata, hogy diplomamunkája keretében vizsgálja a párhuzamos programok absztrakcióalapú, illetve korlátos modellellenőrzési verifikációs lehetőségeit.

A hallgató feladatának a következőkre kell kiterjednie:

- Mutasson be létező absztrakció alapú és korlátos modellellenőrzési technikákat párhuzamos programok verifikációjára!
- Elemezze, hogyan lehetne javítani a létező megközelítések hatékonyságán algoritmikus szempontból!
- Matematikailag igazolja a kifejlesztett vagy módosított algoritmusok helyességét!
- Implementálja a bemutatott algoritmusokat és a kifejlesztett optimalizációkat egy modellellenőrző eszközben!
- Értékelje ki az implementált algoritmusok teljesítményét párhuzamos programok egy nagyobb méretű benchmark készletén!

Tanszéki konzulens: Bajczi Levente (doktorandusz)

Budapest, 2024. szeptember 12.

…………………………………
Dr. Dabóczi Tamás
tanszékvezető
egyetemi tanár, DSc.

Budapesti Műszaki és Gazdaságtudományi Egyetem
Villamosmérnöki és Informatikai Kar
Mesterséges Intelligencia és Rendszertervezés Tanszék

1117 Budapest, XI. Magyar tudósok körútja 2.
I ép. E szárny, IV. em. E444.
Tel.: (+36 1) 463-2057, fax: (+36 1) 463-4112

HALLGATÓI NYILATKOZAT

Alulírott *Telbisz Csanád*, szigorló hallgató kijelentem, hogy ezt a szakdolgozatot meg nem engedett segítség nélkül, saját magam készítettem, csak a megadott forrásokat (szakirodalom, eszközök stb.) használtam fel. Minden olyan részt, melyet szó szerint, vagy azonos értelemben, de átfogalmazva más forrásból átvettem, egyértelműen, a forrás megadásával megjelöltem.

Hozzájárulok, hogy a jelen munkám alapadatait (szerző(k), cím, angol és magyar nyelvű tartalmi kivonat, készítés éve, konzulens(ek) neve) a BME VIK nyilvánosan hozzáférhető elektronikus formában, a munka teljes szövegét pedig az egyetem belső hálózatán keresztül (vagy autentikált felhasználók számára) közzétegye. Kijelentem, hogy a benyújtott munka és annak elektronikus verziója megegyezik. Dékáni engedéllyel titkosított diplomatervek esetén a dolgozat szövege csak 3 év eltelte után válik hozzáférhetővé.

Budapest, 2024. december 6.

Telbisz Csanád
hallgató

Kivonat

A többmagos processzorok biztonságkritikus rendszerekben történő térhódításának köszönhetően egyre gyakrabban használnak többszálú programokat. A szoftververifikáció komplexitása viszont tovább nő a párhuzamosság megjelenésével a program szálainak nagyszámú lehetséges átlapolódása miatt. A komplexitásnövekedés eredménye, hogy a megfelelő tesztlefedettség elérése még nagyobb kihívást jelent, a naiv verifikációs technikák pedig gyakran gyakorlatilag használhatatlanná válnak. A formális verifikációs technikák megadhatják a kívánt biztonsági garanciákat, ugyanakkor kifinomult algoritmusokra van szükség a párhuzamos rendszerek komplexitásának kezeléséhez.

A részbenrendezésen alapuló redukció (partial order reduction) hatékony technika a modellellenőrzés területén az átlapolódások nagy számának kezelésére. A módszer felismeri, ha a program bizonyos utasításai egymástól független műveleteket végeznek, és ez alapján nem minden szál átlapolódást vizsgál meg. Diplomamunkámban új elméleti keretrendszert adok, melyben a részbenrendezésen alapuló redukciós technikák absztrakcióalapú verifikációval kombinálhatók. Megközelítésem az utasítások közti függőség vizsgálatakor az aktuális absztrakcióban elérhető információkra is támaszkodik. A függőség okának absztrahálásával bizonyos műveletpárok függetlennek tekinthetők, így összességében csökkenthető a modellen belüli függőség. Következésképp a redukciós technika hatékonysága nő.

Az állapottér bejárása során a követő állapotok kiszámítása, vagyis az utasítások kiértékelése költséges feladat, melyhez gyakran egy SMT (Satisfiability Modulo Theories) probléma megoldása is szükséges. Sok esetben azonban egy program utasítás nincs hatással az ellenőrzött tulajdonságra. Ilyen esetekben egyszerűsíthető a követő állapotok kiszámítása. Számos algoritmus létezik, melyek statikusan elemzik a modellt és eltávolítják 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. Az én dinamikus megközelítésem az állapottér bejárása során észleli a szálak aktuális állapota alapján, hogy egy utasítás kihagyható-e.

Számos ígéretes modellellenőrzési technika létezik a párhuzamos programok formális verifikációjára, melyek a programot és az ellenőrzött követelményt SMT problémaként kódolják az érvelést az SMT megoldók erejére bízva. Ezen többszálú programok verifikációjára szolgáló eljárások lényege, hogy a párhuzamos szálak eseményeinek (változóhozzáféréseinek) konzisztens sorrendezését igyekeznek biztosítani. Az eljárás megtalálja és kiküszöböli a párhuzamos események körkörös precedenciájából fakadó inkonzisztenciát. Noha az SMT megoldók sok szempontból hatékonyak, mégiscsak általános célú eszközök, így számos lehetőség van doménspecifikus optimalizálásra. Munkámban egy ilyen kiegészítést javaslok, mely képes megtalálni az ütemezési következetlenségeket és ezekkel bővíteni a kódolt SMT formulát még az SMT megoldó eljárás megkezdése előtt. A megoldó keresési tere ezáltal jelentősen korlátozható, a verifikációs teljesítmény pedig jelentősen javítható.

Abstract

As multi-core processors gain popularity in safety-critical systems, multi-threaded programs are increasingly used. Concurrency introduces a new level of complexity into software verification due to the great number of possible thread interleavings. Achieving satisfying test coverage is even more challenging, and naive verification techniques often become practically infeasible due to this complexity. Formal verification techniques can provide the desired safety guarantees. However, sophisticated algorithms are needed to handle the complexity of a concurrent system.

Partial order reduction has proven to be an effective technique to address the problem of interleavings in model checking. The approach detects independent program statements and skips the exploration of certain thread interleavings based on this information. I present a novel theoretical framework that combines partial order reduction algorithms with abstraction-based verification. My approach relies on supplementary information from the applied abstraction when calculating dependencies between program operations. By abstracting the sources of dependency, certain operations are deemed independent, reducing interdependence within the model. Consequently, the effectiveness of partial order reduction is amplified.

Calculating successor states in software model checking is costly, often requiring solving a Satisfiability Modulo Theories (SMT) problem. However, in many cases, the evaluation of a program statement does not affect the verified property. Successor state calculation can be simplified in such cases. Existing algorithms, such as the cone-of-influence reduction, statically analyze the program and eliminate irrelevant variables. In concurrent software, however, the result of a statement may be used in one interleaving of threads while unused in another. Algorithms that statically analyze the program cannot simplify such statements. My on-the-fly approach detects whether a statement can be simplified during the state space exploration based on the current state of each thread.

Promising model checking approaches have been developed that encode the whole concurrent program and the verified property as an SMT problem, leaving the reasoning to the strength of SMT solvers. These methods for concurrent software verification ensure that events (variable accesses) of concurrent threads are consistently ordered. Inconsistencies in the form of cyclic precedence of concurrent events are detected and excluded. While SMT solvers are highly optimized, they remain general-purpose tools, leaving room for domain-specific optimization. I propose a domain-specific optimization to find scheduling inconsistencies and strengthen the encoding formula before starting the SMT-solving procedure. This way, the solver search space can be greatly reduced, and the overall verification performance can be significantly improved.

Chapter 1

Introduction

The presence of computerized solutions in industrial systems and everyday life highlights the need for reliable software. In safety-critical systems where faults are intolerable, guaranteed - mathematical - correctness is often required, which conventional verification techniques such as testing cannot offer. Formal software verification - a method that can mathematically prove safety guarantees for software - has been a field of active research in the last few decades [54, 30, 77, 1]. Rapid development in technology led to the increasing popularity of multi-core processors and multi-threaded programs in industrial systems. Today, multi-core processors are available for various targets, from personal computers 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. Nonetheless, functionally correct behavior is still crucial.

In *formal software verification*, verified properties are typically reachability criteria (whether a particular error state is reachable by any execution of the program), memory-safety (no memory leak or memory handling issue), termination (whether all program executions terminate), or data race freedom (data race cannot occur in the concurrent program). 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 program's state space [54]. Generally, the input of a model checking algorithm is a *model* and a *formal requirement* (or *specification*, *safety property* equivalently). The output is a verdict: the model is either *safe* or *unsafe*. The input model in the scope of this paper is a multi-threaded program. For reachability analysis, specific points of the verified program are marked as unsafe using *assertions* as formal requirements. If an assertion fails in any possible program execution, the reachability criterion is said to be violated.

Formal verification methods take a program and a safety property and try to prove or refute that the property holds for all possible program executions. The verification of concurrent software has always been a challenge due to the great number of possible thread interleavings. For concurrent programs, the verification aims to find a program execution (a specific interleaving of concurrent instructions) that violates the safety property of the verification or to prove that such execution does not exist.

Formal software verification suffers from the *state space explosion* problem since the state space of a program often grows exponentially with the number of variables and the size of their domains [35]. Efficient approaches to handle this problem are *abstraction* [54, 34] and *bounded model checking* (BMC) [29, 77], among other techniques. Abstraction is a complete and sound approach to model checking, which often induces huge computational costs, limiting the applicability of the technique. Bounded model checking trades com-

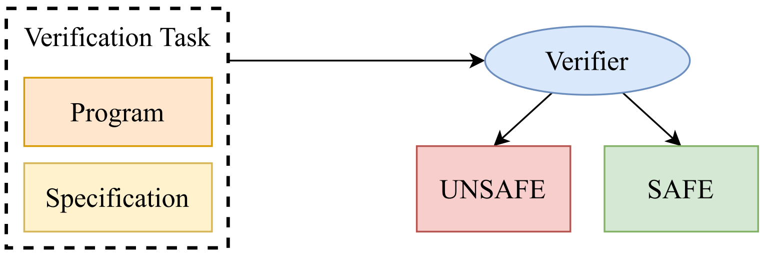


Figure 1.1: Formal Verification with Witness Validation.

pleteness of the approach for performance: it looks for bugs up to a certain depth of the program's behavior. In turn, BMC approaches typically faster. The bound can be applied in different ways: either by constraining the depth of the explored state space or by applying an upper bound for the number of loop iterations in the program. While BMC is often unsuitable for providing a complete proof of correctness of a program (e.g., when we have infinite loops), it is a powerful method for finding safety violations in the program.

Concurrency introduces a new level of complexity to software verification due to the great number of possible thread interleavings. The violation of the safety requirement may occur only in one interleaving of concurrent threads, while the erroneous behavior does not happen in other interleavings. Formal verification algorithms for concurrent programs have to explore all possible behavior. Many algorithms have been developed for concurrent software verification to tackle the complexity of thread interleavings: initially, most were based on exploring the state space or possible steps of the program [76, 44, 1, 67, 52].

One of the most well-known techniques for reducing the effect of the large number of thread interleavings is partial order reduction (POR) [51, 76]. *Partial order reduction* is a technique that reduces the number of interleavings explored by the model checker by considering only a subset of the interleavings. The technique is based on the observation that some interleavings are equivalent, and exploring all of them is unnecessary. The core idea is that the order of adjacent independent program statements does not matter, where dependency is determined by the interaction of statements (e.g., they write the same global variable). In this work, I explore the possibilities of combining partial order reduction with methods based on abstract state space exploration.

Some abstraction-based techniques like *cone-of-influence* (COI) reduction or program slicing eliminate model elements irrelevant to the verified property [18, 60]. It is a commonly used technique in model checking to ignore the program's irrelevant aspects, thus accelerating the verification process. However, traditional program slicing approaches face severe challenges when applied to concurrent programs. The main reason is that the interleavings of the threads can affect the program elements' relevance. In this work, I propose a new cone-of-influence approach designed explicitly for concurrent programs.

Efficient bounded model checking algorithms have also been developed for concurrent programs that symbolically encode the program and the safety property into a formula and then use a SAT or SMT solver to check the satisfiability of the encoded formula [8, 87, 61]. These techniques reason about the partial order of concurrent program instructions to ensure that the found data-flow resulting in a violation of the safety property corresponds to a valid scheduling of concurrent program instructions. In this work, I propose a novel optimization for reasoning about partial orders that significantly reduces the size of the solver search space.

Contributions. This work consists of the following main theoretical contributions to the field of formal verification of concurrent programs:

Thesis I. I combine abstraction-based methods and partial order reduction techniques in a general theoretical framework. I prove that applying partial order reduction in both the traditional and the proposed abstraction-based manner invariably uncovers error states in the abstract state space whenever the program can indeed reach an erroneous state. As a case study, I apply the general framework to creating an abstraction-aware variant of a widely used static partial order reduction algorithm inside a CEGAR loop. The contribution is presented in Chapter 3.

Thesis II. I propose a new statement simplification technique for concurrent programs. It considers the applied level of abstraction and the current thread interleaving during state space exploration to decide whether the statement can influence the safety requirement in the given context. If a statement is irrelevant in the current interleaving of threads, the evaluation of the statement is skipped, sparing the calculation time. The contribution is presented in Chapter 4.

Thesis III. I propose an optimized bounded model checking algorithm for verifying concurrent programs reasoning with partial orders. The proposed approach considerably trims the model search space by strengthening the encoded program formula before starting the verification decision procedure, thereby accelerating the verification. The contribution is presented in Chapter 5.

Experimental evaluations of the proposed approaches and other works related to the field are presented in each chapter. I also compare my solutions to existing state-of-the-art verifier implementations.

As a further practical contribution to the software verification community, I developed some concurrent programs in C (including a C version of the program in Figure 3.2) for benchmarking concurrent program verifiers. The tasks were submitted to the central repository[1] of the international Competition on Software Verification, SV-COMP [20].

[1]https://gitlab.com/sosy-lab/benchmarking/sv-benchmarks/-/merge_requests/1549

Chapter 2

Preliminaries

This section introduces the preliminary knowledge that is necessary to understand the methods introduced in this thesis. I assume that the reader is familiar with graph theory, first-order logic, the boolean satisfiability problem (SAT) and the basic concepts of formal verification.

2.1 Satisfiability Modulo Theories

The *satisfiability modulo theories* (SMT) problem is the decision problem of determining the satisfiability of a first-order logic formula given some background theories [42, 17]. For example, deciding whether the formula $(x < y-1 \ \wedge \ x = y+1) \ \vee \ x > -2*y$ has any real solutions is an SMT problem with linear arithmetic as a theory. SMT is expressive enough to be useful when formalizing a wide range of mathematical or engineering problems. On the other hand, SMT is well-defined and constrained which makes it possible to have efficient algorithms to tackle SMT problems. Therefore, SMT is often used as a base of model checking algorithms [17, 8, 82].

A *first-order theory* $\mathcal{T}$ is defined by a set of symbols (constant, function, and relation symbols) and a set of *axioms* which define the intended semantics of these symbols. A *model* specifies the interpretation of each constant, function and relation, and defines a nonempty entity set (or domain) which variables can instantiate. A $\mathcal{T}$*-model* is a model that also satisfies all axioms of theory $\mathcal{T}$. A formula Φ is $\mathcal{T}$*-satisfiable* if there is a model of Φ that is a $\mathcal{T}$-model.

Several algorithms exist for SMT solving, one traditional method is DPLL(T) [48, 17] which we use to present the concepts in this paper. Other SMT solving procedures often share similar ideas (e.g., conflict clauses as described below), so the method presented in this paper can be adapted to other techniques as well. The DPLL(T) algorithm initially takes the Boolean abstraction $\mathcal{B}(\Phi)$ of the input SMT formula Φ, where a Boolean abstraction means that each symbol in the original formula is simply interpreted as a propositional logic symbol forgetting about the theory (theory formulae become propositional variables). The Boolean abstraction of the above linear arithmetic example formula could be $(p \wedge q) \vee r$ where the propositional logic variables p, q, and r represent the theory formulae $x < y - 1$, $x = y + 1$, and $x > -2 * y$, respectively.

DPLL(T) uses a SAT solver to check the satisfiability of $\mathcal{B}(\Phi)$. If it is unsatisfiable, then Φ is necessarily unsatisfiable, too. However, if the SAT solver finds a model M of $\mathcal{B}(\Phi)$, it is not guaranteed to be a valid model of Φ. A theory solver must be used to determine whether M is consistent with the axioms of the theory. If an inconsistency is found, the theory solver generates a *conflict clause* c in propositional logic which represents a violation of the theory axioms. This way $\mathcal{B}(\Phi) \wedge \neg c$ cannot have the same model M anymore, i.e., the same type of inconsistency is avoided in the future. This procedure can be repeated until either $\mathcal{B}(\Phi)$ augmented with the conflict clauses becomes unsatisfiable or a model is found which is consistent with the theory axioms. In the first case Φ is unsatisfiable, while in the latter case, it is satisfiable with the found model.

Looking at our linear arithmetic example, the Boolean abstraction of the formula is satisfiable: e.g., $p = q = \top$ and $r = \bot$. The linear arithmetic theory solver checks this assignment if it is consistent with the axioms of linear arithmetic: in our case, it is not since $x < y - 1$ and $x = y + 1$ cannot hold at the same time. Therefore, the theory solver generates the conflict clause $p \wedge q$ which prevents the same inconsistency in further calculations. In the next iteration, the SAT solver finds a model with $r = \top$, and the theory solver also finds a model consistent with linear arithmetic (e.g., $x = 1, y = 0$).

In practice, DPLL(T) works with partial models where variables are gradually assigned either by creating a *decision point* or by propagation. Some variable values may be inferred (propagated) from the Boolean abstracted formula (in cases where setting the variable to the negated value would lead to unsatisfiability) but it is also possible to use theory propagation where the theory solver can deduce and propagate the values of variables based on the partial model and the theory axioms. Theory consistency checking and conflict clause generation is also performed based on partial models.

2.2 Representation

In this paper, I assume a computation model of concurrent programs where processes (threads) communicate via shared variables. I assume a sequential consistency memory model (except for Chapter 5 where I need to consider some aspects of weak memory models). Though it would be easy to incorporate extra features into the model (such as heap memory, dynamic thread creation or termination, and synchronization primitives), I strive to keep my presentation simple and skip these details. My implementation for the evaluation naturally supports these features.

Concurrent programs are represented by control-flow automata (CFA) [22]: each process has its own (conventional) CFA representation.

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

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

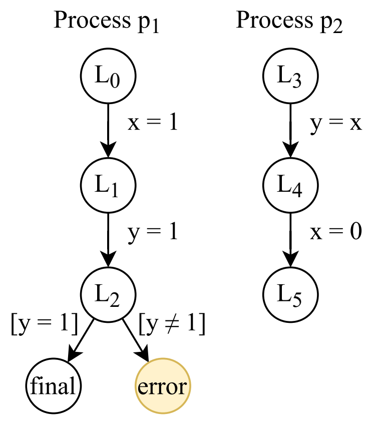
Each variable $v \in V$ has a domain D_v (the possible values for v), and possibly an initial value from its domain. A statement can be a deterministic assignment ($v = \mathit{expr}$), a non-deterministic assignment (*havoc* v) where the new value of v can be anything from its domain, or a guard condition ($[\mathit{cond}]$). For the verification of reachability properties,

```
int x, y;

void thread1() {
  x = 1;
  y = 1;
  assert(y == 1);
}

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

(a) C source code



(b) CFA of the program

Figure 2.1: CFA of a multi-threaded program

some CFA locations are marked as error locations: the program is safe if no error location can be reached by any of its processes.

Example 1. *The source code in Figure 2.1a 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 assertion in the program states the safety requirement. Figure 2.1b depicts the CFA of the two processes. Location* error *is an error location of the CFA.*

I define transition systems (state spaces) as follows:

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

An action α is an *outgoing* action from a state s if there is a transition $(s, \alpha, s') \in T$ for some $s' \in S$. I use the following notations:

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

The state space of a program is a transition system where a state stores the CFA locations of all processes and the values of all variables. A state is an error state if any of the processes is in an error location in the state. 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: $\mathit{expr}(s) := \bigwedge_{v \in V}(v = s(v))$.

[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 for simplifying the formalization, so I omit these details from this example as well.

An action of a transition corresponds to a statement of a single process (processes step asynchronously). I use the Greek alphabet for actions, and I write p_α for the process of action α. Note that $\mathit{written}(\alpha)$ 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: $w.v$. Often, actions are used in notations instead of transitions: an action α used in such a context means a transition with α as its action. I also refer to action sequences as traces. If we have a trace from a state that leads to an error state, I call this trace an error trace.

2.3 Abstraction

An abstraction can be defined with an abstract domain, a precision, and a transfer function [25]. In this paper, I use a simplified definition of the abstract domain:[2]

Definition 3. An abstract domain is a tuple (S, expr), where S is a set of abstract states, and $\mathit{expr} : S \mapsto \mathit{FOL}$ is an expression function mapping 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: I refer to the location of process p in a 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 $\mathit{expr}(c)$ implies $\mathit{expr}(s)$. In my notation, I use s for abstract states and c for concrete states. An abstract state is an error state if any process is in an error location. An abstract trace $w = \alpha_1...\alpha_k$ from the abstract state s_0 ($s_0 \xrightarrow{\alpha_1} ... \xrightarrow{\alpha_k} s_k$) is *feasible* if w is also a trace in the concrete state space ($c_0 \xrightarrow{\alpha_1} ... \xrightarrow{\alpha_k} c_k$) with $c_i \models s_i$; otherwise, w is spurious from s_0. If w is feasible, the sequence of concrete states $c_0, ..., c_k$ is a *concretization* of w. 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 $c \models s$ [25].

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

2.3.1 Common Abstract Domains

In this section, I briefly explain two frequently used abstract domains: explicit-value abstraction [23] and predicate abstraction [47].

Explicit-value abstraction. In explicit-value abstraction, an abstract state is defined by the CFA locations of processes and an abstract variable assignment. The domain D_v of each variable $v \in V$ is extended with a top value $\top_v$. The top value represents an unknown value for the variable. The precision in explicit-value abstraction is the subset of variables $\Pi \subseteq V$ that are explicitly tracked in this abstraction; $\mathit{vars}(\Pi) = \Pi$. The values of other

[2]Abstract states are usually defined as a semi-lattice with a partial order [25], but we do not need those details for this paper, so I simplify.

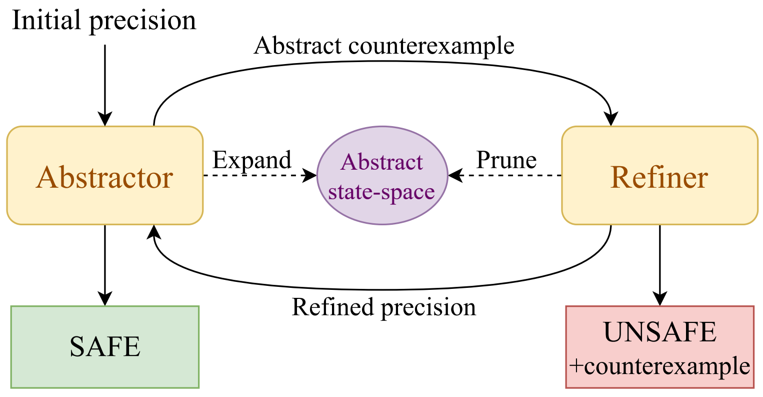


Figure 2.2: The CEGAR-loop

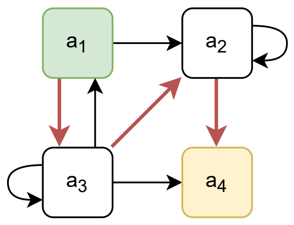
variables are unknown in all abstract states: $s(v) = \top_v$ for each $v \in V \setminus \Pi$. The expression function is $expr(s) = \bigwedge_{v \in V, s(v) \neq \top_v} (v = s(v))$. The result of the transfer function is based on the strongest post-operator under abstract variable assignment [23].

Predicate abstraction. In predicate abstraction, an abstract state is defined by the CFA locations of processes and a combination of FOL predicate. The precision is the 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. The transfer function returns the strongest combination of predicates in the precision that is entailed by the source state and the operation.

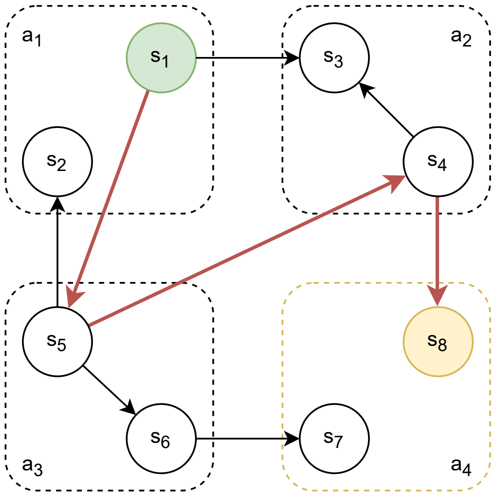
2.3.2 Counterexample-Guided Abstraction Refinement

Counterexample-Guided Abstraction Refinement (CEGAR) [34] is an abstraction-based model checking algorithm. CEGAR starts from a coarse abstraction and iteratively refines the abstraction until it can prove or disprove the analyzed property. The core of the algorithm is the *CEGAR-loop* (see Figure 2.2) which consists of the *abstractor* and the *refiner*. The abstractor builds the abstract state space (an *abstract reachability graph*, ARG [24]) over an abstract domain with a given precision. 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 also safe. On the other hand, when an abstract error is reachable, the refiner checks whether it is a *feasible* or a *spurious* abstract counterexample. The counterexample is an alternating sequence of abstract states and actions from the initial abstract state to an abstract error state. The refiner checks whether this trace is feasible or not, that is, whether there is a concrete variable assignment for each state of the trace that does not contradict the abstract state expressions and the actions of the trace. If feasible, the model is unsafe. If spurious, the precision is refined. The abstract state space is built with this refined precision in the next iteration.

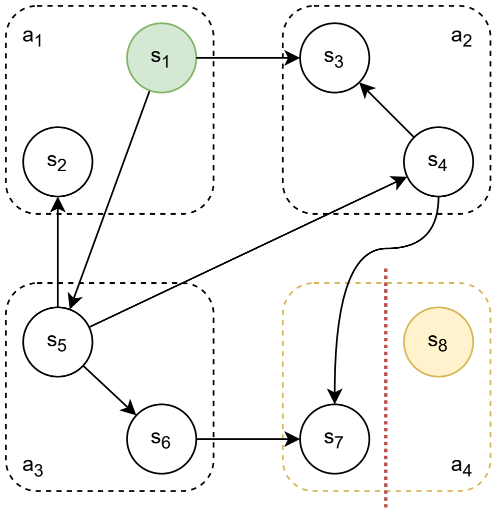
Example 2. *Consider the example from Figure 2.3. We have the abstract state space S from Figure 2.3a with the abstract error state s_3. The abstractor finds the abstract counterexample highlighted in Figure 2.3a. This counterexample leads from the abstract initial state s_0 to the abstract error state s_3 in the abstract state space S: $s_0 \to s_2 \to s_1 \to s_3$. 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.*



(a) Abstract state space S with an abstract counterexample



(b) Feasible counterexample in S_1



(c) Spurious counterexample in S_2

Figure 2.3: CEGAR counterexamples

First, let us assume that the concrete state space abstracted by S is S_1 from Figure 2.3b. 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: $c_0 \to c_4 \to c_3 \to c_7$ 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 2.3c 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.

Chapter 3

Abstraction-Based Partial Order Reduction

Partial order reduction (POR) is an effective technique for handling concurrency, and abstraction is an efficient approach to handling data in model checking. I present a general theoretical framework for combining these model checking paradigms where the advantages of using the two techniques simultaneously are also exploited. I also present a concrete verification algorithm using POR and abstraction together as a case study of my general approach. The novelty of my method lies in the general formulation of POR used during abstract state space exploration. Existing approaches combining these techniques are typically specific to POR or abstraction algorithms [33, 85, 70, 79]. A general approach for combining abstraction and commutativity checking is proposed by *Farzan et al.* [45]. However, that paper does not investigate POR algorithms, simply the general properties of abstract commutativity relations. My work shows how the idea of abstract commutativity applies generally to POR in abstract state space exploration algorithms.

The core concept of POR is to identify equivalent executions [51]. Then, it is enough to check a single representative from each equivalence class. Identifying equivalent interleavings is based on the interaction of threads: dependency is defined between the interacting program operations. Traditionally, a syntactic over-approximation is used as a method of calculating dependency for partial order reduction: two actions are independent if they do not use common shared variables (and the two actions belong to different processes) [51]. For example, let us have a state in the (concrete) state space of the program with two enabled actions from different threads: x++ and y++. No matter, in what order we explore the two actions (x++, y++ or y++, x++), we will reach the same state since the actions operate on different variables. So we can skip the exploration of one of the two paths.

When it comes to combining POR with abstraction, we face the problem that traditional approaches may calculate invalid dependency relations: it is not trivial to apply POR in an abstract state space where the values of variables are not tracked explicitly [33, 79]. Example 3 shows a situation where syntactically independent actions are not commutative in an abstract state space.

Example 3. *Assume that we have an abstraction where we only track the predicates $x > y$ and $x > y + 1$ about our variables. Figure 3.1 demonstrates that two actions that are independent in the traditional sense (no shared variable) may be dependent in the abstract state space. Actions x++ and y++ are dependent since they are not commutative in the abstract state space due to the difference in the labeling of the abstract states. Later, Figure 3.4 shows a case where actions with different variables can even disable each other.*

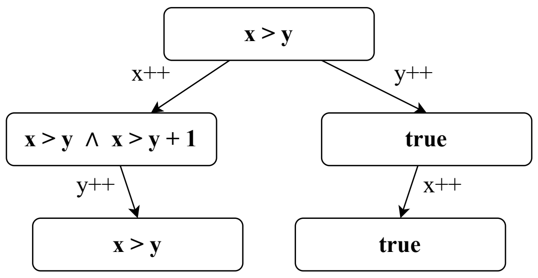


Figure 3.1: Syntactically independent actions that are not commutative in the abstract state space

Even though the syntactic over-approximation of dependency is not a valid dependency relation in the abstract state space, I show that using it for partial order reduction will always find an error state in the abstract state space if an error is reachable in the concrete state space. Furthermore, I restrict the calculation of dependency, to consider fewer actions dependent in an abstract state space. Intuitively, if the source of dependency between two actions is ignored due to abstraction, it is needless to consider these actions dependent. That is, two actions using the same shared variable are only considered dependent if we track any information about any of their shared variables in the abstraction. As a basic example, take the actions $x = 0$ and $x = 1$. Existing methods consider these actions dependent since they both write x. However, my approach allows considering these actions independent when we do not track any information about x in the abstraction.

To further motivate my approach, it is possible to achieve exponential gains in terms of the number of explored interleavings by using my abstraction-based algorithm for partial order reduction. Consider the example from Figure 3.2 with $2N + 1$ processes. The safety of the program can be proven with abstraction by only tracking the predicates $z \bmod 2 = 0$ and $x = 0$ about our variables. As $z \bmod 2 = 0$ is an invariant of the loop of p_0, x will get the value 0 which satisfies the assertion. To prove this, traditional methods would explore all interleavings of instructions using y since these actions would be considered dependent due to y: processes $p_1 - p_{2N}$ have $2N!$ interleavings (potentially resulting in different values of y) not considering the loop of p_0; together there are even more possible interleavings. However, my algorithm notices that we do not track any information about y, so it will not consider the actions using y dependent: this way, my method explores a single execution and guarantees the safety of the program.

Summarizing my contributions: I introduce a general abstraction-based partial order reduction approach which is independent of both the underlying POR algorithm and the abstract domain and which uses the information coming from the abstraction to boost partial order reduction. By proving the correctness of using the new abstraction-based dependency relation, I also prove that using the traditional dependency relation for partial order reduction in an abstract state space is correct as well (which is also a non-trivial statement as testified by Example 3). I have implemented and evaluated my method in the abstraction-based verification tool THETA[83].

The organization of this chapter is as follows. First, Section 3.1 introduces the theoretical background. Section 3.2 presents the novel general approach to combine POR and abstraction in an abstract state space exploration. Section 3.3 demonstrates an algorithm implementing abstraction-based POR. Finally, I evaluate the approach in Section 3.4 using the algorithm introduced in Section 3.3.

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

Process p_0

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

Processes $p_1, ..., p_N$

```
y := y+1
```

Processes $p_{N+1}, ..., p_{2N}$

```
y := y*y
```

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

Figure 3.2: Motivational example for possible exponential gain

3.1 Preliminaries

We need to discuss some further preliminaries before we can dive into the contributions of this chapter. First, I introduce some extra notation used in this chapter, then I present the basic concept of partial order reduction.

For the states s_1, s_2, and the set of variables $U \subseteq V$, I write $s_1 = s_2$ *on* U to denote that $s_1(v) = s_2(v)$ for each $v \in U$. The notation $w \backslash t$ means the transition sequence obtained from w by removing the first occurrence of t from w.

A transition system is action-deterministic if $|I| \leq 1$ and $|\alpha(s)| \leq 1$ for any state $s \in S$ and action $\alpha \in A$ [13]. The state space of a program is not action-deterministic due to non-deterministic assignments, and uninitialized variables. However, *unknown* is a possible value for variables when using abstraction (see details in Section 2.3): an uninitialized variable or a variable after a non-deterministic assignment gets the specific value *unknown*. This way, the state space becomes action-deterministic. In an action-deterministic transition system, I use $\alpha(s)$ for the single state s' with $(s, \alpha, s') \in T$. Partial order reduction algorithms are classically formulated for action-deterministic transition systems [46, 13, 12]. In some cases, instead of using the term action-deterministic, it is said that control non-determinism is allowed [3]. Some works investigate partial order reduction algorithms in non-deterministic state spaces, also considering types of abstraction introducing non-determinism [58]. In those settings, partial order reduction algorithms need to satisfy different properties. However, this research direction is out of this work's scope, so I assume that the abstract state space is action-deterministic.

A further assumption that we need for the algorithms in this chapter is that the state space is finite. For data variables, this is not a real restriction in most cases, as the domain of program variables are typically bounded (e.g., 32-bit integers in most languages) or abstraction can take care of a finite domain (e.g., the size of the domains of each predicate in predicate abstraction is 2). The restriction concerns concurrent programs where new threads are created dynamically in an infinite loop. Naturally, in real scenarios, we do not need an unbounded number of threads (and in fact, the limitations of the execution environment also limits the possible number of running threads at the same time), so this is also only a theoretical limitation. Besides, it is a common assumption to take in the papers presenting partial order reduction algorithms [2, 84, 51].

3.1.1 Partial Order Reduction

Partial Order Reduction (POR) is a well-known technique for avoiding the exploration of redundant thread interleavings in the verification of a multi-threaded program [51]. 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* [73]. 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. The general formulation of dependency is as follows [51]:

Definition 4 (Valid Dependency Relation). Let $TS = (S, A, T, I)$ be an action-deterministic transition system. Let $D \subseteq A \times A$ be a binary, reflexive, and symmetric relation. D is a *valid dependency relation* if for all $\alpha, \beta \in A$, $(\alpha, \beta) \notin D$ (α and β are *independent*) implies that the following two conditions hold for all $s \in S$:

- if $\alpha \in \mathit{enabled}(s)$, then $\beta \in \mathit{enabled}(s)$ iff $\beta \in \mathit{enabled}(\alpha(s))$, *and*
- if $\alpha, \beta \in \mathit{enabled}(s)$, then $\beta(\alpha(s)) = \alpha(\beta(s))$.

α and β are *dependent* if they are not independent. ▪

The first condition means that independent actions can neither disable nor enable each other. The second property states that independent actions commute. Sometimes, *dependency of transitions* is used in this paper: by the dependency of transitions, I mean dependency of their actions. Note that I will introduce relations in this work that are not *valid* dependency relations; however, for semantic reasons, I will refer to them as dependency relations - without the label *valid*.

Since the goal of POR is to avoid exploring multiple traces leading to the same state, the definition cannot be used directly for determining dependency (two actions should be explored in both orders to decide whether they commute). An appropriate approximation of the dependency relation in an abstract state space is a main contribution of this work.

Many algorithms have been introduced for partial order reduction in the last decades. My abstraction-based extension is orthogonal to the underlying POR algorithm. In this work, I build my presentation on the concept of source sets which is the core of optimal dynamic partial order reduction [1].

3.2 Abstraction-Aware Partial Order Reduction

This chapter describes how partial order reduction can be integrated into an abstraction-based model checking algorithm. Since the core concept of the approach is to consider extra information about the used abstraction when applying partial order reduction, I call the algorithm *abstraction-aware partial order reduction*. My approach is independent of the underlying POR algorithm as well as the applied abstract domain. The only requirements are that CFA locations of all processes must be explicitly tracked, and the abstract state space must be action-deterministic, that is the transfer function must return a single successor for each state and operation.

3.2.1 Dependency relations

In my algorithms, different dependency relations are used for partial order reduction. It is important to note that these are not necessarily valid dependency relations in all transition systems. First, I define the syntactic dependency relation.

3.2.1.1 Syntactic Dependency Relation

A syntactic dependency relation denoted by D_S is the classically used syntactic over-approximation of dependency[51]. That is, two actions $(\alpha, \beta) \in D_S$ (α and β are dependent) iff:

- α and β are actions of the same process, *or*
- $vars(\alpha) \cap vars(\beta) \neq \emptyset$, and at least one variable in $vars(\alpha) \cap vars(\beta)$ is written by α or β.

The syntactic dependency relation is a valid dependency relation in the concrete state space [51]. However, in general, it is not a valid dependency relation in the abstract state space; see the motivating example Example 3.

3.2.1.2 Abstraction-Based Dependency Relation

I introduce a new dependency relation for abstract state spaces (similar conditions are proposed in [45]). An abstraction-based dependency relation D_Π is also a syntactic approximation. However, it is defined in an abstraction with respect to the precision of the abstraction.

Definition 5. Let us have an abstract state space built with precision Π, and let D_Π be a binary, reflexive, and symmetric relation. Two actions $(\alpha, \beta) \in D_\Pi$ (α and β are dependent with respect to precision Π) iff:

- α and β are actions of the same process, *or*
- $vars(\alpha) \cap vars(\beta) \cap vars(\Pi) \neq \emptyset$, and at least one variable in $vars(\alpha) \cap vars(\beta) \cap vars(\Pi)$ is written by α or β. ▪

In other words, the second condition means that α and β may still be independent if they use common variables. They are only dependent when the abstraction stores any information about any variable that they both access. Note that D_Π is a subset of the syntactic dependency relation D_S for any precision Π: the first condition is the same for both relations, and the second condition of D_Π implies the second condition of D_S. As a consequence, D_Π is not a valid dependency relation in the abstract state space either (same counterexample from Example 3). Furthermore, D_Π is not necessarily a valid dependency relation in the concrete state space.

Example 4. *To illustrate the difference between the syntactic and the abstraction-based dependency relation, consider the actions* `x=1` *and* `[x>0]`*. They are dependent based on the syntactic dependency relation since they both use the variable x, and the first action writes it. For the abstraction-based dependency relation, we need an abstraction with a precision. First, let the precision be* $\Pi = \{x < y, z = 1\}$*: then the two actions are dependent in*

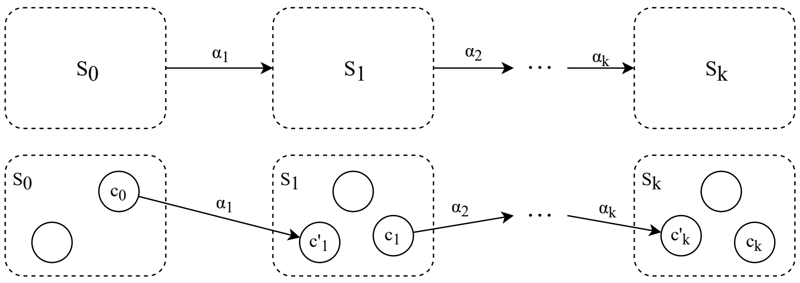


Figure 3.3: Abstract trace with a partial concretization

D_Π since we have information about x in this abstraction. However, when the precision is $\Pi = \{0 < y, z = 1\}$, then our actions are independent in D_Π as the value of x is completely ignored in this abstraction.

As we have seen, the introduced relations are not valid dependency relations. However, I show in the next sections that using these relations (D_Π in particular) for partial order reduction in an abstract state space with precision Π, feasible errors are still found.

3.2.2 Partial Feasibility

First, I generalize the concept of abstract trace feasibility by introducing partial feasibility. Intuitively, a partial concretization of an abstract trace is a partial variable assignment for each abstract state of the trace with only a subset of variables assigned so that the partial variable assignment does not contradict the abstract state expressions. The motivation for introducing partial feasibility is that variables ignored in the abstraction may spoil feasibility.

Example 5. *Let us have a program with the following three independent actions (with the variables being initially zero):*

α: `x=1` | β: `[y=0] reach error` | γ: `[z=1] z=0`

Clearly, γ is not enabled in the initial concrete state due to z being zero. Let us have the abstraction where only the variables x and y are tracked explicitly. Now, γ is enabled in the abstract initial state as we have no information about z. A partial order reduction algorithm may explore the trace $w = \gamma.\alpha.\beta$ and no other traces as all actions are independent. Even though w is not feasible, it will be enough for us in a sense that is formalized by partial feasibility.

Definition 6. An abstract trace $w = \alpha_1 ... \alpha_k$ from an abstract state s_0 ($s_0 \xrightarrow{\alpha_1} ... \xrightarrow{\alpha_k} s_k$) is *partially feasible* for the set of variables $P \subseteq V$ if there are concrete states $c_0, ..., c_k$ such that:

- $c_i \models s_i$ for each $0 \leq i \leq k$, and
- for each $0 \leq i < k$, $\exists c'_{i+1}$ such that $c_i \xrightarrow{\alpha_{i+1}} c'_{i+1}$, $c'_{i+1}(p) = c_{i+1}(p)$ for each process p, and $c'_{i+1} = c_{i+1}$ on P.

We refer to $c_0, ..., c_k$ as a *partial concretization* of w for P. ▪

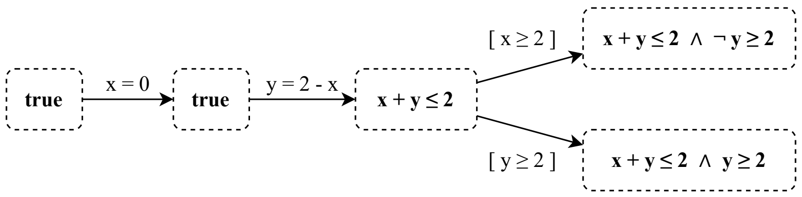


Figure 3.4: Example of a partially infeasible abstract trace

If w consists of a single action α, I say that α is a partially feasible action from the abstract state. Note that w being partially feasible for V means that w is feasible. The connection between feasibility and partial feasibility is straightforward: if w is not partially feasible for a $P \subseteq V$, then w is not feasible; and if w is feasible, then w is partially feasible for any $P \subseteq V$. In practice, I will use $P = vars(\Pi)$ for a precision Π: thus, partial feasibility will only depend on variables that we have information about in the current abstraction.

Example 6. *Figure 3.3 illustrates partial feasibility: we have the abstract trace $\alpha_1 ... \alpha_k$ from the abstract state s_0 and it has a partial concretization $c_0, ..., c_k$ (for the sake of simplicity, I omitted the values of variables from the figure).*

We can see an example of an abstract trace that is not partially feasible for $\{x, y\}$ in Figure 3.4. The figure shows an abstract state space built with a precision consisting of the predicates $x + y \leq 2$ and $y \geq 2$. The state expression of the initial state is true*, and it remains* true *even after executing $x = 0$ since none of the predicates or their negated form is a consequence of this action. However, $y = 2 - x$ implies $x + y \leq 2$. It still allows both $[x \geq 2]$ and $[y \geq 2]$ to be enabled. Even though $x = 0$, $y = 2 - x$, $[x \geq 2]$ is an abstract trace from the initial state, it is trivially not partially feasible since there is no possible partial variable assignment for $vars(\Pi) = \{x, y\}$ meeting the requirements of Definition 6 due to x. Also note that the actions $[x \geq 2]$ and $[y \geq 2]$ (belonging to different processes) disable each other even though they are independent based on the syntactic dependency relation.*

The following lemma states that if we have a sequence of concrete states $c_0, ..., c_k$ that satisfy the conditions in Definition 6, then there is indeed an abstract trace in the abstract state space whose partial concretization is the sequence $c_0, ..., c_k$.

Lemma 1. Let Π be the precision of the abstraction and $P = vars(\Pi)$; and $c_0, ..., c_k$ be concrete states such that for each $0 \leq i < k$, $\exists c'_{i+1}$ with $c_i \xrightarrow{\alpha_{i+1}} c'_{i+1}$, $c'_{i+1}(p) = c_{i+1}(p)$ for each process p, and $c'_{i+1} = c_{i+1}$ on P. Let s_0 be any abstract state with $c_0 \models s_0$. Then: $\alpha_1 ... \alpha_k$ is an abstract trace that exists in the abstract state space from s_0. ▪

Proof. The abstract state space being an over-approximation of the concrete state space means that if there is a transition $c_i \xrightarrow{\alpha_{i+1}} c'_{i+1}$ in the concrete state space, then there is a transition $s_i \xrightarrow{\alpha_{i+1}} s_{i+1}$ in the abstract state space where $c_i \models s_i$ and $c'_{i+1} \models s_{i+1}$. The expression function of abstract states only uses the variables in the precision. Thus, from $c'_{i+1}(p) = c_{i+1}(p)$ for each process p, and $c'_{i+1} = c_{i+1}$ on P, it follows that $c'_{i+1} \models s_{i+1}$ implies that $c_{i+1} \models s_{i+1}$. Now, with induction for i from 0 to $k - 1$, we get that $\alpha_1 ... \alpha_k$ is an abstract trace in the abstract state space from s_0. □

3.2.3 Relaxed Partial Order Representation

We can connect the partial order representation defined by a dependency relation with partial feasibility. Let $w_1 \approx^{\Pi} w_2$ denote that abstract traces w_1 and w_2 can be transformed into each other by successively swapping adjacent actions that are independent in D_Π. Thus, the relation $\approx^\Pi$ defines equivalence classes (Mazurkiewicz traces [73]) on abstract traces. The following theorem states that either all abstract traces are partially feasible in such an equivalence class or none.

Theorem 1. Let us have an abstract state space S built with precision Π, let $s \in S$ be an abstract state, w_1 and w_2 be transition sequences with $w_1 \approx^\Pi w_2$, and $P = vars(\Pi)$.

Then, w_1 is a partially feasible abstract trace from s for P iff w_2 is a partially feasible abstract trace from s for P. ▪

Proof. I prove that swapping two adjacent actions independent based on D_Π in a partially feasible abstract trace w will result in an abstract trace w' that is also partially feasible. The case is symmetric for w_1 and w_2: I can assume that w_1 is partially feasible. Since $w_1 \approx^\Pi w_2$ means that the partially feasible abstract trace w_1 can be transformed into w_2 by successively swapping adjacent independent actions, and partial feasibility is preserved in each step, it follows that w_2 is partially feasible.

Let $w = q.\alpha.\beta.r$ for some traces q and r ($s \xrightarrow{q} s_q \xrightarrow{\alpha} s_\alpha \xrightarrow{\beta} s_{\alpha\beta} \xrightarrow{r} s_w$) and $w' = q.\beta.\alpha.r$ from state s with w being partially feasible from s for P, and $(\alpha, \beta) \notin D_\Pi$. I check that w' is also partially feasible from s for P. Let $A = vars(\alpha)$, and $B = vars(\beta)$.

Since w is partially feasible, we have a partial concretization $c, ..., c_q, c_\alpha, c_{\alpha\beta}, ..., c_w$ of w with $c_q \models s_q$, $c_\alpha \models s_\alpha$, $c_{\alpha\beta} \models s_{\alpha\beta}$. Ignoring the end of w, we get that $c, ..., c_q, c_\alpha, c_{\alpha\beta}$ is a partial concretization of $q.\alpha.\beta$ for P. My goal is to show that there is a partial concretization $c, ..., c'_q, c_\beta, c_{\beta\alpha}$ of $q.\beta.\alpha$ such that $c_{\alpha\beta} = c_{\beta\alpha}$.

Throughout the proof, I will compare the values of variables in different concrete states. For this, as a reference, let us make the following observations:

Observation 1 *Since $(\alpha, \beta) \notin D_\Pi$, $A \cap B \cap P = \emptyset$ or neither α nor β modifies any variable in $A \cap B \cap P$. In both cases, $c_1(v) = c_2(v)$ for each variable $v \in A \cap B \cap P$ for any concrete states c_1, c_2 with $c_1 \xrightarrow{\alpha} c_2$ or $c_1 \xrightarrow{\beta} c_2$.*

Observation 2 *The action α can only change the values of variables in A, so if we have concrete states c_1 and c_2 with $c_1 \xrightarrow{\alpha} c_2$, then $c_1 = c_2$ on $\overline{A}$. Similarly for β.*

In Figure 3.5, I visualize the concrete and abstract states appearing in the proof. The variable values are represented by Venn diagrams in each concrete state. Sets with the same colors (and numbers) indicate that variables belonging to them have the same values. The colored numbers in the text should be interpreted such that the set of variables mentioned just before the colored numbers is the union of sets represented by the numbers.

Using Definition 6 of partial feasibility for the partial concretization $c, ..., c_q, c_\alpha, c_{\alpha\beta}$: there is a concrete state c'_α such that $c_q \xrightarrow{\alpha} c'_\alpha$ and $c'_\alpha = c_\alpha$ on P $\boxed{1}\,\boxed{2}\,\boxed{4}\,\boxed{10}$; similarly, there is a concrete state $c'_{\alpha\beta}$ such that $c_\alpha \xrightarrow{\beta} c'_{\alpha\beta}$ and $c'_{\alpha\beta} = c_{\alpha\beta}$ on P $\boxed{1}\,\boxed{2}\,\boxed{10}\,\boxed{13}$.

Based on Observation 2 for α, $c_q = c'_\alpha$ on $\overline{A}$ $\boxed{0}\,\boxed{2}\,\boxed{4}\,\boxed{7}$. Putting it together, $c_q = c_\alpha$ on $P \setminus A$ $\boxed{2}\,\boxed{4}$. Let c'_q such that $c'_q(p) = c_q(p)$ for each process p[1], and $c'_q = c_q$ on $\overline{B} \cup P$ $\boxed{0}\,\boxed{1}\,\boxed{2}\,\boxed{3}\,\boxed{4}\,\boxed{5}$. Also, let $c'_q = c_\alpha$ on $B \setminus P$ $\boxed{8}\,\boxed{9}$. Defining c'_q this way, with Observation 1 we can conclude that $c'_q = c_\alpha$ on B $\boxed{1}\,\boxed{4}\,\boxed{8}\,\boxed{9}$.

[1]For better readability, I will omit CFA locations from now on. Based on the notations, all states with the same name differing only in an apostrophe have the same CFA locations for each process.

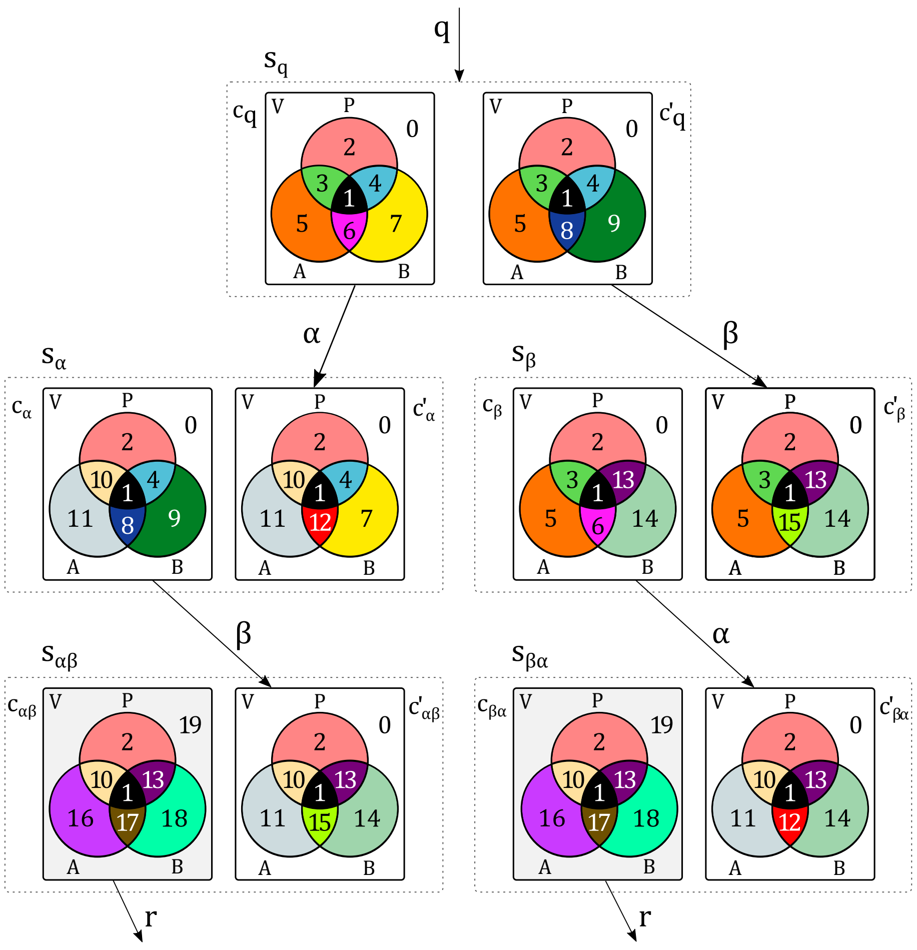


Figure 3.5: Variables of states in the proof of Theorem 1.

From partial feasibility we know that β is enabled in c_α. The guard condition of β only depends on the values of variables in B. Therefore, by the fact that $c'_q = c_\alpha$ on B, β is enabled in c'_q: there is a concrete state c'_β with $c'_q \xrightarrow{\beta} c'_\beta$.

Let c_β be a concrete state with $c_\beta = c'_\beta$ on $\overline{A} \cup P$ $\boxed{0}\,\boxed{1}\,\boxed{2}\,\boxed{3}\,\boxed{13}\,\boxed{14}$. Also, let $c_\beta = c_q$ on $A \setminus P$ $\boxed{5}\,\boxed{6}$. Using the definitions of c_β and c'_q, and Observation 2 for β: $c_\beta = c'_\beta = c'_q = c_q$ on $(A \cap P) \setminus B$ $\boxed{3}$. By the definition of c_β and Observation 1, $c_\beta = c_q$ on A $\boxed{1}\,\boxed{3}\,\boxed{5}\,\boxed{6}$. This way, the fact that $\alpha \in \mathit{enabled}(c_q)$ implies that α is enabled in c_β.

Now let $c'_{\beta\alpha}$ be such that $c_\beta \xrightarrow{\alpha} c'_{\beta\alpha}$. Let $c_{\beta\alpha}$ be such that $c_{\beta\alpha} = c'_{\beta\alpha}$ on P $\boxed{1}\,\boxed{2}\,\boxed{10}\,\boxed{13}$. Also, let $c_{\beta\alpha} = c_{\alpha\beta}$ on $\overline{P}$ $\boxed{16}\,\boxed{17}\,\boxed{18}\,\boxed{19}$. I show that $c_{\beta\alpha} = c_{\alpha\beta}$ even on P $\boxed{1}\,\boxed{2}\,\boxed{10}\,\boxed{13}$.

Again, consider that $c_q = c_\beta$ on A $\boxed{1}\,\boxed{3}\,\boxed{5}\,\boxed{6}$, thus α from c_q and c_β will result in the same new values for $v \in A$: $c'_\alpha = c'_{\beta\alpha}$ on A $\boxed{1}\,\boxed{10}\,\boxed{11}\,\boxed{12}$. From Observation 2 for β, and the definitions of the following states, $c_{\alpha\beta} = c'_{\alpha\beta} = c_\alpha = c'_\alpha = c'_{\beta\alpha} = c_{\beta\alpha}$ on $A \cap P$ $\boxed{1}\,\boxed{10}$. Symmetrically, $c_{\alpha\beta} = c'_{\alpha\beta} = c'_\beta = c_\beta = c'_{\beta\alpha} = c_{\beta\alpha}$ on $B \cap P$ $\boxed{1}\,\boxed{13}$. Finally, $c_{\beta\alpha} = c_q = c_{\alpha\beta}$ on $P \setminus (A \cup B)$, since neither α, β, nor the definition of any concrete state changed the value of such a variable v $\boxed{2}$.

Thus, $c_{\beta\alpha} = c_{\alpha\beta}$ on V (all variables). It can be easily seen that process locations are the same in $c_{\beta\alpha}$ and $c_{\alpha\beta}$, so $c_{\beta\alpha} = c_{\alpha\beta}$. Based on the above definitions of c_β and $c_{\beta\alpha}$, the above proven property that $c_{\beta\alpha} = c_{\alpha\beta}$, and using Lemma 1 for the sequence of concrete states $c, ..., c_q, c_\beta, c_{\beta\alpha}, ..., c_w$, we get that $w' = q.\beta.\alpha.r$ is a partially feasible abstract trace for P from the abstract state s which proves the theorem. $\square$

Also, take the following lemma which states that a partially feasible trace w can be extended to a partially feasible trace with an action that is enabled in the first concrete state of any partial concretization of w; it will be useful later.

Lemma 2. Let Π be the precision of the abstraction, s be an abstract state, and let the trace $w = w_1...w_n$ be a partially feasible trace from s for $vars(\Pi)$. Let $c_0, ..., c_n$ be a partial concretization of w from s, and let $\alpha \in \mathit{enabled}(c_0)$ such that $(\alpha, w_i) \notin D_\Pi$ for each $1 \leq i \leq n$. Then, $w.\alpha$ is also a partially feasible trace from s for $vars(\Pi)$. ▪

Proof. Throughout the proof, $1 \leq i \leq n$ and $0 \leq j \leq n$. Let $W = \bigcup_i vars(w_i)$, that is, the variables used by any action of w; and let $c_\alpha = \alpha(c_0)$.

I define the concrete states c'_j such that $c'_j = c_\alpha$ on $vars(\alpha) \setminus W$, and $c'_j(v) = c_j(v)$ otherwise. As for the CFA locations, $c'_j(p) = c_j(p)$ for each process $p \neq p_\alpha$, and $c'_j(p_\alpha) = c_\alpha(p_\alpha)$.

It can be easily checked that the conditions of Lemma 1 are met by the concrete states $c_0, c'_0, ..., c'_n$. For c_0 and c'_0, there is c_α with $c_0 \xrightarrow{\alpha} c_\alpha$, and $c_\alpha = c'_0$ on $vars(\Pi)$ (see the definitions of these states and Observation 1 for α and w_i). From c'_0, the conditions are met because $c_0, ..., c_n$ is a partial concretization of w for $vars(\Pi)$, and $c'_j = c_j$ on W, and $c'_j(p_{w_i}) = c_j(p_{w_i})$ for each process p_{w_i} (as $p_\alpha \neq p_{w_i}$ based on $(\alpha, w_i) \notin D_\Pi$).

Then, by Lemma 1, the trace $\alpha.w$ is partially feasible from s for $vars(\Pi)$ with the partial concretization $c_0, c'_0, ..., c'_n$. Since $(\alpha, w_i) \notin D_\Pi$, $\alpha.w \approx^\Pi w.\alpha$ which implies by Theorem 1 that $w.\alpha$ is partially feasible from s for $vars(\Pi)$. $\square$

So far, I have defined equivalence classes on abstract traces with the relation D_Π. Evidently, if a partially feasible abstract trace reaches an error state, then all other traces in its equivalence class reach an error state. It comes from the fact that error states are defined as states where any process is in an error location. Since all traces of an equivalence class contain the same actions (cf. they can be obtained from each other by successively swapping certain actions), if an action leads to an error location, it will reach the error in all traces regardless of the order of actions.

3.2.4 Source Sets for Abstraction-Aware Partial Order Reduction

We need an algorithm that explores only a single trace (or as few as possible) per equivalence class. Any POR algorithm could be chosen from the literature. Since I strive for generality, I use the concept of source sets [3] as necessary conditions on the correctness of a POR algorithm can be formulated with source sets [2]. I slightly modify its formulation to demonstrate that using the abstraction-based (or the traditional) dependency relation for partial order reduction in the abstract state space yields correct results.

Using the relation $\approx^\Pi$ on abstract traces, I relax the definition of *weak initials* and *source sets* from [3]:

Definition 7 (Weak Initials). Let s be an abstract state, Π be the precision of the abstraction, and w be a transition sequence from s such that w is partially feasible from s for $vars(\Pi)$. For w, the set $WI_s^\Pi(w)$ of weak initials in s is a set of actions: $\alpha \in WI_s^\Pi(w)$ iff there are transition sequences v_1 and v_2 such that $\alpha.v_1 \approx^\Pi w.v_2$, and $w.v_2$ is partially feasible from s for $vars(\Pi)$. ▪

Definition 8 (Source Sets). Let s be a state, and W be a set of transition sequences such that w is an abstract trace from s for each $w \in W$. A set $P \subseteq \mathit{enabled}(s)$ is a *source set for* W *in* s if for each transition sequence $w \in W$, $\mathit{WI}^{\Pi}_{s}(w) \cap P \neq \emptyset$. ▪

For abstract traces w and w' from the abstract state s, I use $w \sqsubseteq_s w'$ to denote that $w.v \approx^{\Pi} w'$ for some transition sequence v.

Example 7. *To demonstrate weak initials and source sets, take the program of Figure 3.2, and let s_0 be the abstract initial state. Consider two precisions Π_1 and Π_2. In the first case, let us explicitly track all variables of the program, so $\mathit{vars}(\Pi_1) = \{x, y, z\}$. In the other case, let us only track the predicates $z \bmod 2 = 0$ and $x = 0$, so $\mathit{vars}(\Pi_2) = \{x, z\}$.*

Let w_1 be the following trace of the program: first, the actions of processes p_1, ..., p_{2N} in order (actions writing y), then the actions of p_0 (with first branch of the if*). Let w_2 another trace, where the actions of p_0 come first, and then the actions of processes p_1, ..., p_{2N}. Evidently, both w_1 and w_2 are feasible traces from s_0 in both abstract state spaces (explored with precision Π_1 and Π_2).*

In the first case, the weak initials $\mathit{WI}^{\Pi_1}_{s_0}(w_1)$ only contains $y := y + 1$ (of p_1). To see this, note that no action α of the program is independent in D_{Π_1} with all preceding actions of α in w_1 due to y being used by the first actions of all threads: it is impossible to consecutively swap adjacent independent elements to obtain another trace w_1' starting with α ($\nexists w_1'$ such that w_1' starts with α and $w_1' \approx^{\Pi_1} w_1$). Similarly, $\mathit{WI}^{\Pi_1}_{s_0}(w_2) = \{z := z{+}2{*}y\}$. Since w_1 and w_2 are both transition sequences from s_0, neither $P_1 = \{y := y + 1\}$ nor $P_2 = \{z := z{+}2{*}y\}$ is a source set for all transition sequences in s_0, since $P_1 \cap \mathit{WI}^{\Pi_1}_{s_0}(w_2) = P_2 \cap \mathit{WI}^{\Pi_1}_{s_0}(w_1) = \emptyset$. In fact, the only source set in s_0 is enabled(s_0) itself.

In the second case, actions using y are not dependent in D_{Π_2}. So $\mathit{WI}^{\Pi_2}_{s_0}(w_1)$ and $\mathit{WI}^{\Pi_2}_{s_0}(w_2)$ both contain all enabled actions in s_0 (the first actions of all threads can be swapped with independent swaps). In fact, $\mathit{WI}^{\Pi_2}_{s_0}(w) = \mathit{enabled}(s_0)$ for any trace w starting from s_0. Therefore, any action $\alpha \in \mathit{enabled}(s_0)$ forms a source set alone since α is part of the weak initials of all traces starting from s_0: e.g., $\{z := z + 2 * y\}$ is a source set in s_0.

We can use this relaxed definition of source sets to formulate the correctness of abstraction-aware partial order reduction. The goal is to reduce the size of the abstract state space by exploring only a subset of enabled actions from each state. Generally, it is required from such a state space reduction that if an error state can be reached in the original state space, then an error state is also reachable in the reduced state space; this is to ensure that reachability analysis performed in the reduced and the original abstract state space yield equivalent results.

However, in our context of abstraction, it is enough to have an error state in the reduced abstract state space if there is a feasible trace from the initial abstract state to an abstract error state in the original abstract state space. That is, if abstract error states can only be reached with spurious traces (meaning that there is no error state in the concrete state space), there is no need to include an error state in the reduced state space.

Lemma 3. Let s be an abstract state, Π be the precision of the abstraction, and w be a trace from s such that w is partially feasible from s for $\mathit{vars}(\Pi)$. If $\alpha \in \mathit{WI}^{\Pi}_{s}(w)$ and $\alpha \notin w$, then

1. $w.\alpha$ is also partially feasible from s for $\mathit{vars}(\Pi)$, and
2. $\alpha.w \approx^{\Pi} w.\alpha$. ▪

Proof. From Definition 7 of weak initials, there are transition sequences v_1 and v_2 such that $\alpha.v_1 \approx^\Pi w.v_2$. This means that $w.v_2$ can be obtained from $\alpha.v_1$ by successively swapping adjacent independent actions. Since $\alpha \notin w$, $\alpha \in v_2$ necessarily. Also, for any action $\beta \in v_1$ but $\beta \notin w$, $\beta \in v_2$. Thus, any $\beta \notin w$ (including α) preceding any action $\gamma \in w$ in $\alpha.v_1$ must be independent with such γ actions: $(\beta, \gamma) \notin D_\Pi$.

So we can first move all such β after w by successive independent swapping steps without swapping α with any other such β. Thus, we get the transition sequence $w.\alpha.v_2'$ such that $\alpha.v_1 \approx^\Pi w.\alpha.v_2' \approx^\Pi w.v_2$. Since $w.v_2$ is partially feasible from s based on the definition of weak initials, $w.\alpha.v_2'$ is partially feasible as well by Theorem 1. Thus, its prefix $w.\alpha$ is partially feasible from s which proves the first statement.

For the second statement, take again that $(\alpha, \gamma) \notin D_\Pi$ for each $\gamma \in w$ since α precedes every other action in $\alpha.v_1$. This, by definition of the relation $\approx^\Pi$ means that $\alpha.w \approx^\Pi w.\alpha$.□

Let $W_\Pi(s)$ denote the set of partially feasible traces from an abstract state s for $vars(\Pi)$ in an abstract state space with precision Π. The following theorem (a modified version of the corresponding theorem in [2]) guarantees the correctness of a partial order reduction algorithm with certain conditions.

Theorem 2. Let S be the original abstract state space built with precision Π, and S_R be the reduced abstract state space obtained from S by restricting the set of actions that are explored from each state. If the following two conditions are satisfied:

1. for each state s in S_R, the set of explored actions is a source set for $W_\Pi(s)$ in s,
2. for each cycle in S_R, if an action α is enabled in all states of the cycle, then α is explored from some state of the cycle,

then for each state s in S_R and abstract trace w from s in S such that w is partially feasible for $vars(\Pi)$, there is a transition sequence w' in S_R such that $w \sqsubseteq_s w'$.[2] ▪

The proof proceeds similarly to the proof of the original theorem in [2] though some statements need more thorough justification.

Proof. I prove the theorem by induction on the length of w. The base case with $|w| = 0$ is trivial. For the inductive step, let us have the trace $w \in W_\Pi(s)$: by definition of $W_\Pi(s)$, w is partially feasible from s. By condition (1), some action $\alpha \in WI_s^\Pi(w)$ is explored from s in S_R. We have two cases:

1. $\alpha \in w$

   By definition, $\alpha \in WI_s^\Pi(w)$ means that $\alpha.(w \backslash \alpha) \approx^\Pi w$ (otherwise, $\alpha.v_1$ could not be transformed into $w.v_2$ by successive independent swapping steps when applying Definition 7 for α and w). This implies together with the assumption that w is partially feasible from s that $\alpha.(w \backslash \alpha)$ is also a partially feasible abstract trace from s based on Theorem 1. As a consequence, $(w \backslash \alpha)$ is a partially feasible abstract trace for $vars(\Pi)$ from $\alpha(s)$.

   From the induction hypothesis for state $\alpha(s)$ and the partially feasible trace $(w \backslash \alpha)$ from $\alpha(s)$, we know that the reduced state space S_R contains a trace w'' with $(w \backslash \alpha) \sqsubseteq_{\alpha(s)} w''$. This way, S_R also contains the trace $\alpha.w''$ from s. We now have

[2]Note that w' is not guaranteed to be partially feasible from s.

that $\alpha.(w\backslash\alpha) \sqsubseteq_s \alpha.w''$. From this, along with $\alpha.(w\backslash\alpha) \approx^\Pi w$, we can easily infer from the definitions of $\approx^\Pi$ and $\sqsubseteq_s$ that $w \sqsubseteq_s \alpha.w''$, so we can take $\alpha.w''$ as w' in the theorem.

2. $\alpha \notin w$

   Let $\alpha_1 = \alpha$. Then, using Lemma 3, $\alpha_1 \in WI_s^\Pi(w)$, $\alpha_1 \notin w$, and the assumption about the partial feasibility of w imply that $w.\alpha_1$ is a partially feasible trace from s, and $\alpha_1.w \approx^\Pi w.\alpha_1$. This, together with Theorem 1 implies that $\alpha_1.w$ is also a partially feasible trace from s. Thus, w is a partially feasible trace from $\alpha_1(s)$.

   Again, by condition (1), some action $\alpha_2 \in WI_{\alpha_1(s)}^\Pi(w)$ is explored from $\alpha_1(s)$ in S_R. Continuing in this way, we have two cases:

   - There is a sequence of actions $\alpha_1, ..., \alpha_k$ such that for each $1 \leq i \leq k-1$, w is a partially feasible trace from $\alpha_{i-1}(...(\alpha_1(s)))$ with $\alpha_i \notin w$, and $\alpha_k \in w$, and for each $1 \leq i \leq k$, $\alpha_i \in WI_{\alpha_{i-1}(...(\alpha_1(s)))}^\Pi(w)$. Now, we can extend the reasoning in case 1 of the proof (with α_k being in a similar position as α in case 1): we have a trace w'' such that S_R contains the trace $\alpha_1...\alpha_k.w''$ from s such that $\alpha_1...\alpha_k.(w\backslash\alpha_k) \sqsubseteq_s \alpha_1...\alpha_k.w''$. Knowing that $\alpha_1...\alpha_k.(w\backslash\alpha_k) \approx^\Pi \alpha_k.(w\backslash\alpha_k).\alpha_1...\alpha_{k-1}$ implies $\alpha_k.(w\backslash\alpha_k) \sqsubseteq_s \alpha_1...\alpha_k.w''$. Since $\alpha_k.(w\backslash\alpha_k) \approx^\Pi w$, we have that $w \sqsubseteq_s \alpha_1...\alpha_k.w''$, so we can take $\alpha_1...\alpha_k.w''$ as w' in the theorem.
   - There is an unbounded sequence of actions $\alpha_1, \alpha_2, ...$ such that for each $i = 1, 2, ...$, w is a partially feasible trace from $\alpha_{i-1}(...(\alpha_1(s)))$, $\alpha_i \notin w$ but $\alpha_i \in WI_{\alpha_{i-1}(...(\alpha_1(s)))}^\Pi(w)$. Consequently, there must be a loop in this unbounded sequence of actions (since the state space if finite) with the first action w_1 of w enabled in all states of the loop. By condition (2), w_1 must be explored from at least one state of the loop, and we are back in case 1 of the proof. □

If an error state is reachable in the concrete state space from the initial state with trace w, w is also an abstract trace from the abstract initial state s_0 leading to an abstract error state, since the abstract state space is an over-approximation of the concrete state space. As w is a feasible trace, it is also partially feasible for any subset of variables. As a consequence, Theorem 2 can be used for w: there is an abstract trace w' in the reduced abstract state space with $w \sqsubseteq_{s_0} w'$. Since error locations are sinks in the CFA, any state reachable from an error state is also an error state, so w' also reaches an abstract error state. Note that this w' is not necessarily feasible, only partially feasible for the variables in the precision. However, in such a case, the refinement step of the CEGAR algorithm will realize that w' is not feasible, and it will refine the abstraction.

3.3 Static Abstraction-Aware Partial Order Reduction Algorithm

Several algorithms have been presented in the literature for partial order reduction [51, 46, 1]. There are two main approaches: *static* and *dynamic* partial order reduction [13]. In the static version, the model (i.e., the CFA of the program) is analyzed, and the reduced state space is generated based on this static analysis. The dynamic approach constructs the reduced state space during the model checking. Although abstraction-aware partial order reduction uses on-the-fly information about the current abstraction, its core partial order reduction algorithm can be implemented in a static fashion.

Algorithm 1: Calculating a Source Set from s with Π

```
Input: $s, \Pi$
Output: $P$                                      /* Source set in $s$ */
1 $P \leftarrow \{\alpha\}$ for some $\alpha \in enabled(s)$
2 while any new item is added to $P$ do
3     $P \leftarrow P \cup \{\alpha \ : \ \alpha \in enabled(s) \setminus P, \ \exists \beta \in future\_actions(s, \alpha), \exists \gamma \in$
        $P$ with $(\beta, \gamma) \in D_\Pi\}$
4 end
```

Though my abstraction-aware extension can be applied to any partial order reduction algorithm, dynamic approaches tend to have much more complex formulations [2], so I will restrict myself to the presentation of a static approach in this paper. My static source set-based abstraction-aware partial order reduction algorithm is similar to Overman's algorithm [75, 51]. Before presenting the algorithm that calculates source sets, the following notions are introduced (similar concepts can be found in [2]):

Definition 9 (May-enabled action in a state). An action α is *may-enabled* in a state s, if $\alpha \in enabled(s)$ or α can become enabled after a sequence of transitions from processes other than p_α. ▪

An action $\alpha \notin enabled(s)$ can be *may-enabled* in a state s when the source location of α is $s(p_\alpha)$, but the guard condition of α evaluates to false in s.[3]

Definition 10 (Future actions). For an $\alpha \in enabled(s)$, $future\_actions(s, \alpha)$ is a set of actions: $\beta \in future\_actions(s, \alpha)$ iff there is a transition sequence $w = w_1...w_n$ from s, where $w_n = \beta$, and the first action β_0 of p_β in w is either α or $\beta_0 \notin enabled(s)$. ▪

The condition on β_0 in the definition of *future_actions* implies that β_0 is may-enabled in s. If β is in the same process as α, then $\beta_0 = \alpha$. Otherwise, β_0 is not enabled in s.

We can easily compute an over-approximation of $future\_actions(s, \alpha)$ by analyzing the static model of the program. Initially, the actions of process p_α are collected with a graph search of the CFA of p_α. An action $\beta \notin enabled(s)$ from another process that is may-enabled in s can be enabled by an action γ reached in the CFA of p_α (when γ writes a variable that β uses in its guard condition). Then, $future\_actions(s, \beta)$ is computed recursively to collect more future actions.

With the help of *future_actions*, we can compute source sets in a state. The current state s and the precision Π of the current abstraction are the inputs of Algorithm 1. Initially, any enabled action (or practically all enabled actions of a single process) is put in the source set(-to-be) P. As long as any new action is added to P, the following is repeated: $future\_actions(s, \alpha)$ is calculated for each $\alpha \in enabled(s) \setminus P$. If there is any action $\beta \in future\_actions(s, \alpha)$ that is dependent with an action $\gamma \in P$, α is added to P.

Before proving the correctness of Algorithm 1, I note the following property of software. Branches in a program are defined in an if-elseif-else manner, that is the execution can proceed on a branch independent of the variable assignment.

Property 1 *For any location l of a CFA the disjunction of guard conditions of all outgoing actions from l is* true.[4]

[3]If processes can be created or terminated dynamically, actions can be may-enabled in other ways, too; e.g., first actions of processes and join operations.

[4]An action without a guard is technically an action whose guard is *true*.

Theorem 3. A set P returned by Algorithm 1 for a state s and precision Π is a source set in s for $W_\Pi(s)$. ▪

Proof. Let us check the definition of source sets, that is, for each trace from s, one of its weak initials is in P. Let $w = w_1 ... w_n \in W_\Pi(s)$ be a partially feasible abstract trace from s. We have two cases:

1. $\exists w_i \in P$ for some $1 \leq i \leq n$ (that is w contains an action from P). Let $w_f \in P$ be the first occurrence of an element of P in w: $w_j \notin P$ for $1 \leq j < f$.

   Then, $(w_j, w_f) \notin D_\Pi$. To see this, assume the opposite: there is a w_d dependent with w_f based on D_Π, and $d < f$. Since w_f is the first action from P in w, w_d can be reached from s with actions from processes that do not have actions in P. That is, $w_d \in \mathit{future\_actions}(s, \alpha)$ for some $\alpha \in \mathit{enabled}(s) \setminus P$. In this case, Algorithm 1 would have added α to P in line 3 (with w_d as β, w_f as γ, and α as α using the notation of the algorithm). This did not happen, so our indirect supposition is wrong.

   Since w_f is independent with all actions preceding w_f in w, $w_f.(w \setminus w_f) \approx^\Pi w$ which means by definition that $w_f \in \mathit{WI}_s^\Pi(w)$. So one of the weak initials of w is in P, indeed.

2. $\nexists w_i \in P$ for $1 \leq i \leq n$.

   This supposition implies that all actions in w are independent with all actions in P based on D_Π. Assume the opposite: there is a w_d for some $1 \leq d \leq n$, and $\gamma \in P$ such that $(w_d, \gamma) \in D_\Pi$. Reasoning is similar to case 1: since $\nexists w_i \in P$, $w_d \in \mathit{future\_actions}(s, \alpha)$ for some $\alpha \in \mathit{enabled}(s) \setminus P$. Then, Algorithm 1 would have added α to P which did not happen.

   Since all actions in w are independent with all actions in P, there is at least one $\alpha \in P$ that is a weak initial of w. For this, take a partial concretization $c_0, ..., c_n$ of w from s ($c_0 \models s$). Take any action $\delta \in P$: if $\delta \in \mathit{enabled}(c_0)$, let $\alpha = \delta$. Otherwise, there is a $\delta' \in \mathit{enabled}(c_0)$ of process p_δ (starting from the same CFA location as δ) based on Property 1. Furthermore, $\delta' \in P$ since $(\delta', \delta) \in D_\Pi$ (as they belong to the same process) and $\delta' \in \mathit{future\_actions}(s, \delta')$, so Algorithm 1 adds δ' to P. As a consequence, δ' is independent with all actions in w. In this case, let $\alpha = \delta'$. Since $\alpha \in \mathit{enabled}(c_0)$, and $(\alpha, w_i) \notin D_\Pi$, we can infer that $\alpha.w \approx^\Pi w.\alpha$, and $w.\alpha$ is partially feasible from s based on Lemma 2. This means by definition that $\alpha \in \mathit{WI}_s^\Pi(w)$, indeed. So one of the weak initials of w is in P, again. □

3.4 Experiments

In this section, I evaluate the efficiency of my algorithmic contributions presented in Section 3.3. First, I introduce the plans of the experiment in Section 3.4.1, along with the research questions I aim to answer in Section 3.4.1.1, and the technical configuration details in Section 3.4.1.2. Then, I present and discuss the results of the experiment in Section 3.4.2 with respect to the research questions. Finally, I outline the conclusions of my experiments in Section 3.4.3, and discuss the potential threats to their validity in Section 3.4.4.

3.4.1 Experiment Design

The goal of my experiment is to evaluate the performance of the *static abstraction-aware partial order reduction* algorithm presented in Section 3.3. To facilitate the experimentation, I implemented both a traditional static partial order reduction approach (later *SPOR*), as well as the *abstraction-aware* static partial order reduction technique introduced in this paper (later *AASPOR*). In pursuit of a fair comparison among the algorithms, both implementations are extensions of the THETA [83] verification framework, which had prior support for multi-threaded C programs (later *NOPOR*) [15]. In my experiments, I executed different *configurations* of THETA over a set of *input programs* written in C.

3.4.1.1 Research Questions

To evaluate the presented approach, I aim to answer the following research questions.

RQ3.1 How does the performance of *AASPOR* compare to *SPOR* over the *explicit* abstract domain?

RQ3.2 How does the performance of *AASPOR* compare to *SPOR* over the *predicate* abstract domain?

RQ3.3 How does the practical performance compare to the theoretically exponential gain over the program family introduced in Figure 3.2?

3.4.1.2 Experimental Configuration

I used the subset of the concurrency safety benchmark suite[5] of SV-COMP [19] that is parsable by THETA for RQ3.1-RQ3.2, and a direct implementation of Figure 3.2 for RQ3.3 with $N := 2^{0 \leq i \leq 8}$. The configurations were executed on virtual machines with Intel Core (Haswell) processors, 3 dedicated CPU cores were allocated to each task. Experiments regarding RQ3.3 were executed with 1800 seconds of timeout, while all others used 900 seconds as their time limit. I used a *sequence interpolation*-based refinement strategy with depth-first search and thread-safe large-block encoding [57]. For the predicate domain I used *atoms* as the basis of predicate splitting. For the explicit domain I used a maximum number of enumerated successor states (*maxenum*) of 1 to preserve action-determinism, which is required from the abstract state space.

3.4.2 Experimental Results

I executed 4 different configurations of THETA on the SV-COMP benchmark suite seen in Table 3.1. Out of the 605 tasks successfully parsed by THETA and supported by the analysis, a common subset of 334 tasks were successfully solved by the *EXPL* configurations and 357 tasks by the *PRED* configurations within the time limit. No configuration reported any wrong results. Table 3.1 shows the number of successfully solved tasks and the sum of CPU time over the commonly solved tasks per abstract domain, as well as the average explored transition count over the common subsets.

Based on the results, utilization of *AASPOR* as opposed to *SPOR* reduced both the verification time and the number of expanded transitions, and managed to solve more

[5]gitlab.com/sosy-lab/benchmarking/sv-benchmarks/-/commit/2fa025c8

domain	por	explored actions	CPU time	solved tasks
EXPL	**SPOR**	15854	5916s	338
	AASPOR	12963	5516s	344
PRED	**SPOR**	11625	34982s	365
	AASPOR	10453	33850s	369

Table 3.1: Different metrics of the evaluation

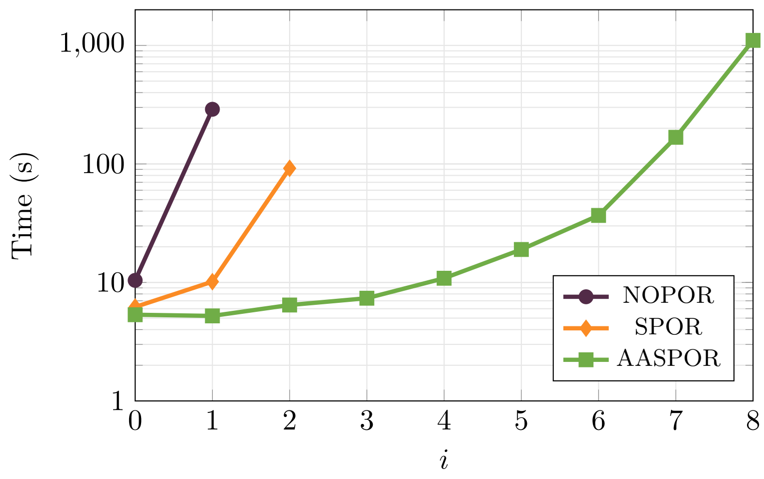


Figure 3.6: Execution time given i for $N := 2^i$ in Figure 3.2.

tasks. In both configuration-pairs consisting of an *AASPOR* and a respective *SPOR* configuration, *AASPOR* outperformed the traditional *SPOR* algorithm. While the number of explored transition is significantly reduced, the overall CPU time and the number of solved tasks are only marginally affected.

RQ3.1 In the *predicate* abstract domain, by using *AASPOR*, the sum of CPU time utilization decreased by 6.77%, and the number of expanded transitions by 18.23%. The number of solved tasks increased by 6.

RQ3.2 In the *explicit* abstract domain, by using *AASPOR*, the sum of CPU time utilization decreased by 3.23%, and the number of expanded transitions by 10.08%.

In addition, I executed three configurations of THETA on the program family in Figure 3.2, as seen in Figure 3.6. I used the *predicate* abstract domain with initial predicates extracted from the program (*allassumes*).

RQ3.3 While *NOPOR* only solved 2 tasks ($N = 1, N = 2$) within the time limit, and *SPOR* solved 1 additional task $N = 4$, the *AASPOR* configuration solved all 9 configurations, up to $N = 256$. Indeed, my abstraction-based POR algorithm scales much better on this task.

3.4.3 Evaluation Summary

As demonstrated by the data in Figure 3.6, there are certain scenarios where the utilization of *AASPOR* may lead to vastly improved verification performance (see RQ3.3). In contrast, in the subset of the benchmark suite of SV-COMP parsable by Theta, this advantage is less pronounced, presumably due to the lack of patterns where an abstraction-based verification algorithm could show its strengths. However, the significantly fewer explored transitions, and the decreased execution time of solved tasks shows that the presented approach is confidently outperforming its baseline (RQ3.1-RQ3.2).

3.4.4 Threats to Validity

The following factors may influence the validity of the experiments.

Internal Validity. Consistency and accuracy of the experiments were insured by using the BenchExec framework [28]. Memory consumption statistics may deviate between executions due to the managed nature of the Java virtual machine, therefore such metrics are not used. CPU time and therefore solved tasks may be influenced by external factors such as other processes or environmental temperature fluctuations, therefore minute differences are disregarded.

External Validity. The results of the experiments are at risk of not being generalizable due to the relatively low number of input tasks. As mentioned in Section 3.4.3, the lack of certain patterns in the benchmark suite leads to the diminished advantage of the *AASPOR* algorithm. However, the SV-COMP benchmark suite is the *de-facto* standard for academic verification tool development, and immense work would be necessary to extend the suite with further real-life examples.

Construct Validity. In order to corroborate that the right metrics are used in the evaluation of the experiments, I considered both the user-facing and the backend-related interactions of the verification tool. The number of solved tasks, and the CPU time necessary for the solution both directly affect the user's ability to verify programs at hand, and the decrease in expanded transitions will influence the number of solver invocations, reducing the load on the entire system. Therefore, these metrics accurately represent the expected outcomes of the executions.

3.5 Related Work

Partial order reduction has been a field of active research in the last decades [84, 51, 76, 13, 46, 1]. Early POR methods [84, 13, 51] approximated the conflicts between actions statically. Later, a depth-first search manner dynamic partial order reduction (DPOR) algorithm was introduced [46], where dependence between actions is decided dynamically during the state space exploration looking at the exploration stack. Source DPOR is a dynamic POR algorithm [1] whose extension, Optimal DPOR [1] is proven to be optimal in the number of explored interleavings. Many optimizations exist for determining dependency where information is retrieved from the state space search context: in these works, actions are considered dependent only in certain states under certain conditions [6, 10, 86], although these conditions are typically not related to information about the applied abstraction.

Several works use POR in the context of an abstraction-based verification algorithm [21, 85, 80, 59, 70, 79, 53]. However, most of them do not take advantage of using information about the current abstraction in order to increase the reducing effect of POR, whereas this is the key concept of my proposed approach.

CPAchecker is a program verification framework that supports several analysis techniques, including Counterexample-Guided Abstraction Refinement (CEGAR) and POR [21]. However, the POR algorithm implemented in CPAchecker is relatively simple: only thread-local operations of different threads are considered independent (where an operation is global if it accesses a global memory location and thread-local otherwise). That is, the application of POR is orthogonal to CEGAR in CPAchecker.

In the works of *Su et al.* [80], *Kroening et al.* [70] and *Wachter et al.* [85], the specific abstraction-based verification algorithm Impact is combined with a POR algorithm. Although some of them use conditional dependency, their conditions are similar to the guarded independence relation described by *Wang et al.* [86], and they do not exploit information about the applied abstraction to reduce dependency. *Kroening et al.* [70] also discuss the necessary extensions for DPOR algorithms when the abstraction-based algorithm uses *covering*. Covering is applied in abstract state space exploration when an abstract state is reached that is over-approximated by another abstract state reached earlier during the exploration. The exploration stops at these states as all possible behavior is already explored from the other (more general) state. Combining this technique with the depth-first style DPOR algorithms needs extra consideration [70]. *Hansen et al.* [59] use POR for the abstraction-based verification of timed automata. Though they use information about the current abstraction, they define relations such that one order of execution simulates the other one. My case is more general: any execution order can be selected for exploration based on my approach and none of the executions have to be the over-approximation of the other.

Several works realize that even the traditional dependency relation is not a valid dependency relation [33, 79]. The combination of POR and abstraction in the work of *Cimatti et al.* [33] is specific to the Explicit Scheduler and Symbolic Threads (ESST) algorithm, while my approach is general. In my general setting, two syntactically independent actions may disable each other in the abstract state space (see Figure 3.4 in Example 6) whereas they implicitly assume in their proof that such situation cannot happen. *Hansen* [58] also considers the application of partial order reduction in non-deterministic abstract state spaces. However, the work focuses on the challenges posed by non-determinism, and not the generalization of the commutativity relation.

A central element of my work is the idea of abstract commutativity relations investigated by *Farzan et al.* [45]. They realize that the commutativity relation can also be relaxed in an abstraction-based setting. However, their theoretical analysis focuses on the properties of abstract commutativity relations and their combinations, and they do not embed the commutativity checking in a verification algorithm. On the other hand, my work assumes an abstract state space exploration and a partial order reduction algorithm (this assumption is still a general partial order reduction in a general abstraction-based verification algorithm). Thus, my work faces the challenges and proposes a solution to the application of abstract commutativity in verification algorithms, constituting the main contribution compared to the work of *Farzan et al.* [45].

Chapter 4

Abstract Data-Flow-Based Statement Reduction

Various techniques have been developed to tackle the state space explosion problem and the great number of possible thread interleavings in concurrent programs. Abstraction reduces the size of the state space by ignoring some details of the original problem. In Chapter 2, I presented Counterexample-guided abstraction refinement (CEGAR) that iteratively refines the abstraction until the desired property can be verified. Other abstraction-based techniques like the cone-of-influence (COI) reduction or program slicing eliminate model elements irrelevant to the verified property [18, 60].

Existing cone-of-influence and program slicing techniques choose eliminable variables or statements using static data-flow analysis based on the control-flow graph of the program [18, 60]. In multi-threaded programs, this kind of elimination is often ineffective due to the communication between threads and the many possible thread interleavings. To address this, I propose an algorithm that can eliminate statements based on the current local states of concurrent threads. Whereas the COI reduction simplifies the model by eliminating completely redundant variables (redundant in all thread contexts) regarding the verified property [18], my approach identifies and simplifies statements on-the-fly that are redundant in the current state of concurrent threads with respect to the verified property. Thus, my method is more fine-grained: it can eliminate statements in certain contexts even if the statement cannot be completely ignored. This is particularly useful when a statement is relevant in one interleaving of threads while it is redundant in another: we can still eliminate it in the interleaving where it is redundant. As my algorithm takes its main advantage from the local states and interleaving of concurrent threads, I focus my presentation on concurrent programs. At the same time, it can also be used for sequential programs, though losing its advantage over existing techniques in this case.

As an example, take the program with two threads from Figure 4.1. Let us take a state s from the state space of the program where process p_2 has executed the statement `y := x` previously (i.e., this statement can be found on the path from the initial state to s). Observe that the value of x cannot be read by any statement of any thread reachable from s in the state space. Thus, it is unnecessary to evaluate `x := 1` or `x := 0` after s. My algorithm detects such situations and eliminates such statements. Note that a traditional COI algorithm could not eliminate the statement `x := 1` as its result may be used.

My statement simplification method is motivated by the considerable runtime of calculating successor states in SMT-based state space exploration [27, 57]. I strive to dynamically identify as many redundant statements as possible in the current exploration context. For

```
   Process p1      |    Process p2
                   |
x := 1             |  y := x
y := 1             |  x := 0
assert(y=1)        |
```

Figure 4.1: Running example (pseudocode of the example in Figure 2.1)

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 (i.e., calculating the successor of the current state with respect to this statement), I check using the data-flow graph whether any other statement can use the result of the statement. I target reachability properties; thus, I am interested in whether the result of the statement is used transitively by a conditional statement, as only conditionals can directly influence whether some marked error locations in the model are reachable. This can be decided by a traversal of the data-flow graph. Redundant statements are eliminated, sparing the time of successor state calculation in such cases. I formulate my algorithm for abstract state space exploration and exploit information about the current abstraction to reduce the number of edges in the data-flow graph.

To further motivate my approach, it is also possible to achieve exponential gains in the number of evaluated statements by using my novel algorithm. Consider the same example again from Figure 3.2 with $2N+1$ processes. The safety of the program can be proven with abstraction by only tracking the predicates $z \bmod 2 = 0$ and $x = 0$ about our variables. Processes $p_1 - p_{2N}$ have $2N!$ interleavings not considering the statements of the loop of p_0. However, my algorithm notices that we do not track any information about y, so the results of statements writing y are not used in the current abstraction: thus, my approach eliminates all of these statements ($2N! * 2N$ statements exactly). My approach also enables any standard partial order reduction algorithm [3, 1] to reduce the number of explored interleavings exponentially that otherwise would have to explore all interleavings. My algorithm achieves this by eliminating the source of dependency between statements. The abstraction-based partial order reduction presented in Chapter 3 is not even required for this; the proposed algorithm in this chapter can also handle the problem.

Contributions. I take the idea of cone-of-influence one step further by deciding on-the-fly during state space exploration 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 (namely, CEGAR). I implemented and evaluated my algorithm in the abstraction-based model checking tool Theta [83].

4.1 Statement Reduction during Dynamic Analysis

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

4.1.1 Data-Flow Graph with Precision

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

Definition 11. Let α, β be actions, and Π be the precision of the abstraction. We say that β *observes* α with precision Π if $\mathit{written}(\alpha) \cap \mathit{read}(\beta) \cap \mathit{vars}(\Pi) \neq \emptyset$.

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

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

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

Definition 12. An abstract data-flow graph is a tuple $G = (A, D, I, \Pi)$ where:

- A is the set of actions of the program (the nodes of the data-flow graph),
- $D \subseteq A \times A$ is the set of direct observation edges: $(\alpha, \beta) \in D$ if β observes α with Π, $p_\alpha = p_\beta$, and β is reachable from α in the CFA of their process,
- $I \subseteq A \times A$ is the set of indirect observation edges: $(\alpha, \beta) \in I$ if β observes α with Π and $p_\alpha \neq p_\beta$.[1] ▪

The data-flow graph can be precomputed for the 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, I simply iterate over the actions of all other processes and add an indirect observation edge wherever needed. So the data-flow graph can be built in polynomial (quadratic) time in the number of CFA edges.

Example 8. *Let us have the simple program from Figure 4.1: its CFA is shown in Figure 4.2. The figure also shows two abstract data-flow graphs: the upper one with a precision where some information is tracked about both x and y ($\mathit{vars}(\Pi) = \{x, y\}$); below, we have no information about x ($\mathit{vars}(\Pi) = \{y\}$). Therefore, no edges start from actions assigning x in the second graph. Solid edges are direct observation edges, dashed edges represent inter-process observations.*

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

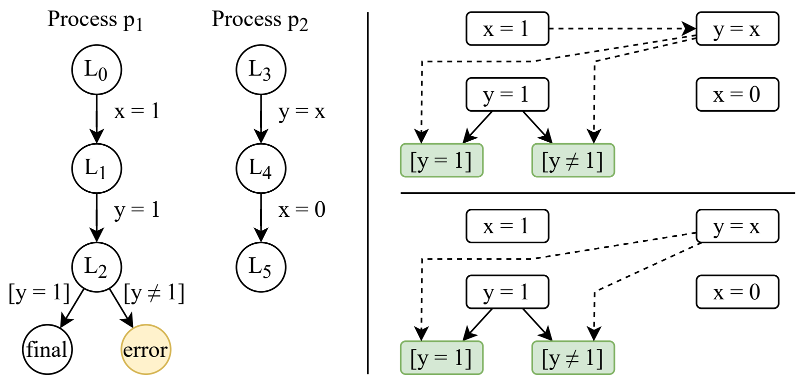


Figure 4.2: CFA of two processes and data-flow graphs with different precisions

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

Let Π be the precision of the abstraction, and $G = (A, D, I, \Pi)$ the computed abstract data-flow graph. Let s be a state, $\alpha \in \mathit{outgoing}(s)$: our goal is to decide whether α can be transitively observed later during the program execution in a relevant way. I target reachability properties, so relevant actions are the actions with guard conditions since 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 4.2. Thus, the evaluation of α can be skipped if there is no trace from the current state where a *real observer* transitively observes α[2]. This can be decided using the data-flow graph. To formalize this idea, I introduce the following definitions:

Definition 13. 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 an abstract trace w in the abstract state space from s with $\alpha \in w$ and $p_\alpha = p$. ▪

Intuitively, if α is transitively observed by an action β in a trace starting from the current state s, then there is a path in the data-flow graph from α to β only passing through graph nodes (actions) which can still be reached from s by one of the processes. Formally, I define conditions for the enabledness of the data-flow graph edges:

Definition 14. Let s be an abstract state, and $G = (A, D, I, \Pi)$ an abstract data-flow graph.

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

Rephrasing the previous paragraph: if there is a trace from s where α is transitively observed by an action β, then there is a sequence $\alpha_1, ..., \alpha_n$ such that $\alpha_1 = \alpha$, $\alpha_n = \beta$, $(\alpha_i, \alpha_{i+1}) \in D \cup I$ for each $1 \leq i < n$, and (α_i, α_{i+1}) is enabled in s. Using the definition, deciding the enabledness of a data-flow graph edge amounts to answering reachability questions in the state space (see Definition 13) which is also the original purpose of the

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

verification of reachability properties: seemingly, the problem has not become easier. However, $reachable(s, p)$ can be over-approximated by checking reachability in the CFA of the program[3].

Example 9. *Let us continue our example from Figure 4.2 with a precision* Π *such that* $vars(\Pi) = \{x, y\}$ *(i.e., the upper data-flow graph in Figure 4.2). In the initial state where both processes are in their initial locations (*L_0 *and* L_3*), all actions may be reachable in the future by one of the processes since we over-approximate reachability in the state space by reachability in the CFA. Thus, all data-flow graph edges are enabled, so there is a path of enabled data-flow graph edges from both outgoing actions* $x = 1$ *and* $y = x$ *to a real observer (e.g., to* $[y = 1]$*). However, if we have a state where the processes are in locations* L_0 *and* L_4*, then* $y = x$ *can never be reached from this state, so all data-flow graph edges leaving or targeting* $y = x$ *are disabled. That is, there is no path from* $x = 1$ *and* $x = 0$ *to a real observer in this state, so these actions are not transitively observed by a real observer, and thus, do not have to be evaluated from this state.*

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

Using an adequate data structure, edge enabledness in the data-flow graph can be over-approximated in constant time using CFA reachability information. For indirect edges, CFA reachability information can either be stored in a 2D array (with constant time indexing) or a more memory-efficient, but slightly more over-approximating approach based on strongly connected components can be used (by storing the CFA strongly connected component id number for each CFA edge and comparing these ids on-the-fly). All direct observation edges reachable in G from an action $\alpha \in outgoing(s)$ are enabled based on Definition 14.

For each action $\alpha \in outgoing(s)$, we traverse the data-flow graph from α in the way introduced above. If a real observer is reached, then the value produced by α is used (or at least may be used, c.f., the applied over-approximations), so we evaluate α properly to calculate the successor states $\alpha(s)$. However, if no real observer is reached, then the value is unused, making α unnecessary to evaluate. Instead, the successor state s' can be the state differing from the current state s only in the location of the process of α: $s'(p_\alpha)$ is the target location of α. The following method is used to determine the successor states: for the single variable $v \in written(\alpha)$, if $v \in vars(\Pi)$, the original statement assigning a new value to v is replaced by a *havoc* v statement; if $v \notin vars(\Pi)$, the original statement is removed (more precisely replaced with a *no operation* statement that has no effect). Using *havoc* on the variables tracked in the current abstraction is necessary for the refinement step of CEGAR (see Section 4.1.3 for more details).

Algorithm 2 summarizes the presented method of statement simplification based on dynamic data-flow analysis. Theorem 4 proves that using Algorithm 2 for state space exploration yields correct results, that is, it reaches an error state whenever an error state is reachable with a feasible trace in the original state space. By original state space, I mean the abstract state space explored without the introduced statement simplification (i.e., for each $\alpha \in outgoing(s)$, the successor states $\alpha(s)$ are all explored).

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

Algorithm 2: State Space Exploration with Statement Simplification

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

Theorem 4. Algorithm 2 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 also a feasible abstract error trace from some $s' \in successors$. We have the following cases:

1. α is transitively observed by a real observer.

   Then α is not simplified, so $successors = \alpha(s)$. Naturally, if there is a feasible abstract error trace from s in the form $\alpha.w$, then w is a feasible abstract error trace from at least one element of $successors = \alpha(s)$.

2. α is not observed transitively by a real observer, and $v \notin vars(\Pi)$ for the single item $v \in written(\alpha)$.[4]

   In this case, α practically has no effect since no information is tracked about v in the current abstraction. So an assignment of v only performs a location update for p_α. This is exactly how s' defined in lines 15-16, so $successors = \alpha(s)$ in this case, as well. Similarly to case 1, there is a feasible abstract error from at least one element of $successors = \alpha(s)$.

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

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

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

As the property proven above is preserved in each exploration step, it follows by induction that if a feasible abstract error trace is available from the initial state, then there is a feasible abstract error trace in the state space explored by Algorithm 2, as well, which proves the theorem. $\square$

4.1.3 Statement Simplification with CEGAR

My proposed algorithm can be used by the abstractor of CEGAR for abstract state space exploration (see Section 2.3.2 for an introduction of CEGAR). However, it is important that a potential counterexample provided to the refiner must contain the original actions even if my algorithm simplified them during the state space exploration. For a reason, consider a program with a single process which is *Process* p_1 from Figure 4.2, but let the action from L_1 to L_2 be $y = x$. Let our precision only track information about x: $vars(\Pi) = \{x\}$. An error state is reachable in the abstract state space with the clearly spurious trace ($x = 1$, $y = x$, $[y \neq 1]$). However, my algorithm simplifies $x = 1$ and $y = x$ since they are not observed by a conditional action with this precision. If the refiner only sees the simplified actions in the trace, i.e., (*havoc* x, *no operation*, $[y \neq 1]$), it cannot spot the contradiction. Concluding that the counterexample is feasible, it would give a wrong *unsafe* verdict.

My algorithm presented previously can also be used in verification algorithms other than CEGAR. In such a case, it may be possible to drop lines 11-14 of Algorithm 2 and always use lines 15-17 to define the successor state when α is not observed transitively by a real observer. However, I focus on CEGAR as the base algorithm in this work, making the *havoc* necessary when the assignment of a variable in the precision is simplified. If the successor state is defined as in lines 15-16 instead of using a *havoc*, the refiner may see a contradiction. For example, assume that the value of variable x is explicitly tracked in a CEGAR iteration, and the abstractor finds a counterexample trace. The trace contains a state s where $x = 0$, and the next action α assigns 1 to x. However, my algorithm noticed that α cannot be transitively observed by any real observer, so it skipped the evaluation of the statement of α. Based on lines 15-16 of the algorithm, the value of x would be the same (namely 0) in the state s' after α in the trace. Then, the refiner finds a contradiction here as the value of x cannot be 0 after an action that assigns 1 to x (we have seen in the previous paragraph that the counterexample must contain the original actions). On the other hand, the precision could not be refined based on this misleading contradiction, and the CEGAR algorithm could get stuck in endless iterations.

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

It is worth mentioning that my algorithm cannot introduce new spurious counterexamples and degrade performance this way. Intuitively, guard conditions cannot get enabled as a side-effect of my algorithm since Algorithm 2 only simplifies statements that are not observed transitively by any conditional statement. Thus, the evaluation of guard conditions is not affected, so new (spurious) counterexamples cannot emerge. As for originally feasible counterexamples, they remain feasible with my algorithm as feasible traces are always available in the abstract state space explored by our algorithm based on the proof of Theorem 4.

4.2 Experimental Evaluation

In this section, I evaluate the efficiency of my algorithmic contributions presented in this chapter. The goal of my experiment is to evaluate the performance of my proposed dynamic statement simplification algorithm. I refer to my novel algorithm as a dynamic cone-of-influence-based statement simplification algorithm (or `DCOI` for short) in my experiments. I compare my algorithm to a baseline using static cone-of-influence (`SCOI`) to see how much further reduction is achieved (that is when `DCOI` and `SCOI` are both enabled). Since `DCOI` can do the job of `SCOI` so to say, it is also meaningful to investigate the performance with only `DCOI` while `SCOI` is disabled. I am interested in the effects of my algorithm in different abstract domains, therefore I investigate the effect of our proposed algorithm in two abstract domains frequently used in state-of-the-art verification tools [20]: explicit-value abstraction (later `EXPL`) [23] and (Cartesian) predicate abstraction (later `PRED`) [16].

I implemented [5] my algorithm as an open-source extension of the THETA verification tool [83] which already had a built-in CEGAR algorithm and a static cone-of-influence preprocessing step, and had prior support for multi-threaded C programs including a partial order reduction algorithm [15]. I also compare my results to other state-of-the-art verifiers using abstract state space exploration.

4.2.1 Research Questions

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

RQ4.1 What proportion of statements can be simplified or completely eliminated using the proposed algorithm?

RQ4.2 How is the time of successor state calculation affected by my algorithm?

RQ4.3 How is the overall verification performance affected by the algorithm?

RQ4.4 What practical performance improvement can we observe on programs where theoretically exponential gain is expected?

[5] https://github.com/csanadtelbisz/theta/commit/f2f1f8d

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

Table 4.1: Different metrics of the evaluation

4.2.2 Experimental Configuration

In my experiments, I executed different configurations of THETA over a set of input programs written in C from the concurrency safety reachability category benchmark suite[6] of SV-COMP [20] (715 tasks) that is parsable by THETA (602) for RQ4.1-RQ4.3 and a direct implementation of Figure 3.2 for RQ4.4 with $N = 2^{0 \leq i \leq 7}$. I executed 6 configurations on the SV-COMP benchmarks: both abstract domains (`EXPL`, `PRED`) with the three different cone-of-influence methods (`SCOI`, `SCOI+DCOI`, `DCOI`). The benchmark tests were executed on virtual machines with Intel Core (Haswell) processors, 2 dedicated CPU cores were allocated to each task. Each verification task had a time limit of 900 seconds (1800 seconds for RQ4.4) and a memory limit of 15GB. I used a sequence interpolation-based refinement strategy for the refinement step of CEGAR, and depth-first state space exploration with thread-safe large-block encoding and an abstraction-based partial order reduction algorithm [15] in the abstraction phase. I used atoms as the basis of predicate splitting for the predicate domain; and I used a maximum number of enumerated successor states (maxenum) of 1 for the explicit domain [57]. My backend SMT solver was Z3. For the exponential gain program, I used the predicate abstract domain with an initial precision obtained by extracting branching conditions from the program. I also used a partial order reduction algorithm [3] after applying `DCOI` (the same POR algorithm is also used when `DCOI` is disabled).

4.2.3 Experiment Results

In the concurrency safety benchmark suite, THETA was able to parse 602 programs. No configuration provided wrong results. Table 4.1 shows the results for different metrics aggregated by configuration. For a fair comparison, the aggregated values are calculated over the common subset of correctly solved tasks by abstract domain: a common subset of 332 tasks was solved with the configurations using explicit-value abstraction, and 350 with predicate abstraction. The *simplified by `DCOI`* column shows the average proportion of simplified statements (including completely eliminated statements) simplified by `DCOI` compared to all statements. The *successor calculation* and *CPU time* columns are the sum of successor state calculation and CPU times of commonly solved tasks.

The results confirm the reduction potential of my algorithm: configurations using `DCOI` greatly outperform (depending on the abstract domain) the baselines without `DCOI` in terms of both successor state calculation and overall verification performance. It is also in line with my expectations that using `DCOI` without `SCOI` leads to slightly better per-

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

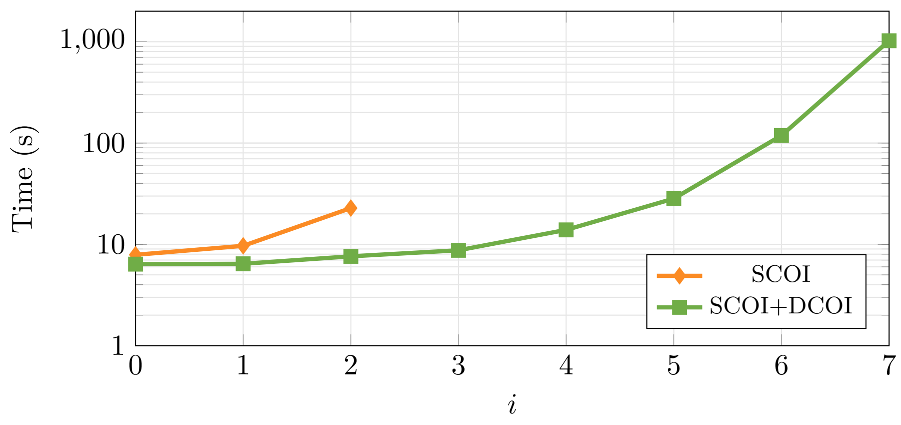


Figure 4.3: Execution time given i for $N := 2^i$ in Figure 3.2

formance since `DCOI` can also eliminate the statements removed by `SCOI` with a minor overhead while the time of `SCOI` is completely spared. Let us interpret the results by answering the research questions:

RQ4.1 `DCOI` simplifies 19.6% of all statements on average (14% completely eliminated, while 5.6% replaced by *havoc* with explicit-value abstraction; 14.4% and 5.2% respectively with predicate abstraction). This confirms the relevance of my method: a significant subset of statements is unnecessary in certain thread interleavings for verifying the given property of the program.

RQ4.2 My algorithm greatly reduces the time of successor state calculation: by 29.9% with explicit abstraction and 31.8% with predicate abstraction. A significant part of successor state calculation is taken by SMT-solvers solving SMT problems (especially when using predicate abstraction). Thus, the overall system load is significantly decreased by reducing the SMT problem solving time.

RQ4.3 Overall performance is also improved, especially for predicate abstraction: `DCOI` reduces the overall CPU time compared to the baseline by 3.7% using explicit abstraction, and by 14.6% using predicate abstraction. It was my expectation to have better improvement with predicate abstraction since it is more costly to compute which tracked predicates (or their negations) are entailed by the previous abstract state and the current action. Thus, successor state calculation takes a greater portion of the whole verification in predicate abstraction leading to a greater impact of my algorithm. The number of solved tasks is only slightly increased probably because the complexity of input tasks is not linearly increasing. The overhead of my algorithm is not huge though not completely negligible: 197 seconds and 206 seconds for `SCOI+DCOI` and `DCOI`, respectively, aggregated for all tasks with explicit-value analysis which is 3.6% and 3.8% of all CPU time. Similarly, my algorithm ran for a total of 342 and 345 seconds with predicate abstraction taking 1% of all CPU time in both cases.

RQ4.4 Even though the baseline used the same static COI and the same partial order reduction algorithm, it could only solve the three smallest tasks in the set (up to $N = 4$), whereas `DCOI` was able to verify 8 tasks, up to $N = 128$ as seen in Figure 4.3. Indeed, my proposed algorithm scales much better on this program.

Comparison with the state-of-the-art. I also compare my solution to state-of-the-art verifiers. I select the best performing verifiers from SV-COMP 2024 concurrency category [20] that use some kind of abstract state space exploration algorithm. Two state-of-the-art tools with analyses conceptually similar to my approach are CPAchecker [14] and

PIchecker [81]. By conceptually similar, I mean that they also use state space exploration of some form (unlike other tools that encode the program into an SMT formula for instance). Other successful tools in SV-COMP are either bounded model checkers (such as Dartagnan [40], Deagle [62], and CSeq [36]) that would be unfair choices for comparison with a complete model checking algorithm; or use a conceptually different trace abstraction algorithm (such as Ultimate Automizer [63], GemCutter [68], and Taipan [43]); or use some advanced algorithm or tool selection strategies without implementing own analyses (such as PeSCo [78], and Graves [71]).

I executed the two verifiers on the same SV-COMP benchmark data, on the same hardware with the same limits. CPAchecker uses a standard state space exploration technique for multi-threaded programs combined with a BDD analysis [21] and achieved to solve 346 tasks correctly[7]. PIchecker is built on CPAchecker and has multiple analyses for concurrent software [81]. One that uses CEGAR and Craig interpolation could verify 246 tasks while its main method, a BDD analysis with an elaborate partial order reduction algorithm could verify 388 tasks. My solution solving 358 tasks thus ranks second among these similar analyses. While my contribution does not bring Theta to the first place among these tools, it reduces the advantage of PIchecker. As a reference, the most solved tasks by a single tool (using conceptually different techniques) was 452 in the SV-COMP 2024 concurrency reachability category [20]. Though I only evaluated tools with conceptually similar algorithms for a fair comparison, bounded model checker tools using state space exploration could also benefit from my proposed optimization.

4.2.4 Threats to Validity

The following factors may influence the validity of my experiments.

Internal validity. I used BenchExec [28] to ensure accuracy. I ran my experiments on virtual machines in the cloud computing platform of our university. External factors such as loads on other virtual machines of the host and shared resources may have influenced the results.

External validity. The SV-COMP benchmark suite is considered a de facto standard for academic benchmarking in software verification. Theta can only parse a limited subset of SV-COMP concurrent benchmark programs which further reduces generalizability. However, there might be more redundant model elements in real-world software than in the simplified programs of the SV-COMP benchmarks, making my technique disadvantaged on the benchmark set. Thus, my algorithm may achieve greater reduction in industrial applications.

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

[7]CPAchecker also has a predicate analysis for concurrent software [26] but the algorithm is a bit dated and this analysis can only verify 159 tasks.

4.3 Related Work

Several works aim to simplify the model by eliminating redundant model elements based on data-flow analysis [18, 72, 64, 60, 44, 74]. However, these techniques only statically analyze and simplify the input model which is limited compared to my on-the-fly data-flow analysis. These static approaches have the advantage that they have to be executed only once before the state space exploration while my algorithm is performed at each successor state calculation. On the other hand, my experiments in Section 4.2 show that my algorithm does not have a significant runtime overhead, so it is worth running my algorithm several times to eliminate further statements. There are dynamic program slicing techniques as well [60, 69]. However, *dynamic* in that context means those techniques use actual input values or already discovered error traces for slicing [5]. These techniques also do not take advantage of the local states and interleaving of threads (most of them formulated for sequential programs [69]) which is the basis of my approach.

Many algorithms have been developed for model checking concurrent programs that reduce the number of explored thread interleavings such as partial order reduction or maximum causality reduction [1, 67, 4]. Some works perform dynamic data-flow analysis in various ways to improve the reduction potential of these techniques [32, 10, 67, 4], though they only use data-flow analysis to reduce the number of explored interleavings and not to simplify statements. These techniques take explored traces and discover redundant statements within these traces: they use this information to explore even less interleavings (e.g., by ignoring these statements when calculating a dependency relation [10]). The works of *Huang* [67] and *Agarwal et al.* [4] discover causality connections and build causality constraints between statements (events) of a trace and simplify these formulae by eliminating irrelevant statements which is a similar concept to my approach. However, they only use this idea to simplify these constraints, but they still completely explore traces first. So my approach could achieve further reduction in these cases as well. In other words, these works aim to reduce the size of the explored state space whereas my purpose is to accelerate the exploration of a (reduced) state space by skipping the evaluation of certain program statements. My algorithm is orthogonal to these techniques and could be applied on top of them to further improve the performance by eliminating further model elements.

Chapter 5

Verification with Partial Orders

Efficient algorithms for model checking multi-threaded programs often involve reasoning about *happens-before* relations which define a partial order of concurrent program instructions [8, 7]. The program is symbolically encoded along with some scheduling constraints based on the possible happens-before relations. The encoding is often expressed as a Satisfiability Modulo Theories (SMT) formula. The key concept of these bounded model checking approaches is to use the models (variable assignments) provided by the SMT solver to analyze possible partial orders and prevent scheduling inconsistencies that may arise in these models during the verification. A scheduling inconsistency corresponds to a cycle in the happens-before relation, since a valid execution of concurrent threads must have a linearization of instructions. Any model found by the SMT solver whose interpretation leads to a happens-before cycle does not represent a valid execution of the multi-threaded program. Inconsistencies or conflicts in the model can be converted into a conflict clause that can be used to exclude invalid program executions when looking for an execution that violates the requirement of the verification [87, 61].

Even though these techniques have gradually improved and adapted to work efficiently with SMT solvers, these methods put most of the reasoning into the SMT solver. While SMT solvers are generally optimized, leaving most calculations to a general-purpose SMT solver is not optimal in performance. SMT solvers ignorant of the domain of concurrent programs often cannot make smart enough choices at decision points when looking for a satisfying assignment of the encoded formula.

My work aims to considerably reduce the model search space of the SMT solver by constraining the encoding formula. I achieve this by analyzing the program structure and possible partial orders of program instructions, and collect possible scheduling inconsistencies, i.e., cycles that may arise in the happens-before relations. Figure 5.4 shows simple structures of partial orders that lead to a cycle in the happens-before relation, and therefore represent an invalid partial order of concurrent program events. Conflict clauses formulated from these conflicts are appended to the encoding formula. My contribution enables to greatly improve the performance and scalability of the verification. I formulate a general framework for reasoning with possible partial orders of concurrent program instructions. Then, I propose a novel approach and a specific algorithm that searches for possible conflicts of bounded size. On one hand, applying a bound keeps a limit on the size of the encoding formula, on the other hand, my evaluation shows that a great proportion of invalid behavior is already excluded with the small bound.

Most works try to reduce the size of the encoding formula [87, 61, 82]. My method follows these approaches in that certain basic scheduling constraints are omitted. On the other hand, my approach performs early computations, and preserves some useful information about partial orders. This way, I also add new elements to the formula: conflicting partial orders that cannot occur together in a valid program execution. I select these extra constraints in a way that is useful for the SMT solver which instead of putting an overhead on the decision procedure, accelerates it considerably.

While reading the literature and analyzing algorithms in published papers, I discovered a theoretical problem with a published verification algorithm described in [82]. Since the problem is complex, its presentation is technical and does not fit in the limits of this thesis. As a consequence of this discovered problem, the problematic algorithm is insufficient for model checking concurrent programs under sequential consistency. I propose a way to circumvent the limitations of the algorithm by extending with other techniques.

Contributions. This chapter presents the following contributions:

- I discovered a theoretical problem with a published algorithm as mentioned above. I formulate a sound verification algorithm in Section 5.3 by combining existing concepts. The resulting algorithm is suitable for model checking concurrent programs under sequential consistency.
- I propose an optimization to partial order-based verification by automatically discovering scheduling conflicts. The method is presented and proven to be sound in Section 5.4.
- I created an implementation of my theoretical contributions as an open-source extension of the verification tool THETA [83]. My implementation achieves the best result among state-of-the-art bounded model checkers in terms of the number of solved problems on a large set of benchmark programs.

Figure 5.1 summarizes the approach with my contributions. The verification task is encoded into an SMT formula which is given to the SMT solver. If the formula is unsatisfiable, then the program is safe. Otherwise, we have to check whether the model returned by the SMT solver represents a valid execution of concurrent instructions (my first contribution in this chapter concerns this decision). If the model given by the SMT solver represents a valid program execution indeed, then the program is unsafe. Otherwise, a conflict clause is generated which is given to the SMT solver to exclude this inconsistent model. This process is repeated until a safe or an unsafe verdict is reached. My second contribution corresponds to the *Automatic conflict finder* box: my goal is to automatically[1] discover conflicting situations and generate conflict clauses before giving the verification formula to the SMT solver.

This chapter is structured as follows: a description of further necessary preliminary knowledge is given in Section 5.1 and Section 5.2. I present the verification algorithm in Section 5.3, and the proposed optimization in Section 5.4. Finally, I evaluate the presented approaches in Section 5.5 and present the related work of the field in Section 5.6.

[1]The word *automatic* only refers to the fact that no SMT solver is needed for this approach. The decision diamond in Figure 5.1 is also automatic in the sense that it does not require human interaction.

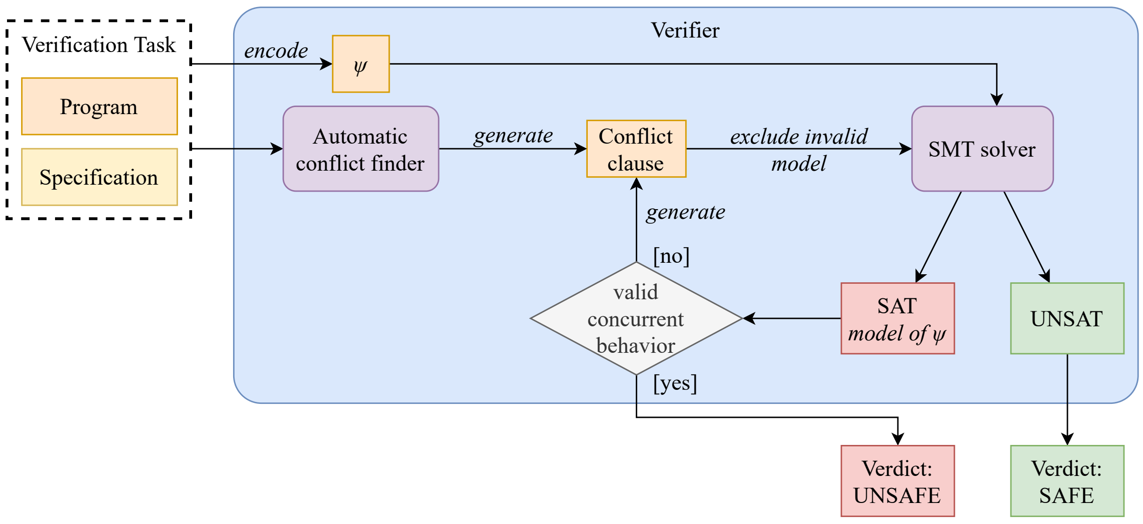


Figure 5.1: Verification with Partial Orders

5.1 Weak Memory Models

This work does not really focus on weak memory models. However, the accurate theoretical presentation of the paper requires some references to weak memory models. Therefore, we need a few words to have a basic understanding of the concept of memory models. To improve efficiency, most processors apply various kinds of memory reordering. A memory model describes what types of memory or instruction reordering is allowed for a given processor [11]. A weaker memory model allows more types of reordering. The verification problem under a specified memory model $\mathcal{M}$ asks whether there is a sequence of program instructions violating the safety requirement that conforms to $\mathcal{M}$ by respecting the allowed types of instruction reordering. For the presentation of this paper, the only relevant aspect of memory models is the following: if a sequence of instructions is a valid program execution under a memory model $\mathcal{M}_1$, the same sequence is also a valid execution under all memory models $\mathcal{M}_2$ such that $\mathcal{M}_2$ is strictly weaker than $\mathcal{M}_1$ (where strictly weaker means that all types of reordering allowed by $\mathcal{M}_1$ is also allowed by $\mathcal{M}_2$). Therefore if a program is proven safe under the (strictly) weaker memory model $\mathcal{M}_2$, it is also safe under the stronger $\mathcal{M}_1$.

The sequential consistency model does not allow any kind of memory reordering. Generally, memory models are often defined via a happens-before relation which specifies for each pair of concurrent instructions in a declarative manner if one must happen before the other or not; that is, if their order is important or not: in the latter case, they may be reordered [55]. Since my work is based on reasoning with happens-before relations, I give a detailed introduction in the next section.

5.2 Partial Orders

In this chapter, I use an event-based representation of multi-threaded programs [8]. Each memory access is represented by an *event* e characterized by the memory address (which variable or memory location is accessed), access type (*read* or *write*), and a guard condition grd_e (an event is enabled if its guard condition is true; in other words, the event is part of the program execution if its guard is true). The concurrent behavior of an execution is represented by a *happens-before* relation $\prec$ on events. Intuitively, $e_1 \prec e_2$ simply means that e_1 must precede e_2 in the program executions represented by this happens-before relation. An execution is *valid* (or consistent) if there is no cycle in $\prec$ ($\nexists e$ such that $e \prec e$) [7], in other words, if there is a possible linearization (or sequentialization) of events respecting $\prec$.

Example 10. *Take the program of Figure 5.3a. Each variable access in the program is an event: Figure 5.3b associates an index to each variable usage to have a unique reference to events. The happens-before relation relates program events as demonstrated by the edges in Figure 5.3c (ignore the different labels and colors of edges for now). For example, the event x_2 must* happen before *x_3 in all program executions represented by that specific happens-before relation. In fact, the happens-before relation in the figure does not represent any valid execution due to the cycle in the relation (the transitive closure of the relation contains a self-loop: e.g., $x_2 \prec x_2$).*

The happens-before relation can be defined by giving a list of base relations and a list of rules (or axioms) prescribing that if certain event pairs are happens-before-related, then other events must also be related [61, 88]. The most straightforward rule is transitivity (formally given in Axiom 1 later). The definition of the happens-before relation varies by memory model. In memory models defined by an acyclic happens-before relation, the conformance of a program execution to the memory model means that retrieving the base relations from the execution and applying all rules to the happens-before relation results in an acyclic happens-before relation.

The following base relations can relate the events of the program (a detailed example of partial orders is given later in Section 5.3, see Figure 5.3) [7, 61]:

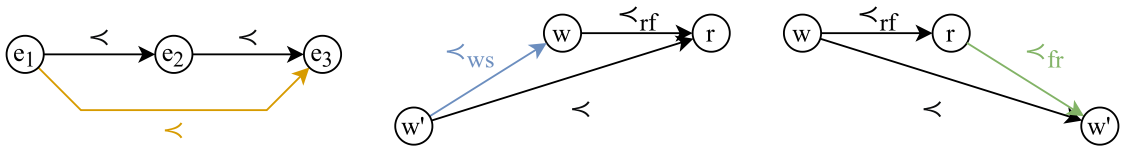
- The *program order* $\prec_{po}$ is a total order of events of the same thread. It is the order of events respecting the order of instructions in the program code[2].
- The *read-from* relation $\prec_{rf}$ relates a write event w to a read event r (written $w \prec_{rf} r$) if r reads the value written by w.
- The *coherence order* relates write events to the same variable (or memory location)[3].

The above list is only an example: there can be other base relations used in the definition of the happens-before relation, and these are not necessary either. An important property of the *rf-relation* is that if $w \prec_{rf} r$, then there cannot be another write event w' to the same memory address between w and r. Otherwise, r could not read the value written by w as it would have been overwritten by w'. Some rules (derivation rules) can be devised from this property [61, 88]:

- A *write-serialization* order relates two writes $w' \prec_{ws} w$ if there is a read r such that $w \prec_{rf} r$ and w' precedes r. In such a case, w' cannot happen between w and r, so it must precede w as well.
- A *from-read* order relates a read and a write event $r \prec_{fr} w'$ if there is a write w such that $w \prec_{rf} r$ and w precedes w'. Similarly, w' cannot happen between w and r, so it must happen after r as well.

[2]In a practical implementation where we can dynamically start and join threads, a thread creation or join event is also related to the first/last event of the thread in $\prec_{po}$ as expected.

[3]I did not associate a notation to the coherence order because I will not use it formally in this work.



(a) Transitivity **(b)** Write-Serialization **(c)** From-Read

Figure 5.2: Deriving Happens-Before Relations

While the ideas presented in this chapter can be generalized to memory models with an acyclic happens-before relation, I will focus my presentation on sequential consistency (SC) being the most basic and intuitive memory model widely used in the literature and practice [8]. In fact, I will consider a slightly weaker memory model than SC for practical reasons: weak sequential consistency (wSC) [88]. The happens-before relation of SC is defined by the program order, read-from and coherence order relations as well as the from-read and transitivity rules. The only difference in wSC is that the coherence order is replaced by the write-serialization rule. The difference may seem a minor technicality (both the coherence order and write-serialization relate write events), but the truth is wSC is generally easier to handle: deciding whether a program execution conforms to wSC can be done in polynomial time [88] while (surprising as it may seem) the same decision problem for SC is NP-complete [50]. Since my approach discovers cycles in the happens-before relation, I can use that of wSC instead of SC: any cycle in the happens-before relation of wSC can also be found in the happens-before relation of SC as SC is strictly stronger than wSC [88].

I define formally the happens-before relation of wSC and I will use this happens-before relation throughout the rest of the thesis:

Definition 15 (Happens-Before Relation $\prec$). $e_1 \prec e_2$ holds for events e_1 and e_2 if:

- $e_1 \prec_{po} e_2$, *or*
- $e_1 \prec_{rf} e_2$, *or*
- $e_1 \prec e_2$ *can be derived by a sequence of derivation steps based on the axioms Axiom 1, Axiom 2, and Axiom 3.* ▪

Axiom 1 (Transitivity Derivation) *For any events e_1, e_2, and e_3:*

$(e_1 \prec e_2) \wedge (e_2 \prec e_3) \Rightarrow (e_1 \prec e_3)$.

Axiom 2 (Write-Serialization Derivation) *For any write events w and w', and any read event r such that they all belong to the same memory address:*

$(w \prec_{rf} r) \wedge (w' \prec r) \wedge grd_{w'} \Rightarrow (w' \prec w)$.

Axiom 3 (From-Read Derivation) *For any write events w and w', and any read event r such that they all belong to the same memory address:*

$(w \prec_{rf} r) \wedge (w \prec w') \wedge grd_{w'} \Rightarrow (r \prec w')$.

I will sometimes emphasize the used derivation step by writing $\prec_{ws}$ or $\prec_{fr}$ for orders derived by Axiom 2 or Axiom 3, respectively. Figure 5.2 visually depicts the derivation rules coloring the derived partial order in each case. Program order and read-from orders come directly from the program code or from the data-flow of an execution, so they cannot be derived from other partial orders, and thus, do not need axioms.

initially: `x = y = 0`

Thread t_1	Thread t_2
`if(y==1) x=1;`	`y=x;`
`else x=1-y;`	`x=1-y;`

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

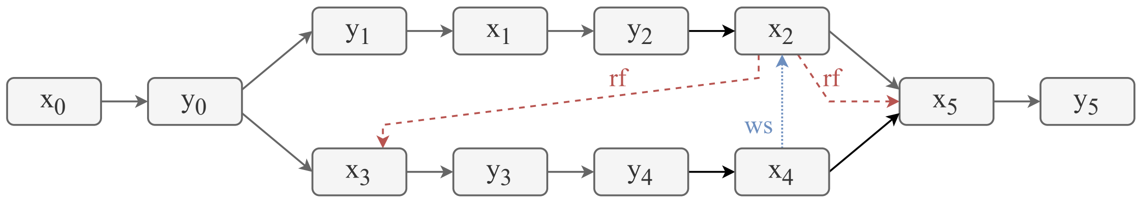
(a) Original program

initially: $x_0 = y_0 = 0$

Thread t_1	Thread t_2
if$(y_1$==$1)\ x_1$=$1;$	y_3=$x_3;$
else x_2=1-$y_2;$	x_4=1-$y_4;$

finally: assert$(x_5$==1 \|\| y_5==$1)$

(b) Indexed variable accesses



(c) Program order and some other related events

Figure 5.3: Running example

5.3 Verification with Partial Orders

The verification algorithm is a bounded model checking algorithm requiring a loop-free concurrent program and a safety property. Programs with loops can be handled by unrolling loops: complete loop unrolling can be applied in many cases, while a bounded unrolling may be necessary in some case. In the latter case, the proof of safety is valid up to the loop unroll bound [77]. The program and the property are symbolically encoded into an SMT formula: some constraints come straight from the program instructions, while scheduling constraints ensure that concurrent behavior is properly modeled. A theory solver is integrated into the SMT solver ensuring that scheduling constraints are satisfied.

A program execution is correct if it satisfies the safety property given as input to the verification task. The program is safe if all valid executions are correct. The aim of the verification is to prove that the program is safe, or to find a valid execution where the correctness property is violated.

I adapt and combine the methods and ideas of *He et al.* [61], *Sun et al.* [82] and *Zennou et al.* [88] for verifying multi-threaded programs under sequential consistency being recent and efficient partial order-based approaches. While reasoning about program executions conforming to the sequential consistency (SC) memory model is NP-complete [50], *Zennou et al.* observe that reasoning is polynomial for weak sequential consistency (wSC), a slightly weaker memory model than SC [88]. The idea is to reason efficiently under wSC first. If the program is found safe under wSC, it is also safe under the stronger SC model. If an execution violating the safety property is found that is valid under wSC, it must be checked if it is also valid under SC. The algorithm for reasoning under wSC is taken from *Sun et al.* [82]. Checking SC validity is done by applying the encoding of *He et al.* [61] for determining the existence of a total order on write events (being the only difference between SC and wSC).

5.3.1 Symbolic Encoding of Multi-threaded Programs

For the symbolic encoding of the program, each variable access is converted to an SMT variable. Practically, the name of the SMT variable can be the name of the accessed program variable with a unique index[4]. I write w_{x_i} or r_{x_i} for a write or read event (or e_{x_i} in general). By abusing the notation, I simply refer to $grd_{e_{x_i}}$ as grd_{x_i}.

First, the assignments of the program are encoded into the first-order formula ρ_{va}. For each write event w_{x_i}, if w_{x_i} is enabled, then the value of x_i is obtained from the right-hand side expression $expr_{x_i}$ of the assignment. Formally, we have $\rho_{va} := \bigwedge_{\forall w_{x_i}} (grd_{x_i} \Rightarrow x_i = expr_{x_i})$.

As an example, take the simple program from Figure 5.3a with two threads. Figure 5.3b shows the program where variable accesses are marked with unique indices. For example, the guard of the write event w_{x_1} is $grd_{x_1} := (y_1 = 1)$, while $grd_{x_3} := true$. The value assignment of the program is as follows:

$$x_0 = 0 \wedge y_0 = 0 \wedge (y_1 = 1 \Rightarrow x_1 = 1) \wedge (y_1 \neq 1 \Rightarrow x_2 = 1 - y_2) \wedge y_3 = x_3 \wedge x_4 = 1 - y_4$$

The program order constraints are encoded into ρ_{po} based on the order of instructions per each thread. In case of branches, we can simply put the branches one after another in the total order since only one branch can be enabled in any valid execution anyway. Furthermore, thread creation and join operations are also reflected in ρ_{po} by for example relating events of the creator thread before the thread creation to events of the created thread. The program order is represented in Figure 5.3c with solid black edges[5].

To encode read-from constraints, a new Boolean variable $rf^x_{i,j}$ is introduced for each pair of events w_{x_i} and r_{x_j} such that r_{x_j} may read the value written by w_{x_i}. If $rf^x_{i,j}$ is true, then r_{x_j} gets its value from w_{x_i} (i.e., $w_{x_i} \prec_{rf} r_{x_j}$), therefore x_i must be the same value as x_j and both events must be enabled. Formally, we have $rf^x_{i,j} \Rightarrow x_i = x_j \wedge grd_{x_i} \wedge grd_{x_j}$ which is called the *RF-Val* constraint [61]. Also, if a read event is enabled then it must read its value from a write event which leads us to the *RF-Some* constraint: $grd_{x_j} \Rightarrow rf^x_{i_1,j} \vee ... \vee rf^x_{i_n,j}$ assuming that r_{x_j} can read from $w_{x_{i_1}}$, ..., or $w_{x_{i_n}}$. We also have *RF-Ord* which connects the *rf* variables and the $\prec_{rf}$ relation: $rf^x_{i,j} \Leftrightarrow w_{x_i} \prec_{rf} r_{x_j}$. The above constraints are encoded in the formula $\rho_{rf\text{-}val}$, $\rho_{rf\text{-}some}$, and $\rho_{rf\text{-}ord}$ respectively.

In our example, we would create the following rf-variables for the read event x_3: $rf^x_{0,3}$, $rf^x_{1,3}$, $rf^x_{2,3}$ as x_3 could read from any of these three write events. An *RF-Val* constraint would be $rf^x_{1,3} \Rightarrow x_1 = x_3 \wedge y_1 = 1$ (since grd_{x_3} is *true*). An *RF-Some* constraint would be $rf^x_{0,3} \vee rf^x_{1,3} \vee rf^x_{2,3}$ (the implication is simplified as x_3 is still always enabled). Finally, the $rf^x_{0,3} \Leftrightarrow w_{x_0} \prec_{rf} r_{x_3}$ *RF-Ord* constraint would give the semantics of the $rf^x_{0,3}$ variable.

In case where a violating program execution is found by the first stage of the verification, we also need to encode write serialization constraints to check SC conformity. For this purpose, similarly to read-from constraints, a new Boolean variable $ws^x_{i,j}$ is introduced for each pair of same-memory write events. If $ws^x_{i,j}$ is true, then w_{x_i} happens-before w_{x_j}. Formally, we have the following constraints: $ws^x_{i,j} \Rightarrow grd_{x_i} \wedge grd_{x_j}$ called *WS-Cond*; $grd_{x_i} \wedge grd_{x_j} \Rightarrow ws^x_{i,j} \vee ws^x_{j,i}$ called *WS-Some*; and $ws^x_{i,j} \Rightarrow w_{x_i} \prec_{ws} w_{x_j}$ called *WS-Ord* [61].

Finally, we have a correctness property to encode. Since we are looking for safety violations, the correctness property is negated to obtain an error property. The focus of this paper is the verification of error reachability: this way, the error formula ρ_{err} is the guard condition of the marked error in the program. This is often expressed as an assertion in the program: the error formula is the negation of the asserted expression (in conjunction

[4]Some works refer to this form as concurrent static single assignment [61] which name is slightly misleading as static single assignment originally means a compilable form of the program with fresh copies of variables per write access [37]. The applied form cannot be compiled and has fresh copies per each access.

[5]The program order relation is in fact the transitive closure of the solid black edges in Figure 5.3c.

with the guard of branches needed to reach the assert statement). The error property in the example is the negation of the asserted expression, that is, $\neg(x_4 = 1 \vee y_3 = 1)$.

Now, we have obtained all constraints that we need to formulate the verification problem as an SMT problem. We have two kinds of constraints:

$$\begin{aligned}\psi_{ssa} &:= \rho_{va} \wedge \rho_{err} \wedge \rho_{rf\text{-}val} \wedge \rho_{rf\text{-}some} \quad (\wedge \rho_{ws\text{-}val} \wedge \rho_{ws\text{-}some})\\ \psi_{ord} &:= \rho_{po} \wedge \rho_{rf\text{-}ord} \quad (\wedge \rho_{ws\text{-}ord})\end{aligned}$$

where ψ_{ssa} only contains expressions that standard SMT solvers can handle, while ψ_{ord} contains the ordering formulae that need a specific theory solver (note the $\prec$ symbols in ψ_{ord} which is not a symbol of a usual SMT theory). The whole formula then becomes $\Psi = \psi_{ssa} \wedge \psi_{ord}$. If Ψ is satisfiable, then a satisfying model gives a valid execution that leads to an error. However, if Ψ is unsatisfiable, there are no valid executions where the error property is also met, thus the program is safe [61]. I use the formulae without the parts in brackets for the first check under wSC. If it turns out to be unsatisfiable, then the program is safe under both wSC and SC. However, if it is satisfiable, I extend the formulae with the bracketed parts to check under SC.

5.3.2 Ordering Consistency Theory

For the theory solver that can handle ψ_{ord}, we need to define a set of axioms for propagation rules and conflict detection as we have seen in Section 2.1. Fortunately, I have already defined Axiom 1, Axiom 2, and Axiom 3 which can be used as the axioms of the theory solver. Performing propagation and conflict detection with these axioms allows us to check if there is any execution violating the safety property that is valid under the wSC model [88, 82]. If the constraints for a total order of writes is also included in the encoding formula in the potential second stage of the verification algorithm, then these axioms can be used to check the existence of a violating execution under SC [61].

The events and the happens-before relation can be represented in the theory solver as a graph where the vertices are events and edges are relations. At the beginning of the SMT solving procedure, all *rf* variables are unassigned, therefore only edges corresponding to $\prec_{po}$ are present in the graph.

The theory solver has three main components: theory propagation, consistency checking, and conflict clause generation. Theory propagation is applied when a new ordering variable *rf* is assigned *true* by the SMT solver. Then the theory axioms are used to derive all possible new orders. As theory propagation is not crucial for the main contribution of this paper, I refrain from going into further details regarding the propagation algorithm. I refer the interested reader for several possible propagation algorithms described in [61, 82]. Consistency checking is also straightforward: we have to check whether $e \prec e$ holds for any event e in the current partial variable assignment after propagating event orders. An order $e \prec e$ (a conflict) is equivalent to a self-loop in the event graph, so consistency checking amounts to checking the presence of a self-loop in the event graph. As conflict clause generation is crucial for the methods presented in this work, I elaborate a bit more on that.

If an inconsistency is found by the theory solver, it must generate a conflict clause that prevents this inconsistency and provide this conflict clause to the SMT solver. Therefore, the theory solver stores derivation reasons for each derived happens-before order as first-order logic formulae [82]. The following definition defines reasons of orders using the notation of the corresponding axioms where needed:

Definition 16. The reasons for different types of orders are defined as:

- $reason(e_1 \prec_{po} e_2) := true$, since these orders come from the program structure and must be satisfied at all times,
- $reason(w_{x_i} \prec_{rf} r_{x_j}) := rf^x_{i,j}$, that is the corresponding Boolean variable,
- $reason(w_{x_i} \prec_{ws} w_{x_j}) := ws^x_{i,j}$, that is the corresponding Boolean variable,
- the reason for an order derived by Axiom 1 is:
  $reason(e_1 \prec e_3) := reason(e_1 \prec e_2) \land reason(e_2 \prec e_3)$,
- the reason for an order derived by Axiom 2 is:
  $reason(w' \prec_{ws} w) := reason(w \prec_{rf} r) \land reason(w' \prec r) \land grd_{w'}$,
- the reason for an order derived by Axiom 3 is:
  $reason(r \prec_{fr} w') := reason(w \prec_{rf} r) \land reason(w \prec w') \land grd_{w'}$. ▪

Now, when the theory solver finds an inconsistency, that is, a self-loop $e \prec e$ in the event graph, it can simply generate the conflict clause $reason(e \prec e)$. Adding the negation of this formula to the set of assertions prevents the SMT solver from finding a model with the same loop in the event graph again.

As an example, consider a partial variable assignment corresponding to Figure 5.3c in our running example: where $rf^x_{2,5}$ and $rf^x_{2,3}$ have been assigned *true*. The solver can now perform a write-serialization derivation on the events w_{x_4}, w_{x_2}, and r_{x_5} since the preconditions of Axiom 2 hold: $w_{x_2} \prec_{rf} r_{x_5}$, $w_{x_4} \prec_{(po)} r_{x_5}$, and w_{x_4} is always enabled. So the order $w_{x_4} \prec_{ws} w_{x_2}$ is inferred and added to the event graph as Figure 5.3c also demonstrates. Then, we can find a loop in the event graph, which conflict is also found as a self-loop (e.g., $w_{x_2} \prec w_{x_2}$) by the theory solver after applying the transitivity axiom a few times. Then, the following conflict clause is generated:

$$\begin{aligned}
&reason(w_{x_2} \prec_{rf} r_{x_3}) \land reason(r_{x_3} \prec_{po} w_{x_4}) \land reason(w_{x_4} \prec_{ws} w_{x_2}) = \\
&reason(w_{x_2} \prec_{rf} r_{x_3}) \land true \land \big(reason(w_{x_2} \prec_{rf} r_{x_5}) \land reason(w_{x_4} \prec_{po} r_{x_5}) \land grd_{x_4}\big) = \\
&\quad rf^x_{2,3} \land rf^x_{2,5}
\end{aligned}$$

where some *true* reasons, and *true* guards are omitted.

5.4 Automatic Conflict Detection

The presented verification algorithm is heavily integrated into SMT solving by putting a considerable portion of reasoning into an SMT theory. While this is a nice and general approach to the problem, leaving all calculations to the SMT solver (and the theory solver therein) might not be optimal in performance. I present a novel way to improve the performance and scalability of the verification by performing some pre-processing step before giving the encoded formula to the SMT (and theory) solver.

It has been introduced in Section 5.3.2 how the theory solver finds conflicts, inconsistencies in the scheduling constraints. An experimental analysis of conflicts found in real programs revealed that the majority of these conflicts are simple in the sense that the self-loops found in the event graph are typically derived from only a few other relations. Having such simple conflicts motivates a pre-processing algorithm that can easily find these conflicts without the computational overhead of the SMT solving procedure. The idea of my proposed algorithm is to explore the program structure and retrieve possible conflicts of bounded size in the number of derivation axioms necessary to obtain them.

In this section, I present my general framework for reasoning with an over-approximation of the happens-before relations covering all possible behavior. Then, I present a general approach to discover potential conflicting concurrent behavior. After presenting the general approach, I also introduce an algorithm in this section that turns out to be both efficient in terms of complexity and effective in terms of impact in my evaluation.

5.4.1 Over-Approximation of the Happens-Before Relation

At a pre-processing stage, we cannot speak about existing conflicts since they arise during the SMT solving procedure when *rf* variables get assigned by the SMT solver, new happens-before relations are derived by the ordering consistency theory solver using the axioms of Section 5.3.2, and self-loops are found in the event graph. Before starting the SMT solving, the values of *rf* variables are unknown since they are variables. Therefore, pairs of events are not *rf*-related yet, axioms have not been applied, and actual conflicts have not been found until my proposed algorithm is executed. Instead, the algorithm deals with a *potential rf* relation and tries to find possible conflicts. I use the $<$ symbol for *potential* happens-before relations[6] to highlight the semantic difference from the real happens-before relation. I also define this potential relation similarly to the original happens-before relation. First, I introduce the potential program order and potential read-from relations:

- the potential program order is simply the same as the original program order relation: $<_{po} = \prec_{po}$ (it is based on the program code);
- $w <_{rf} r$ if r may read the value written by w (that is, when an *rf* variable is created during the symbolic encoding of the program).

Definition 17 (Potential Happens-Before $<$). $e_1 < e_2$ holds for events e_1 and e_2 if:

- *$e_1 <_{po} e_2$, or*
- *$e_1 <_{rf} e_2$, or*
- *$e_1 < e_2$ can be derived by a sequence of derivation steps based on the axioms Axiom 4, Axiom 5, and Axiom 6..* ▪

Axiom 4 (Transitivity Derivation for $<$) *For any events e_1, e_2, and e_3:*

$(e_1 < e_2) \wedge (e_2 < e_3) \Rightarrow (e_1 < e_3)$.

Axiom 5 (Write-Serialization Derivation for $<$) *For any write events w and w', and any read event r such that they all belong to the same memory address:*

$(w <_{rf} r) \wedge (w' < r) \Rightarrow (w' < w)$.

Axiom 6 (From-Read Derivation for $<$) *For any write events w and w', and any read event r such that they all belong to the same memory address:*

$(w <_{rf} r) \wedge (w < w') \Rightarrow (r < w')$.

[6]Some related works use the *may* prefix for the over-approximation of certain relations or sets, or some alternative terminology [56]. Instead, I opted for the *potential* prefix to avoid the slightly ambiguous meaning of *may* and the grammatically dreadful *may-happens-before* name.

Remember that the happens-before relation relates events of the program in the context of an execution (or set of executions). The newly introduced potential happens-before relation relates events of the program without restricting the scope to any executions: this is a relation describing potential concurrent behavior based on the static program structure. This is also the reason for omitting the guard condition from the new axioms compared to the original axioms: we cannot speak about enabled guards without the scope of an execution.

Example 11. *To illustrate the difference between $<$ and $\prec$, take our running example from Figure 5.3. Events x_2 and x_3 are* rf*-related ($x_2 \prec_{rf} x_3$) in the execution $x_0, y_0, y_1, y_2, x_2, x_3, y_3, y_4, x_4, x_5, y_5$ (with x_1 disabled). However, $x_2 \not\prec_{rf} x_3$ in the trace $x_0, y_0, y_1, x_3, y_3, y_2, x_2, y_4, x_4, x_5, y_5$. Since x_3 may read the value written by x_2 (as in the first case), $x_2 <_{rf} x_3$ universally.*

The potential happens-before relation can also be seen as a real happens-before relation in a model with all *rf* variables set to *true*. As a consequence, the potential happens-before relation naturally contains a lot of cycles in most cases (except for programs with very simple structures). My proposed approach is to discover cycles in the potential happens-before relation and use these potential conflicts to strenghten the program encoding formula.

Even though some elements of the potential happens-before relation might not materialize as an element of the real happens-before relation of a valid execution, the found potential happens-before loops cannot occur in any valid execution of the program. Thus, we can use these loops similarly to infer conflict clauses and supply the negations of these conflict clauses to the SMT solver as assertions. To convert the found loops in $<$ to propositional formulae, reasons can be defined similarly to Definition 16. I intentionally omit the *ws* variables and the explicit encoding of write serialization constraints in this section to keep the approach on a more general level: this way we only discover loops that are conflicting with both the wSC and the SC models.

Definition 18. The reasons for different types of the potential happens before relation are:

- $\mathit{reason}(e_1 <_{po} e_2) := \mathit{true}$,
- $\mathit{reason}(w_{x_i} <_{rf} r_{x_j}) := rf^{x}_{i,j}$,
- the reason for an order derived by Axiom 4 is:
  $\mathit{reason}(e_1 < e_3) := \mathit{reason}(e_1 < e_2) \wedge \mathit{reason}(e_2 < e_3)$,
- the reason for an order derived by Axiom 5 is:
  $\mathit{reason}(w' <_{ws} w) := \mathit{reason}(w <_{rf} r) \wedge \mathit{reason}(w' < r) \wedge \mathit{grd}_{w'}$,
- the reason for an order derived by Axiom 6 is:
  $\mathit{reason}(r <_{fr} w') := \mathit{reason}(w <_{rf} r) \wedge \mathit{reason}(w < w') \wedge \mathit{grd}_{w'}$. ▪

This definition is indeed the same syntactically for $<$ as Definition 16 for $\prec$. Note that even guard conditions are used similarly and they are not omitted from the reason formulae even though guards are omitted from the axioms for $<$. This is due to the semantic difference of *reasons* used in the two contexts. For the original happens-before relation, $\mathit{reason}(e_1 \prec e_2)$ refers to the preconditions that induce $e_1 \prec e_2$. For the potential happens-before relation, $\mathit{reason}(e_1 < e_2)$ does not refer to the preconditions inducing $e_1 < e_2$; it refers to the preconditions that make the potential $e_1 < e_2$ materialize as a real $e_1 \prec e_2$. The following lemma formalizes this connection between *reasons* for $<$ and the original happens-before relation. The subsequent theorem states the soundness of my approach.

Lemma 4. If $\mathit{reason}(e_1 < e_2)$ is true in a model of the formula, then $e_1 \prec e_2$ can be derived in this model. ▪

Proof. I prove the lemma by structural induction on the definition of *reason*. For this proof, it is useful to imagine $\mathit{reason}(e_1 < e_2)$ as a tree structure (I refer to it as the *reason tree*) where $\mathit{reason}(e_1 < e_2)$ is the root node, each non-leaf node is a *reason* of a pair of $<$-related events, the children of a node are given by Definition 18, and leaf nodes are first-order terms:

- *true*, when its parent contains a pair of events related in $<_{po}$,
- an *rf* variable, when its parent contains a pair of events related in $<_{rf}$, or
- a guard condition, when its parent is derived from other potential orders by Axiom 5 or Axiom 6.

Looking at Definition 18 of *reasons* for the potential happens-before relation, we can observe that $\mathit{reason}(e_1 < e_2)$ is in fact a large logical *and* expression of the leaf terms of its *reason tree*. If $\mathit{reason}(e_1 < e_2)$ is *true*, then all leaf terms must also be *true* due to the nature of the *and* operator. Thus, we can start from the leaves of the *reason tree* and prove by induction for each (non-leaf) node $\mathit{reason}(f_1 < f_2)$ that $f_1 \prec f_2$ can be derived. Reaching the root of the *reason tree* proves that $e_1 \prec e_2$ can be derived.

We have two base cases for non-leaf nodes with only leaf children:

1. The node is $\mathit{reason}(f_1 <_{po} f_2)$:

   $\mathit{reason}(f_1 <_{po} f_2)$ is always true by definition, and $<_{po} = \prec_{po}$ implies $f_1 \prec f_2$.

2. The node is $f_1 <_{rf} f_2$ with $f_1 = w_{x_i}$ and $f_2 = r_{x_j}$:

   $\mathit{reason}(f_1 <_{rf} f_2) = \mathit{rf}^x_{i,j}$ by definition. The leaf $\mathit{rf}^x_{i,j}$ is true in the model based on our observation for leaves which means that $f_1 \prec_{rf} f_2$ (c.f., RF-Ord constraints).

For the inductive case of a non-leaf node with some non-leaf children, we have three cases:

1. $\mathit{reason}(f_1 < f_2)$ with $f_1 < f_2$ derived by Axiom 4 as the transitive closure of $f_1 < f'$ and $f' < f_2$:

   We know that $f_1 \prec f'$ and $f' \prec f_2$ can be derived by the induction hypothesis. Applying the transitivity axiom Axiom 1 for the original happens-before relation immediately gives that $f_1 \prec f_2$.

2. $\mathit{reason}(f_1 <_{ws} f_2)$ with $f_1 = w'$ and $f_2 = w$, and $f_1 <_{ws} f_2$ derived by Axiom 5 from $w <_{rf} r$ and $w' < r$ for some read event r:

   Again, we know that $w \prec_{rf} r$ and $w' \prec r$ from the $<_{rf}$ base case and the induction hypothesis. We also know that the leaf child $\mathit{grd}_{w'}$ is also *true* based on our observation above. This way, Axiom 2 gives that $w' \prec_{ws} w$, that is, $f_1 \prec f_2$.

3. $\mathit{reason}(f_1 <_{fr} f_2)$. The case is completely analogous with the previous case. □

Theorem 5. Let Ψ be the encoding formula of a program and a safety property, and let E be a set of events e such that $e < e$.

Ψ is satisfiable if and only if $\Psi \wedge \bigwedge_{e \in E} \neg reason(e < e)$ is satisfiable. ▪

Proof. Trivially, if the right-hand formula is satisfiable, then Ψ is similarly so since the right-hand formula is more restrictive.

For the other way, first I observe that no valid execution satisfies $reason(e < e)$ for any self-loop $e < e$. To see this assume that we have a valid execution which satisfies $reason(e < e)$. Lemma 4 implies that $e \prec e$ can be derived in this model. Thus, we reach a contradiction since no valid execution can have a loop in its (original) happens-before relation.

By our observation, all valid executions must satisfy $\neg reason(e < e)$ for any self-loop in $<$. Since for each $e \in E$, all valid executions satisfy $\neg reason(e < e)$, we arrive to the conclusion that any execution satisfying Ψ also satisfies the right-hand formula. □

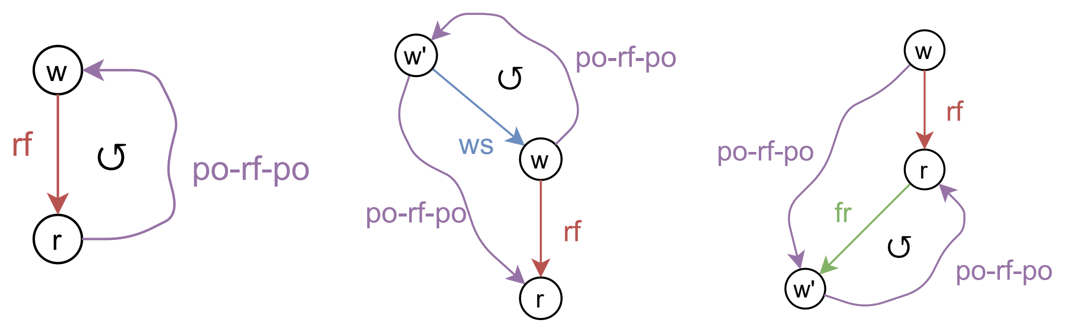
Theorem 5 basically says that no valid execution satisfies reasons of self-loops in the potential happens-before relation, therefore the negations of these reasons are properties that generally characterize all possible program executions. Thus, any algorithm that finds any kind of cycles in the potential happens-before relation and uses these conflicts to strengthen the encoding formula as explained above is safe to use and does not change the verification outcome. I formalize such an algorithm in the next section. The algorithm is constrained to find simple conflicts to have a bound for its complexity and a bound for the size of the extended encoding formula which is desirable in practice. However, different and more complicated algorithms looking for complicated cycles could also be used theoretically, and their correctness would also be guaranteed by Theorem 5.

5.4.2 Bounded Cycles in the Potential Happens-Before Relation

My proposed algorithm looks for potential conflicts, loops in the potential happens-before relation with the basic structures depicted in Figure 5.4. A *po-rf-po* edge means that it could be replaced by at most one *rf* and optional *po* edges before or after this *rf*. Formally, $e_1 <_{\textit{po-rf-po}} e_2$ if $e_1 <_{po} e_2$ or there are events e_1', e_2' such that $e_1 <_{po} e_1' <_{rf} e_2' <_{po} e_2$. To put it simple, the algorithm looks for potential conflicts that may be derived from an *rf*, a *ws*, or an *fr* and another *rf* or some *po* relations. Algorithm 3 describes the algorithm used to find potential conflicts before SMT solving. I abuse the notation by writing $w' <_{ws} w <_{\textit{po-rf-po}} w'$ for example to denote the self-loop $w' < w'$.

Looking at Algorithm 3 or the visualization from Figure 5.4, it is straightforward that Algorithm 3 indeed finds self-loops in $<$. Thus, it is safe to use this algorithm for optimization based on Theorem 5. The impact of the algorithm on verification performance is revealed in Section 5.5.

Finding these simple conflicts is polynomial in the number of program events. The complexity of the proposed algorithm for finding these simple conflicts is $\mathcal{O}(n^2 \cdot m)$ where n is the size of the $<_{rf}$ relation (the number of potentially rf-related event pairs which is at most quadratic in the number of events) and m is the number of write events in the program. Naturally, $n^2 \cdot m$ is a gross overestimate in most cases since the foreach loop of line 6 only iterates over write events belonging to the same memory location as the rf-relation of the outer loop. The size of the encoding formula is increased with a similar complexity by adding the discovered conflict clauses. Evaluation shows however that adding these clauses accelerates the SMT decision procedure.



(a) Conflict with *rf* **(b)** Conflict with *ws* **(c)** Conflict with *fr*

Figure 5.4: Conflicts with basic structures

Algorithm 3: Finding simple conflicts

```
$conflicts \leftarrow \{\}$
foreach $w <_{rf} r$ do
    foreach $w' <_{rf} r'$ do
        if $r' <_{po} w$ and $r <_{po} w'$ then
            $conflicts \leftarrow conflicts \cup \{w <_{rf} r <_{po} w' <_{rf} r' <_{po} w\}$   // Fig.  5.4a
        end
    end
    foreach $w'$ that may write the same memory location as $w$ do
        if $w' <_{po\text{-}rf\text{-}po} r \wedge w <_{po\text{-}rf\text{-}po} w'$ then
            $conflicts \leftarrow conflicts \cup \{w' <_{ws} w <_{po\text{-}rf\text{-}po} w'\}$   // Fig.  5.4b
            $conflicts \leftarrow conflicts \cup \{r <_{fr} w' <_{po\text{-}rf\text{-}po} r\}$   // Fig.  5.4c
        end
    end
end
```

5.5 Experimental Evaluation

In this section, I evaluate the efficiency of my automatic conflict detection optimization for partial order-based verification of concurrent software.

Benchmark tests were executed on virtual machines in the cloud computing platform of our university with Intel Core (Skylake) processors. Three dedicated CPU cores were allocated to each task. Each verification task had a time limit of 900 seconds and a memory limit of 15GB.

The goal of my experiments is to analyze the impact of strengthening the encoding formula by generated conflicts described in Section 5.4 on the overall verification performance. To have a broader image of how my algorithm helps existing techniques, I implemented two different methods for partial order-based verification. My first implementation resembles the approach of [87] where the initial encoding formula does not contain all scheduling constraints and the formula is gradually refined when the SMT solver finds a conflicting assignment. My second implementation follows the scheme described in Section 5.3 with a dedicated ordering consistency theory solver integrated into Z3 [41] via user propagators [31]. I refer to these versions as `BASIC` and `PROP` (short for propagator), respectively. I implemented[7] these proof-of-concept solutions in the THETA verification framework which could already parse C programs into control-flow automata [83].

5.5.1 Research Questions and Experiment Setup

I aim to answer the following questions for the evaluation of my technique:

RQ5.1 How many (simple) conflicts are found by my algorithm and how does this reduce the number of conflicts occurring during the decision procedure?

RQ5.2 What is the impact of further constraining the encoding formula on the size of the SMT solver model search space?

RQ5.3 What is the computational overhead of finding conflicts? In turn, how is the SMT solving time and the overall verification time reduced?

RQ5.4 How does the performance of my solution compare to the state-of-the-art?

I executed different configurations of THETA over the subset of input programs written in C from the concurrency safety reachability category benchmark suite[8] of SV-COMP [20] (715 tasks) that is parsable by THETA and supported by the partial-order based analysis: 544 tasks in total. The configurations are the two baseline implementations (`BASIC` and `PROP`), and the optimized versions (`BASIC+OPT` and `PROP+OPT`). For RQ5.4, I compare my solution to DEAGLE [62], a state-of-the-art bounded model checker built on CBMC based on the preventive propagation algorithm of [82], winner of the concurrency category of SV-COMP 2023 [19]. DEAGLE is shown to outperform other bounded model checkers, therefore I only compare to DEAGLE [82, 19]. I apply loop unwinding for programs with loops as a standard technique in bounded model checking. If the number of loop iterations can be statically decided, the loop is unwinded according to this number. Otherwise, loops are unrolled up to two iterations (in such cases, only a bounded proof of safety may be obtained). This loop unwinding tactic allows a fair comparison with DEAGLE applying the same method [62].

5.5.2 Experiment Results

Table 5.1 shows the results for the most important metrics of the evaluation. Each configuration produced bounded proofs for two or three more tasks where the tasks are in fact unsafe beyond the loop unroll bound. However, I excluded these cases from the statistics where the bounded verdict differs from the unbounded verdict. Except for the number of solved tasks, the values in the table are average values per task aggregated for the commonly solved tasks per algorithm type (508 tasks commonly solved by the `BASIC` configurations and 512 by the `PROP` configurations). The first two columns show relevant metrics to the end-user: number of verified programs and the overall CPU time. The next two columns contain information about the decision procedures on the level of reasoning about partial orders: first the number of propagated (in case of `PROP`) or refinement (in case of `BASIC`) clauses, then the time taken by the SMT solver are listed. Finally, the last three columns give an intuition of the size of the search space explored by the SMT solver. I interpret these figures along with further statistics from the benchmark tests by answering the research questions.

[7]https://github.com/csanadtelbisz/theta/tree/xcfa-oc-ws
[8]https://gitlab.com/sosy-lab/benchmarking/sv-benchmarks

algorithm	solved tasks	CPU time	prop./ref. clauses	Solver time	Z3 inner decisions	Z3 inner propagations	Z3 inner restarts
BASIC	510	24.4s	548	15.5s	117539	1019525	8.35
BASIC+OPT	517	14.2s	223	5.5s	28074	289943	1.20
PROP	514	22.8s	1017	15.1s	16886	301890	6.83
PROP+OPT	520	15.8s	599	7.3s	9203	243810	5.36

Table 5.1: Different metrics of the evaluation of the optimized verification algorithm

RQ5.1 An average of 1125.7 conflicts are detected by our proposed method per task on the subset of commonly solved tasks of all configurations. Out of this, there are 514.5 conflicts with the structure of Figure 5.4a and 611.2 conflicts following Figure 5.4b or Figure 5.4c on average (note that we have the same number of Figure 5.4b and Figure 5.4c conflicts based on Algorithm 3). Strengthening the encoding formula with these conflict clauses found before starting the SMT solving procedure greatly reduces the number of conflict clauses produced by the decision procedures (see Table 5.1): the number of refinement clauses produced by the BASIC configurations is reduced by an average of 59.3% while the number of propagated conflict clauses by PROP is reduced by 41.1%. Note that more conflicts are generated the new way (simple conflicts discovered by my method and the conflicts generated by the decision procedures) than the original decision procedures produce, however this verbosity positively affects the performance of SMT solving.

RQ5.2 Various statistic data provided by Z3 can give us a rough estimate of the impact of my algorithm on the solver search space. The last columns of Table 5.1 show the average number of decision points, performed (unit and binary) propagation operations and necessary restarts. These numbers refer to inner Z3 operations (i.e., not the number of conflict clause propagations by PROP for instance). For the exact interpretation of these concepts, I refer the interested reader to a book on SMT [17] or the Z3 code base[9]. For the purposes of this evaluation, it is enough to see that the number of these important operations are substantially reduced indicating that the search space becomes much smaller when my algorithm is used. These values are reduced by 70-85% for the BASIC algorithm and by 20-45% for the PROP algorithm. Note that PROP working with a theory solver already achieves a huge reduction in the search space compared to BASIC (which is no surprise based on the paper of Sun et al. [82]); my proposed algorithm can still perform a significant further reduction for PROP.

RQ5.3 The runtime overhead of my conflict generation algorithm is not substantial, often negligible: it is around 0.6 seconds per task on average which means less than 5% of all CPU time. In turn, the time of the SMT solving procedure and thus the overall verification time is significantly reduced: using my optimization the SMT solver is 64.9% faster in case of the BASIC algorithm, and 51.4% faster in case of PROP. The overall CPU time of the verification is reduced by a significant 41.7% for BASIC and 30.7% for PROP on average. The CPU times are also demonstrated in the scatter plots of Figure 5.5 where the majority of points are under the diagonal indicating the performance improvement of my proposed technique.

[9]https://github.com/Z3Prover/z3

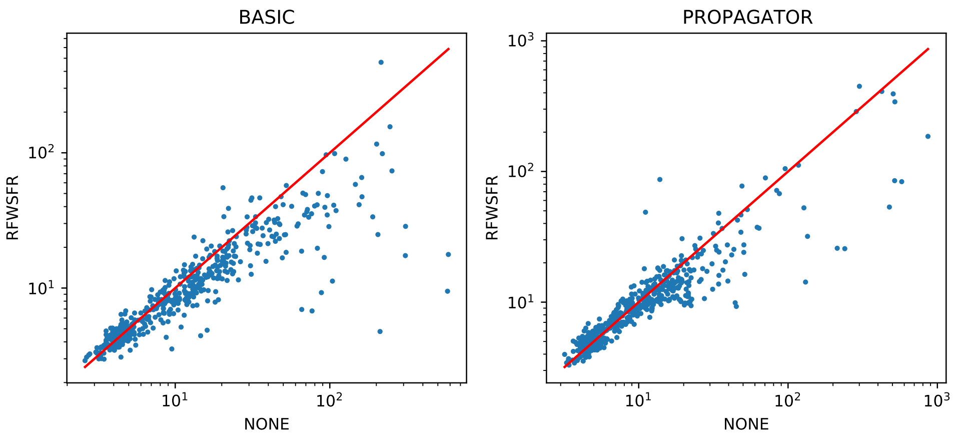


Figure 5.5: CPU time improvement of the optimized verification

RQ5.4 Firstly, I note that I do not intend to compete with DEAGLE in terms of absolute performance (CPU time) as my proof-of-concept implementation runs on the JVM while DEAGLE is built on native CBMC code optimized over decades. Having said that, my evaluation shows that my solution with the best configuration (`PROP` algorithm working with similar principles as DEAGLE extended with my optimization) could verify as much tasks as DEAGLE: 520 tasks. Levelling with a verifier with a highly optimized codebase clearly shows the potential of my contributions. Furthermore, my solution could produce bounded proofs for two more tasks where the tasks are in fact unsafe beyond the loop unroll bound. DEAGLE is shown to outperform other state-of-the-art bounded model checkers and no other published results surpass DEAGLE [82, 19, 20] which implies that my solution also outperforms other existing tools. It is also important to note that DEAGLE is limited to verification under *weak sequential consistency* (see the supplement material) while my solution is suitable for the stronger and more useful *sequential consistency* memory model.

In summary, I can conclude that my solution and implementation *advances the state-of-the-art* in bounded model checking multi-threaded programs.

5.5.3 Threats to Validity

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

Internal validity. Consistency and accuracy of the experiments were insured by using the BenchExec framework [28]. I executed my experiments on virtual machines in a cloud computing platform. 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.

External validity. The SV-COMP benchmark suite is considered a de facto standard for academic benchmarking of software verification algorithms. Still, evaluation results might not generalize well to real-life industrial programs. To justify the impact of my proposed algorithm more thoroughly, I implemented and tested my extension to the verifier algorithm with two state-of-the-art baseline approaches.

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, number of solved tasks) and *backend-related* information (such as the number of conflicts and the solver search space) were used. Therefore, these metrics accurately represent the expected outcomes of the executions.

Furthermore, I would like to note that in my comparative evaluation I used the binary of Deagle submitted to SV-COMP 2024 [20]. I also tried to reproduce the results by building the binary of Deagle from its source code[10] but I failed to get the same results as several incorrect verdicts (both false positives and negatives) were produced by this version. However, if Deagle is indeed only capable of the slightly worse results, that only affects the placement of my solution among state-of-the-art techniques positively. Thus, this is only a positive threat to the validity of my evaluation.

5.6 Related Work

In the 2010s, *Alglave et al.* introduced partial order-based symbolic approaches for model checking multi-threaded programs [8]. Early works using partial orders typically include all scheduling constraints in the encoding formula. Each memory event is associated with a clock value, and scheduling constraints are formulated in *integer difference logic*. Alternative solutions use the theory of partial strings for describing partial order constraints [65, 66].

Later, several improvements have been developed for these approaches lazily encoding the scheduling constraints. *Yin et al.* developed an abstraction-refinement method for gradually refining the encoding formula starting without scheduling constraints [87]. *He et al.* proposed a dedicated theory solver integrated into the SMT solver Z3 [41] for reasoning about scheduling constraints [61]. The theory solver is able to perform value propagation and to detect conflicts based on the partial variable assignment of the SMT solver. They also reduce the size of the encoding formula as some constraints can be replaced by reasoning in the theory solver. An improved *preventive* propagation algorithm is introduced by *Sun et al.* [82]. Their algorithm is able to detect conflicts even before the SMT solver reaches a partial variable assignment implying the conflict; the theory solver prevents conflicting variable assignments by propagating some necessary preventive clauses. In fact, whenever only one related event pair is missing from a cycle, their solution already detects the cycle and excludes it by a corresponding clause.

While many of these algorithms assume a sequential consistency memory model, extending partial order-based reasoning to weak memory models has long been researched as well [7, 38, 82]. Recently, *Haas et al.* proposed a theory with a theory solver for specifying and handling general memory consistency models [55] with several optimizations [56]. The presented approaches serve as the base algorithms of successful bounded model checkers for concurrent software [62, 87, 39].

Using a dedicated theory solver already significantly constrains the SMT solver during exploring the search space of the formula for a satisfying model [61, 82]. However, the SMT solver may still explore redundant branches of the search space and backtrack unnecessarily with these techniques, since the conflicts are only detected (or prevented) when a corresponding partial variable assignment is reached by the SMT solver. Even the preventive propagation algorithm of *Sun et al.* [82] is only one step ahead: it can only infer one missing partial order for a conflict, so the other elements of the happens-before relation

[10]https://github.com/thufv/Deagle

leading to a cycle still needs to be justified by a corresponding variable assignment. On the other hand, my proposed technique prevents many conflicts right from the beginning. Thus, we can say that my method is even more preventive than the algorithm described in [82]. As my experiments reveal, the clauses obtained from these simple conflicts are often completely instructive for the SMT solver in the sense that only a few or no further propagation steps have to be performed by the theory solver if my proposed optimization is used.

Reasoning with over- or under-approximations of happens-before relations have been used by several works [9, 49, 56]. These approximations are useful for enhancing the encoding formula, though these enhancements typically try to reduce the formula by removing unnecessary or lazily computable constraints. My approach on the other hand adds new constraints to the formula.

Chapter 6

Conclusion

For Thesis I, Chapter 3 presented and proved the soundness of a generic way of integrating partial order reduction techniques with abstraction-based analyses. The soundness of an abstraction-aware POR algorithm is nontrivial, demonstrated by the proofs in Section 3.2. While such dual approaches have been implemented in the past [33, 85, 45], the close integration of the two techniques may lead to vastly enhanced performance, as testified by the program in Figure 3.2. Practical experimentation showed that this advantage is slightly diminished on conventional multi-threaded programs. Yet, the decreased verification times and the explored state space size clearly show the proposed approach's benefits.

For Thesis II, I presented a novel statement reduction algorithm in Chapter 4 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. 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, especially in cases where successor state calculation takes a significant proportion of verification time, such as in the case of predicate abstraction.

For Thesis III, I proposed an optimization based on discovering potential scheduling inconsistencies. I reason with an over-approximation of happens-before relations on the level of the whole program. My algorithm strengthens the program encoding formula by finding scenarios conflicting with concurrent behavior and converting these into first-order clauses. Since I limit the algorithm to discover cycles of bounded size, it does not pose a serious computational overhead on building the encoding formula. The presented approach can be conveniently integrated into existing methods to reduce the search space of SMT solvers and accelerate the decision procedure. Thus, my algorithm effectively finds the trade-off between preserving helpful information in the encoding formula and decreasing its size. Evaluation reveals a significant boost in verification performance and shows that my solution levels with state-of-the-art verification tools.

Concluding the thesis, the presented algorithms improve the performance of multi-threaded program verification. Therefore, the presented techniques are worth considering by developers of concurrency-focused verification tools as part of their analysis portfolios, especially if they already include abstraction-based and partial order reduction-based analyses (for Theses I and II) or a partial order-based analysis (for Thesis III), allowing for a relatively easy implementation of the proposed approaches.

Acknowledgements

This thesis project could not achieve the results described in the thesis without the help, mentoring, and valuable feedback of my thesis advisor Levente Bajczi as well as my two other supervisors Dániel Szekeres and András Vörös. Thank you for your support and guidance throughout my Master's degree program.

Project no. EKÖP-24-2-BME-118 has been implemented with the support provided by the Ministry of Culture and Innovation of Hungary from the National Research, Development and Innovation Fund, financed under the EKÖP-24-2 funding scheme.

This research was partially funded by the Doctoral Excellence Fellowship Programme (funded by the NRDI Fund of Hungary and the BME University).

List of Figures

Bibliography

[1] Parosh Aziz Abdulla, Stavros Aronis, Bengt Jonsson, and Konstantinos Sagonas. Optimal dynamic partial order reduction. pages 373–384. ACM, 2014. DOI: `10.1145/2535838.2535845`.

[2] 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`.

[3] Parosh Aziz Abdulla, Stavros Aronis, Bengt Jonsson, and Konstantinos Sagonas. Source Sets: A Foundation for Optimal Dynamic Partial Order Reduction. *J. ACM*, 64(4):25:1–25:49, 2017. DOI: `10.1145/3073408`.

[4] Pratyush Agarwal, Krishnendu Chatterjee, Shreya Pathak, Andreas Pavlogiannis, and Viktor Toman. Stateless model checking under a reads-value-from equivalence. volume 12759 of *Lecture Notes in Computer Science*, pages 341–366. Springer, 2021. DOI: `10.1007/978-3-030-81685-8_16`.

[5] Hiralal Agrawal and Joseph Robert Horgan. Dynamic program slicing. pages 246–256. ACM, 1990. DOI: `10.1145/93542.93576`.

[6] Elvira Albert, Puri Arenas, Maria Garcia de la Banda, Miguel Gómez-Zamalloa, and Peter J. Stuckey. Context-Sensitive Dynamic Partial Order Reduction. volume 10426 of *Lecture Notes in Computer Science*, pages 526–543. Springer, 2017. DOI: `10.1007/978-3-319-63387-9_26`.

[7] Jade Alglave, Luc Maranget, Susmit Sarkar, and Peter Sewell. Fences in weak memory models (extended version). *Formal Methods Syst. Des.*, 40(2):170–205, 2012. DOI: `10.1007/S10703-011-0135-Z`.

[8] Jade Alglave, Daniel Kroening, and Michael Tautschnig. Partial orders for efficient bounded model checking of concurrent software. volume 8044 of *Lecture Notes in Computer Science*, pages 141–157. Springer, 2013. DOI: `10.1007/978-3-642-39799-8_9`.

[9] Jade Alglave, Daniel Kroening, Vincent Nimal, and Daniel Poetzl. Don't sit on the fence - A static analysis approach to automatic fence insertion. volume 8559 of *Lecture Notes in Computer Science*, pages 508–524. Springer, 2014. DOI: `10.1007/978-3-319-08867-9_33`.

[10] 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`.

[11] Mohamed Faouzi Atig, Ahmed Bouajjani, Sebastian Burckhardt, and Madanlal Musuvathi. On the verification problem for weak memory models. pages 7–18. ACM, 2010. DOI: `10.1145/1706299.1706303`.

[12] David Baelde, Stéphanie Delaune, and Lucca Hirschi. Partial Order Reduction for Security Protocols. *CoRR*, abs/1504.04768, 2015.

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

[14] Daniel Baier, Dirk Beyer, Po-Chun Chien, Marek Jankola, Matthias Kettl, Nian-Ze Lee, Thomas Lemberger, Marian Lingsch Rosenfeld, Martin Spiessl, Henrik Wachowitz, and Philipp Wendler. Cpachecker 2.3 with strategy selection - (competition contribution). volume 14572 of *Lecture Notes in Computer Science*, pages 359–364. Springer, 2024. DOI: `10.1007/978-3-031-57256-2_21`.

[15] Levente Bajczi, Csanád Telbisz, Márk Somorjai, Zsófia Ádám, Mihály Dobos-Kovács, Dániel Szekeres, Milán Mondok, and Vince Molnár. Theta: Abstraction based techniques for verifying concurrency (competition contribution). volume 14572 of *Lecture Notes in Computer Science*, pages 412–417. Springer, 2024. DOI: `10.1007/978-3-031-57256-2_30`.

[16] 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`.

[17] Clark W. Barrett, Roberto Sebastiani, Sanjit A. Seshia, and Cesare Tinelli. Satisfiability modulo theories. volume 336 of *Frontiers in Artificial Intelligence and Applications*, pages 1267–1329. IOS Press, 2021. DOI: `10.3233/FAIA201017`.

[18] 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`.

[19] 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`.

[20] Dirk Beyer. State of the art in software verification and witness validation: SV-COMP 2024. volume 14572 of *Lecture Notes in Computer Science*, pages 299–329. Springer, 2024. DOI: `10.1007/978-3-031-57256-2_15`.

[21] Dirk Beyer and Karlheinz Friedberger. A Light-Weight Approach for Verifying Multi-Threaded Programs with CPAchecker. volume 233 of *EPTCS*, pages 61–71, 2016. DOI: `10.4204/EPTCS.233.6`.

[22] 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`.

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

[24] 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`.

[25] 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`.

[26] Dirk Beyer, M. Erkan Keremoglu, and Philipp Wendler. Predicate abstraction with adjustable-block encoding. pages 189–197. IEEE, 2010.

[27] 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`.

[28] 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`.

[29] Armin Biere. Bounded model checking. volume 336 of *Frontiers in Artificial Intelligence and Applications*, pages 739–764. IOS Press, 2021. DOI: `10.3233/FAIA201002`.

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

[31] Nikolaj S. Bjørner, Clemens Eisenhofer, and Laura Kovács. Satisfiability modulo custom theories in Z3. volume 13881 of *Lecture Notes in Computer Science*, pages 91–105. Springer, 2023. DOI: `10.1007/978-3-031-24950-1_5`.

[32] 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`.

[33] Alessandro Cimatti, Iman Narasamdya, and Marco Roveri. Software Model Checking with Explicit Scheduler and Symbolic Threads. *Log. Methods Comput. Sci.*, 8(2), 2012. DOI: `10.2168/LMCS-8(2:18)2012`.

[34] 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`.

[35] 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`.

[36] Alex Coto, Omar Inverso, Emerson Sales, and Emilio Tuosto. A prototype for data race detection in cseq 3 - (competition contribution). volume 13244 of *Lecture Notes in Computer Science*, pages 413–417. Springer, 2022. DOI: `10.1007/978-3-030-99527-0_23`.

[37] Ron Cytron, Jeanne Ferrante, Barry K. Rosen, Mark N. Wegman, and F. Kenneth Zadeck. An efficient method of computing static single assignment form. pages 25–35. ACM Press, 1989. DOI: `10.1145/75277.75280`.

[38] Hernán Ponce de León, Florian Furbach, Keijo Heljanko, and Roland Meyer. Portability analysis for weak memory models. PORTHOS: one tool for all models. volume 10422 of *Lecture Notes in Computer Science*, pages 299–320. Springer, 2017. DOI: `10.1007/978-3-319-66706-5_15`.

[39] Hernán Ponce de León, Florian Furbach, Keijo Heljanko, and Roland Meyer. Dartagnan: Bounded model checking for weak memory models (competition contribution). volume 12079 of *Lecture Notes in Computer Science*, pages 378–382. Springer, 2020. DOI: 10.1007/978-3-030-45237-7_24.

[40] Hernán Ponce de León, Thomas Haas, and Roland Meyer. Dartagnan: Leveraging compiler optimizations and the price of precision (competition contribution). volume 12652 of *Lecture Notes in Computer Science*, pages 428–432. Springer, 2021. DOI: 10.1007/978-3-030-72013-1_26.

[41] 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.

[42] Leonardo Mendonça de Moura and Nikolaj S. Bjørner. Satisfiability modulo theories: introduction and applications. *Commun. ACM*, 54(9):69–77, 2011. DOI: 10.1145/1995376.1995394.

[43] Daniel Dietsch, Matthias Heizmann, Dominik Klumpp, Frank Schüssele, and Andreas Podelski. Ultimate taipan and race detection in ultimate - (competition contribution). volume 13994 of *Lecture Notes in Computer Science*, pages 582–587. Springer, 2023. DOI: 10.1007/978-3-031-30820-8_40.

[44] 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.

[45] Azadeh Farzan, Dominik Klumpp, and Andreas Podelski. Stratified Commutativity in Verification Algorithms for Concurrent Programs. *Proc. ACM Program. Lang.*, 7 (POPL):1426–1453, 2023. DOI: 10.1145/3571242.

[46] Cormac Flanagan and Patrice Godefroid. Dynamic partial-order reduction for model checking software. pages 110–121. ACM, 2005. DOI: 10.1145/1040305.1040315.

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

[48] Harald Ganzinger, George Hagen, Robert Nieuwenhuis, Albert Oliveras, and Cesare Tinelli. DPLL(T): fast decision procedures. volume 3114 of *Lecture Notes in Computer Science*, pages 175–188. Springer, 2004. DOI: 10.1007/978-3-540-27813-9_14.

[49] Natalia Gavrilenko, Hernán Ponce de León, Florian Furbach, Keijo Heljanko, and Roland Meyer. BMC for weak memory models: Relation analysis for compact SMT encodings. volume 11561 of *Lecture Notes in Computer Science*, pages 355–365. Springer, 2019. DOI: 10.1007/978-3-030-25540-4_19.

[50] Phillip B. Gibbons and Ephraim Korach. Testing shared memories. *SIAM J. Comput.*, 26(4):1208–1244, 1997. DOI: 10.1137/S0097539794279614.

[51] 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.

[52] Patrice Godefroid. Model Checking for Programming Languages using Verisoft. pages 174–186. ACM Press, 1997. DOI: 10.1145/263699.263717.

[53] R. Govind, Frédéric Herbreteau, B. Srivathsan, and Igor Walukiewicz. Abstractions for the local-time semantics of timed automata: a foundation for partial-order methods. pages 24:1–24:14. ACM, 2022. DOI: `10.1145/3531130.3533343`.

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

[55] Thomas Haas, Roland Meyer, and Hernán Ponce de León. CAAT: consistency as a theory. *Proc. ACM Program. Lang.*, 6(OOPSLA2):114–144, 2022. DOI: `10.1145/3563292`.

[56] Thomas Haas, René Pascasl Maseli, Roland Meyer, and Hernán Ponce de León. Static analysis of memory models for SMT encodings. *Proc. ACM Program. Lang.*, 7 (OOPSLA2):1618–1647, 2023. DOI: `10.1145/3622855`.

[57] 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`.

[58] Henri Hansen. Abstractions for transition systems with applications to stubborn sets. volume 10160 of *Lecture Notes in Computer Science*, pages 104–123. Springer, 2017. DOI: `10.1007/978-3-319-51046-0\_6`.

[59] Henri Hansen, Shang-Wei Lin, Yang Liu, Truong Khanh Nguyen, and Jun Sun. Diamonds Are a Girl's Best Friend: Partial Order Reduction for Timed Automata with Abstractions. volume 8559 of *Lecture Notes in Computer Science*, pages 391–406. Springer, 2014. DOI: `10.1007/978-3-319-08867-9_26`.

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

[61] Fei He, Zhihang Sun, and Hongyu Fan. Satisfiability modulo ordering consistency theory for multi-threaded program verification. pages 1264–1279. ACM, 2021. DOI: `10.1145/3453483.3454108`.

[62] Fei He, Zhihang Sun, and Hongyu Fan. Deagle: An smt-based verifier for multi-threaded programs (competition contribution). volume 13244 of *Lecture Notes in Computer Science*, pages 424–428. Springer, 2022. DOI: `10.1007/978-3-030-99527-0_25`.

[63] Matthias Heizmann, Max Barth, Daniel Dietsch, Leonard Fichtner, Jochen Hoenicke, Dominik Klumpp, Mehdi Naouar, Tanja Schindler, Frank Schüssele, and Andreas Podelski. Ultimate automizer and the commuhash normal form - (competition contribution). volume 13994 of *Lecture Notes in Computer Science*, pages 577–581. Springer, 2023. DOI: `10.1007/978-3-031-30820-8_39`.

[64] 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`.

[65] Alex Horn and Jade Alglave. Concurrent kleene algebra of partial strings. *CoRR*, abs/1407.0385, 2014. URL http://arxiv.org/abs/1407.0385.

[66] Alex Horn and Daniel Kroening. On partial order semantics for sat/smt-based symbolic encodings of weak memory concurrency. volume 9039 of *Lecture Notes in Computer Science*, pages 19–34. Springer, 2015. DOI: `10.1007/978-3-319-19195-9_2`.

[67] Jeff Huang. Stateless model checking concurrent programs with maximal causality reduction. pages 165–174. ACM, 2015. DOI: 10.1145/2737924.2737975.

[68] Dominik Klumpp, Daniel Dietsch, Matthias Heizmann, Frank Schüssele, Marcel Ebbinghaus, Azadeh Farzan, and Andreas Podelski. Ultimate gemcutter and the axes of generalization - (competition contribution). volume 13244 of *Lecture Notes in Computer Science*, pages 479–483. Springer, 2022. DOI: 10.1007/978-3-030-99527-0_35.

[69] Bogdan Korel and Jurgen Rilling. Dynamic program slicing methods. *Information and Software Technology*, 40(11-12):647–659, 1998.

[70] Daniel Kroening, Subodh Sharma, and Björn Wachter. AbPress: Flexing Partial-Order Reduction and Abstraction. *CoRR*, abs/1410.6044, 2014.

[71] Will Leeson and Matthew B Dwyer. Graves-cpa: A graph-attention verifier selector (competition contribution). In *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, pages 440–445. Springer, 2022.

[72] 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.

[73] 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.

[74] Mangala Gowri Nanda and S. Ramesh. Slicing concurrent programs. pages 180–190. ACM, 2000. DOI: 10.1145/347324.349121.

[75] William T. Overman and Stephen D. Crocker. Verification of Concurrent Systems: Function and Timing. pages 401–409. North-Holland, 1982.

[76] 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.

[77] Ishai Rabinovitz and Orna Grumberg. Bounded model checking of concurrent programs. volume 3576 of *Lecture Notes in Computer Science*, pages 82–97. Springer, 2005. DOI: 10.1007/11513988_9.

[78] Cedric Richter, Eyke Hüllermeier, Marie-Christine Jakobs, and Heike Wehrheim. Algorithm selection for software validation based on graph kernels. *Autom. Softw. Eng.*, 27(1):153–186, 2020. DOI: 10.1007/S10515-020-00270-X.

[79] Marcelo Sousa. *Abstractions and independence.* PhD thesis, University of Oxford, UK, 2018. URL https://ethos.bl.uk/OrderDetails.do?uin=uk.bl.ethos.780457.

[80] Jie Su, Cong Tian, Zuchao Yang, Jiyu Yang, Bin Yu, and Zhenhua Duan. Prioritized Constraint-Aided Dynamic Partial-Order Reduction. pages 78:1–78:13. ACM, 2022. DOI: 10.1145/3551349.3561159.

[81] Jie Su, Zuchao Yang, Hengrui Xing, Jiyu Yang, Cong Tian, and Zhenhua Duan. Pichecker: A POR and interpolation based verifier for concurrent programs (competition contribution). volume 13994 of *Lecture Notes in Computer Science*, pages 571–576. Springer, 2023. DOI: 10.1007/978-3-031-30820-8_38.

[82] Zhihang Sun, Hongyu Fan, and Fei He. Consistency-preserving propagation for SMT solving of concurrent program verification. *Proc. ACM Program. Lang.*, 6(OOPSLA2): 929–956, 2022. DOI: `10.1145/3563321`.

[83] 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`.

[84] Antti Valmari. Stubborn sets for reduced state space generation. volume 483 of *Lecture Notes in Computer Science*, pages 491–515. Springer, 1989. DOI: `10.1007/3-540-53863-1_36`.

[85] Björn Wachter, Daniel Kroening, and Joël Ouaknine. Verifying multi-threaded software with impact. pages 210–217. IEEE, 2013.

[86] Chao Wang, Zijiang Yang, Vineet Kahlon, and Aarti Gupta. Peephole Partial Order Reduction. volume 4963 of *Lecture Notes in Computer Science*, pages 382–396. Springer, 2008. DOI: `10.1007/978-3-540-78800-3_29`.

[87] Liangze Yin, Wei Dong, Wanwei Liu, and Ji Wang. On scheduling constraint abstraction for multi-threaded program verification. *IEEE Trans. Software Eng.*, 46(5): 549–565, 2020. DOI: `10.1109/TSE.2018.2864122`.

[88] Rachid Zennou, Mohamed Faouzi Atig, Ranadeep Biswas, Ahmed Bouajjani, Constantin Enea, and Mohammed Erradi. Boosting sequential consistency checking using saturation. volume 12302 of *Lecture Notes in Computer Science*, pages 360–376. Springer, 2020. DOI: `10.1007/978-3-030-59152-6_20`.