MŰEGYETEM 1782

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

Partial Order Reduction for Abstraction-Based Verification of Concurrent Software in the Theta Framework

Bachelor's Thesis

Author
Csanád Telbisz

Advisor
Levente Bajczi

December 8, 2022

Contents

MŰEGYETEM 1782

SZAKDOLGOZAT FELADAT

Telbisz Csanád Ferenc
Mérnökinformatikus hallgató részére

Részleges rendezés redukció többszálú programok absztrakcióalapú formális verifikációjának támogatásához a Theta keretrendszerben

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. Egy módszer ezen probléma megoldására az absztrakció, melynek segítségével csoportosíthatóak a program állapotai, és ezzel lényegesen kisebb állapottér fölött szükséges csak a modellellenőrzőnek működnie. Ezt használja ki az ellenpélda alapú absztrakció finomítás (CEGAR) algoritmus, mely automatikusan meg tud találni optimális absztrakciós szinteket.

Többszálú programok esetén még rosszabbul skálázódik a modellellenőrzés, ezért specializált módszerek szükségesek a komplexitás leküzdéséhez. Egy ilyen módszer a részleges rendezés redukció (POR), mely ekvivalens lefutásokat nem fog redundánsan felderíteni.

Azonban tovább lehet optimalizálni a modellellenőrzés folyamatán ha a két megközelítést együtt alkalmazzuk, és a PORt a CEGAR által generált absztrakt állapottéren végezzük el. Ezen megközelítés kidolgozása és bemutatása a Hallgató feladata szakdolgozatának keretében.

A Hallgató feladatának a következőkre kell kiterjednie:

- Mutassa be a POR technikák absztrakcióalapú verifikációba történő integrálását
- Elemezze, hogyan lehet a CEGAR konfigurációs lehetőségeit kihasználva minél hatékonyabbá tenni a fenti megközelítést
- Implementálja a bemutatott algoritmusok prototípusát a Theta keretrendszerben
- Értékelje ki az implementált algoritmusok teljesítményét az SV-COMP verifikációs verseny benchmark készletén

Tanszéki konzulens: Bajczi Levente (doktorandusz)

Budapest, 2022.10.06.

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

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, 2022. december 8.

Telbisz Csanád
hallgató

Kivonat

A többmagos processzorok biztonságkritikus rendszerekben történő térhódításának köszönhetően egyre gyakrabban használnak többszálú programokat ilyen rendszerekben is, hiszen így lehet legjobban kiaknázni a párhuzamos számítás előnyeit. A szoftververifikáció komplexitása új szintre emelkedik a párhuzamosság megjelenésével a szálak nagyszámú lehetséges átlapolódása miatt. A komplexitásnövekedés eredménye, hogy a megfelelő tesztlefedettség elérése még nagyobb kihívást jelent, a naiv verifikációs technikák pedig gyakorlatilag használhatatlanná válnak. A részleges rendezés redukció (POR) hatékony modellellenőrzési megközelítés a párhuzamosság kezelésére. Az ellenpéldaalapú absztrakciófinomítás (CEGAR) pedig eredményes absztrakción alapuló technika állapot térben történő elérhetőségvizsgálatra.

A részleges rendezés alapú redukció aktívan kutatott területe az utóbbi évtizedeknek. Számos algoritmust publikáltak azzal a céllal, hogy minél nagyobb redukció által minél jobb teljesítményt érjenek el. Jelen dolgozatomban bemutatok néhányat a terület legmeghatározóbb algoritmusai közül. Ugyanakkor ezek a módszerek többnyire egy egyszerű állapottér bejárásra építenek csupán, ami korlátozza a további optimalizálási lehetőségeket.

Munkámban új megközelítését mutatom be a dinamikus POR technikák absztrakcióalapú verifikációba történő integrálásának. Az új módszer egy program utasításai között épített függőségi reláció számítása során az aktuálisan alkalmazott absztrakciót leíró információt is felhasználja. Ha két utasítás közti összefüggőség forrása el van absztrahálva, nyugodtan tekinthetjük ezt a két utasítást függetlennek. A modellbeli összefüggőség mértékének csökkenésével a POR nagyobb redukciót képes elérni. A CEGAR technikákat többféle módon is optimalizálhatjuk, például lusta kiértékeléssel. Dolgozatomban kitérek arra is, hogyan lehet a bemutatott absztrakciót figyelembe vevő POR algoritmust az állapottér lusta kiértékelésű számításával kombinálni. A bemutatott algoritmusok egy lehetséges alkalmazásaként vázolom, hogyan lehet adatváltozókat érintő versenyhelyzetek detektálásához POR alapú redukciót használni. Végül kiértékelem a prezentált algoritmusok teljesítményét.

Abstract

As multi-core processors gain popularity in safety-critical systems, multi-threaded programs are increasingly used in these systems to exploit their full potential. 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 become practically infeasible as a result of this complexity. Partial order reduction (POR) is an effective approach to handle concurrency in model checking. Counterexample-Guided Abstraction Refinement (CEGAR) is an efficient abstraction-based technique for checking reachability in a state space.

Partial order reduction has been an active field of study in recent decades. Several algorithms have been published with the aim of achieving better performance by greater reduction. Some state-of-the-art partial order reduction algorithms are presented in this report. Mostly though, these algorithms only assume a simple state space exploration which limit the possibilities for further optimization.

In this work, I present novel ways to integrate a dynamic partial order reduction algorithm into an abstraction-based verification process. Information is exploited about the applied abstraction when building a dependency relation on operations of a program. If the source of dependency between certain operations is abstracted away, they need not be considered dependent. By decreasing the dependency in the model, the reducing effect of partial order reduction is increased. Counterexample-Guided Abstraction Refinement (CEGAR) has several optimizations including lazy computation. I show how the proposed abstraction-aware partial order reduction algorithm can be combined with the lazy computation of the state space. As an application of the presented algorithms, I introduce how partial order reduction can be used for data race detection. Finally, I evaluate the performance of the proposed algorithms.

Chapter 1

Introduction

Rapid development in technology led to huge advancements in microprocessor systems. Today, multi-core processors are available for various targets from personal computers through smartphones to safety-critical systems. In a critical system, the increased computing capacity of a multi-core processor may add extra resources to the critical functionalities. This reason has lead to the increasing popularity of multi-core processors and multi-threaded programs in even critical systems.

Nonetheless, functionally correct behavior is still crucial in safety-critical systems. Although concurrency brings an additional complexity to the development, the need for safe operation and safety requirements remain a central element of critical systems.

Unfortunately, concurrent software design faces several difficulties. The development of concurrent software requires more prudence from developers as it is easier to overlook unintended behavior in a multi-threaded program. A concurrent program inevitably has a great number of possible thread interleavings. It can be challenging for a developer to consider all possible interactions of the threads.

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

Formal verification can prove safety guarantees for a system. Verification is a challenging task in itself, as the number of possible behaviours can be huge. The verification task is often to determine whether an error location can be reached in the program. Basically, this question can be answered by searching the state space of the program for an error state. Unfortunately, the number of states grows exponentially with the number of variables. This phenomenon is called the state space explosion problem [20].

An efficient approach to handle this vast complexity is *abstraction* [26]. By focusing on some parts of the problem while ignoring other details, we get a smaller representation of the problem. We may have a chance to solve the original problem by analyzing the abstract representation. If we fail to solve the problem using this representation, we can refine our abstraction by considering more details. CEGAR (Counterexample-Guided Abstraction Refinement) is an efficient abstraction-based model checking algorithm [19]. It follows this concept of iterative refinement. Abstraction can most efficiently be applied to data: the values of some variables can be represented by fewer equivalence classes [28].

Concurrency introduces a new level of complexity to software verification due to the great number of thread interleavings. By default, the whole state space has to be explored because a violation of the safety requirement may occur anywhere. Unfortunately, the size of the state space explodes exponentially due to the number of possible thread interleavings. Verification of concurrent programs has to deal with this complexity.

Partial order reduction (POR) is a widely known technique for handling concurrency in model checking [36]. The core concept of POR is to identify equivalent executions (traces). 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.

While partial order reduction is an effective technique for handling concurrency, abstraction is an efficient approach to handling data in model checking. This work aims to develop a highly performant verification algorithm by combining these two model checking paradigms. I integrate POR into a CEGAR-based model checking algorithm, and I show how these algorithms can be applied together.

I also present a novel algorithm that exploits the advantages of using POR in an abstraction-based context. The proposed method is called *abstraction-aware partial order reduction*, where the precision of the abstraction is used to boost the reduction power of POR. I defined a novel dependency relation in the abstract representation of the state space. The size of the new dependency relation is smaller than the size of the original relation. This allows POR to achieve more reduction and thus better performance. I show an approach to combine the new algorithm with the lazy extensions of CEGAR [28] to further increase the performance of the verification.

I have implemented and contributed the proposed methods to the open-source model checking framework Theta [38]. I compared the presented approaches to existing solutions on the widely-used SV-COMP benchmark programs (SV-COMP is a prestigious competition for software verification [10]). The introduced approach leads to performance gains on the benchmark problems compared to the traditional POR and CEGAR approaches.

The main scope of my work is error location reachability analysis. However, I briefly introduce how partial order reduction can be used for the verification of another type of safety requirement: data races [29]. Data races can occur in a program when non-atomic operations from different threads modify the same memory location. This leads to undefined behavior which is best avoided. For this problem, a slightly different formulation is needed.

This thesis is structured as follows. Chapter 2 introduces the essential concepts and definitions necessary for understanding this work. The basics of model checking are explained, along with a quick overview of CEGAR and POR. In chapter 3, the related work is presented. Chapter 4 explains how POR can be combined with CEGAR. First, the used POR algorithm is described in detail, along with its integration into CEGAR. Then, abstraction-aware partial order reduction is explained. The soundness of the algorithms presented in the chapter is proven. Some implementation details are also provided at the end of this chapter. Chapter 5 introduces how POR can improve data race detection. Chapter 6 evaluates the work. It starts with a case study, then the findings of benchmark tests are summarized. Finally, chapter 6 draws conclusions and proposes possible future works.

Chapter 2

Background

This report assumes that the reader is familiar with the basic concepts of concurrent software design and formal software verification. Nevertheless, to avoid the misunderstanding of used concepts and notions, definitions are introduced in this chapter.

2.1 Formal Representation of Software Programs

Though high-level languages (such as C) are convenient for developers, their verification would require a formal model of the language semantics, which can be quite complicated [9]. Thus, for verifying a program written in a high-level language, its source code is transformed into a low-level formalism that is easier to verify.

One such formalism is the *Control Flow Automaton* (CFA) [12].

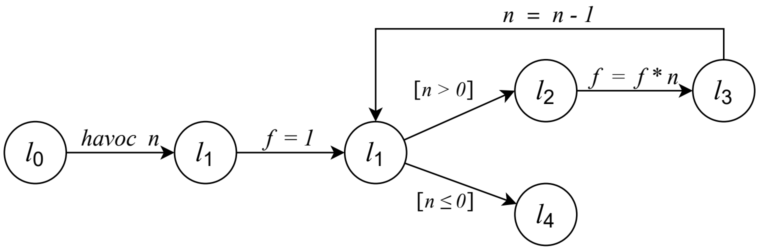
2.1.1 Control Flow Automata

A CFA represents a single-threaded program with the following semantics.

Definition 1 (Control Flow Automaton). A CFA is a tuple $\mathit{CFA} = (V, L, l_0, E)$, where:

- V is a set of variables (each $v \in V$ has a domain D_v: the possible values of v),
- L is a set of control locations (it can be considered as the possible values of the program counter),
- l_0 is the initial location,
- $E \subseteq L \times \mathit{OPS} \times L$ is the set of transitions. A transition is a directed edge in the CFA with a source control location, a target control location, and one operation. An operation ($\mathit{op} \in \mathit{OPS}$) can be:
  - a deterministic assignment of a variable ($v = \mathit{expr}$), where the value of the expression expr becomes the new value of the variable $v \in V$,
  - a non-deterministic assignment of a variable ($\mathit{havoc}\ v$), where the new value of the variable $v \in V$ can be anything from its domain D_v,
  - a guard condition ($[\mathit{cond}]$). A transition with a guard can only be executed if the guard expression is evaluated to true. ▪

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



(a) C source code

(b) CFA of the program

Figure 2.1: Small example to illustrate a CFA

Let us illustrate control flow automata with the following simple example.

Example 1. *The program in Figure 2.1a calculates the factorial of the given number: the value of variable* `f` *is* $n!$ *at the end of the execution of this program.*

Figure 2.1b depicts the CFA of this program. The edges of the CFA correspond to the operations of the program (including condition checks). l_0 *is the initial location. Note that a value from user input is assigned to* `n`*, which translates to the non-deterministic assignment* `havoc n`*.*

2.1.2 Formal Representation of Concurrent Programs

Since the threads of a multi-threaded program are like "single-threaded programs", which can be represented with a CFA, it is reasonable to use an extended form of the CFA to model concurrent programs: we can have a set of processes, where each process has its own CFA [8].

Definition 2 (eXtended Control Flow Automaton (XCFA)). An XCFA is a tuple $XCFA = (V_g, P)$, where:

- V_g is a set of global variables,
- P is a set of processes. A process is a tuple $p = (V_l, CFA)$, where:
  - V_l is a set of local variables,
  - CFA is a CFA (whose variables are $V \subseteq V_g \cup V_l$) extended with the following operations: *start thread* and *join thread*, *atomic begin* and *atomic end.*

The processes of an XCFA step (take a transition) *asynchronously.* ▪

A *start thread* operation creates a new process p_{new} (and marks $p_{new} \in P$ as an active process) and starts the concurrent execution of the new process at its initial CFA location. A *join thread* operation is disabled until the specified process p terminates: after p has terminated, the join thread operation can be fired. *Atomic begin*, and *atomic end* operations mark atomic blocks: while the execution of a process is inside an atomic block, all other processes are disabled. The semantics of these operations are explained in more detail in the next section.

2.1.3 State Space of a Program

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

2.1.3.1 Transition Systems

Transition systems have been defined variously over the years of model checking [7, 23]. In this report, the following definition is used:

Definition 3. 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 is a set of initial states. ▪

An action α is said to be *enabled* in a state s if there is a transition $t = (s, \alpha, s') \in T$ for some $s' \in S$. The following notations are used:

- $s \xrightarrow{\alpha} s'$ denotes the transition (s, α, s'),
- $\mathit{post}(s, \alpha) = \{s' \in S : \quad \exists (s, \alpha, s') \in T\}$, and
- $\mathit{enabled}(s)$ is used to denote the set of enabled actions in s.

A transition system is action-deterministic if $|I| \leq 1$ and $|\mathit{post}(s, \alpha)| \leq 1$ for any state $s \in S$ and action $\alpha \in A$ [7]. The state space of a program is not action-deterministic due to *havoc* statements ($|\mathit{post}(s, \mathit{havoc}\ \ \mathrm{x})| = |D_x|$), and uninitialized variables ($|I| > 1$ possibly). However, *unknown* is a possible value for variables when using abstraction (see details later), which means that an uninitialized variable or a variable after a non-deterministic assignment gets the specific value *unknown*. This way, the state space becomes action-deterministic.

Partial order reduction algorithms are formulated for action-deterministic transition systems as a common practice [6, 7, 22]. Sometimes, instead of using the term action-deterministic, it is said that control non-determinism is allowed [2]. Furthermore, partial order reduction can be applied even for non-action-deterministic systems, though that requires slightly different formulations [37].

2.1.3.2 State Space of a Control Flow Automaton

The state space of a program is a transition system that consists of all the possible and reachable states and transitions between them, as defined below.

A *state* of a CFA represents a control location and the values of the variables at a certain point during the operation of the program: $s = (l, d_1, d_2, ..., d_n)$, where:

- $l \in L$ is the location that the state represents,
- $d_1, d_2, ..., d_n$ are the values of the variables ($v_i = d_i$, $v_i \in V$, $d_i \in D_i$, $1 \leq i \leq n = |V|$).

A *state* of an $\mathit{XCFA} = (V_g, P)$ represents the control locations of all processes and the values of all variables (global and local variables) at a certain point during the operation of the program: $s = (l_1, l_2, ..., l_p, d_1, d_2, ..., d_n)$, where:

- $l_j \in L_{p_j}$ is the current location of process p_j, for $1 \leq j \leq p = |P|$
$(p_j = (V_{lp_j}, \mathit{CFA}_{p_j}),\ \mathit{CFA}_{p_j} = (V_g \cup V_{lp_j}, L_{p_j}, l_{p_j 0}, E_{p_j}))$,
- $v_i = d_i$, the current value of variable v_i, for $1 \leq i \leq n = |V|$
$(v_i \in V,\ d_i \in D_{v_i},\ V = V_g \cup (\bigcup_{p \in P} V_{lp}))$.

An *action* of a *transition* is an operation that the program executes. An action is enabled in a state if that operation can be performed in that state of the program. The process of an action refers to the process of the action's corresponding program operation: the process of an action α is denoted with p_α. A process is *active* or *enabled* in a state if it has any enabled actions in the state.

A transition with action α leads to the new state of the program after executing the operation represented by α. The location of the process of α is the source CFA location of α in the source state, and the target location of α in the target state. Multiple transitions can have the same action (e.g., x++ from a state where $x = 0$ or from another state where $x = 1$).

The operations of an (X)CFA manifest in different ways in the state space:

- For an assignment $s \xrightarrow{v=expr} s'$, the value of v in s' is the value of expression *expr* evaluated in s. The location of the statement's process is the source location of the statement in s and the target location in s'.
- For a *havoc v* statement, there are several transitions, $|D_v|$ exactly, leading to different states. The location of the statement's process changes with each transition as usual (the target CFA location of the statement appears in the target states of the new transitions). The value of v is different in each target state: the values range over the domain of v.
- An action with a guard condition $[\mathit{cond}]$ is enabled in each state s where the location of the action's process is the source location of the action, and expression *cond* evaluates to true in s.
- For a *start thread* action $s \xrightarrow{\text{start thread } p_{new}} s'$, the location of the parent process is as usual in s and s'. In s', a new field appears in the state description that stores the location p_{new}: the value is the initial location of the CFA of p_{new}. Also, new fields appear for each local variable of p_{new} with their initial value (if a local variable is uninitialized, there are several transitions for this start thread operation similar to a havoc statement).
- A *join thread p* operation is enabled in each state s where the location of the action's process is the source location of the action, and p is in its final location.
- An *atomic begin* action disables actions from other processes, that is, no action of another process is enabled in any state reachable from the target state of the atomic begin action until a transition with an *atomic end* operation. Actions from other processes may be enabled starting from the target of the *atomic end* operation.

An initial state of a program is a state where all processes are in the initial location of their main procedure. The values of the variables in an initial state can vary based on the language the program is written in. Uninitialized variables either contain memory garbage (as local variables in C [29]), resulting in several initial states per process, or they are initialized automatically to a default value (as in Java [30]), resulting in one initial state per process.

Since model checking includes searching the state space, the efficiency of a verification algorithm largely depends on the size of the state space, that is, on the number of control locations and variables in the program and the size of their domains. To represent even a single 32-bit integer variable, 2^{32} states would be necessary. With more variables, it would grow exponentially: this is called the *state space explosion problem* [20]. Thus, efficient algorithms are essential to overcome this problem.

2.2 Formal Verification

Formal software verification aims to prove certain properties of a program mathematically [18]. Among others, verified properties can be reachability criteria (whether a certain error state is reachable with any execution of the program), memory-safety (no memory leak or other memory handling issue), or the problem of termination (whether all executions of the program will terminate). In the scope of this work, reachability criteria are considered exclusively.

2.2.1 Model Checking

Model checking is a formal verification technique where properties are verified by analyzing the state space of the program [26]. In general, the input of a model checking algorithm is a *model* (here, an XCFA) and a *formal requirement*. The output of such algorithms is a verdict: the model is either *safe* (it is mathematically proven to be safe) or *unsafe* (a counterexample is provided where the requirement is violated).

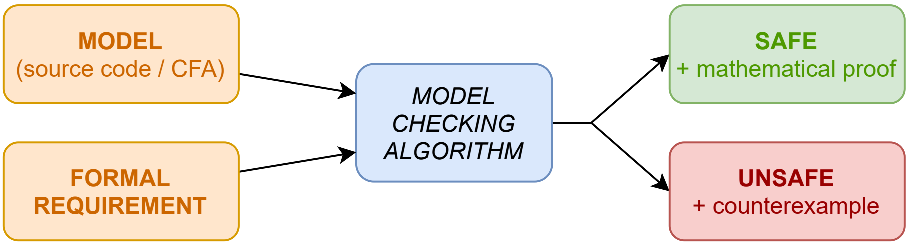


Figure 2.2: Model checking in general.

As for the formal requirement, in reachability analysis, certain points of the program under verification are marked as unsafe. If any possible program execution reaches one such point, the reachability criterion is said to be violated. In the introduced formalism, the (X)CFA, these marked points (locations) are called error locations. So the formal requirement is that no error location is reachable from the initial location(s) of the (X)CFA. A state is an error state in the state space of the program if its location is an error location. In the case of a multi-threaded program, a state is an error state if any of the program's processes is in an error location in that state.

The mathematical problem of model checking is undecidable. Consider any program with an error location at its exit point. To prove that this error location is unreachable is equivalent to answering whether this program always terminates. The termination problem is undecidable [39]. Verification techniques have to face this problem and provide usable algorithms that can verify as much software as possible.

2.2.2 Counterexample-Guided Abstraction Refinement (CEGAR)

CEGAR is an abstraction-based model checking algorithm [19]. It uses abstraction to handle the problem of state space explosion. CEGAR starts from a coarse abstraction of the problem and iteratively refines the abstraction until the problem can be solved. The more coarse the abstraction is, the more details are ignored. This way, there is a chance to answer the original problem by solving a much simpler abstract problem. If the abstract problem is too generic to provide an answer, the abstraction must be refined.

The core of the algorithm is the *CEGAR-loop* which consists of two main parts: the *abstractor* and the *refiner* (see Figure 2.3).

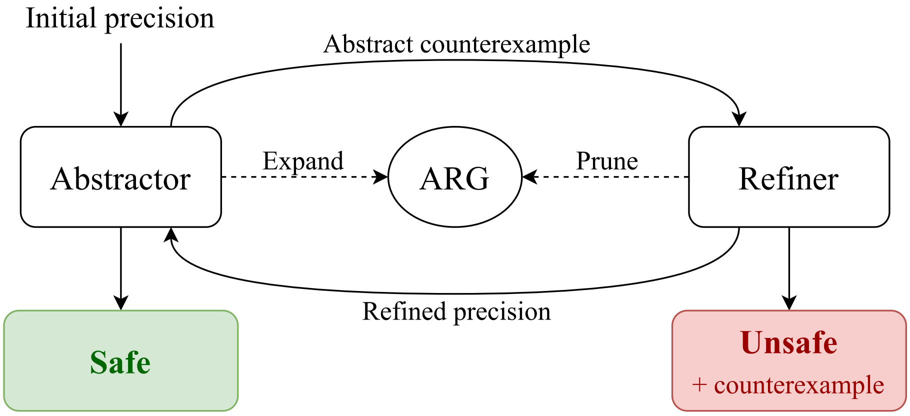


Figure 2.3: The CEGAR-loop.

The abstractor builds the abstract state space (in fact, an *abstract reachability graph*, ARG [14]) where abstract states consist of multiple concrete states. A concrete state is an error state if the control location of one of the processes is marked as an error location.[1] An abstract state is considered an abstract error state if it contains at least one concrete error state. The abstractor tries to prove that no abstract error state is reachable in the abstract state space. If no abstract error state is reachable, the algorithm terminates with a safe verdict since no concrete error state can be reached when its over-approximation is unreachable. If an abstract error state is reachable, the abstractor provides an abstract counterexample to the refiner.

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

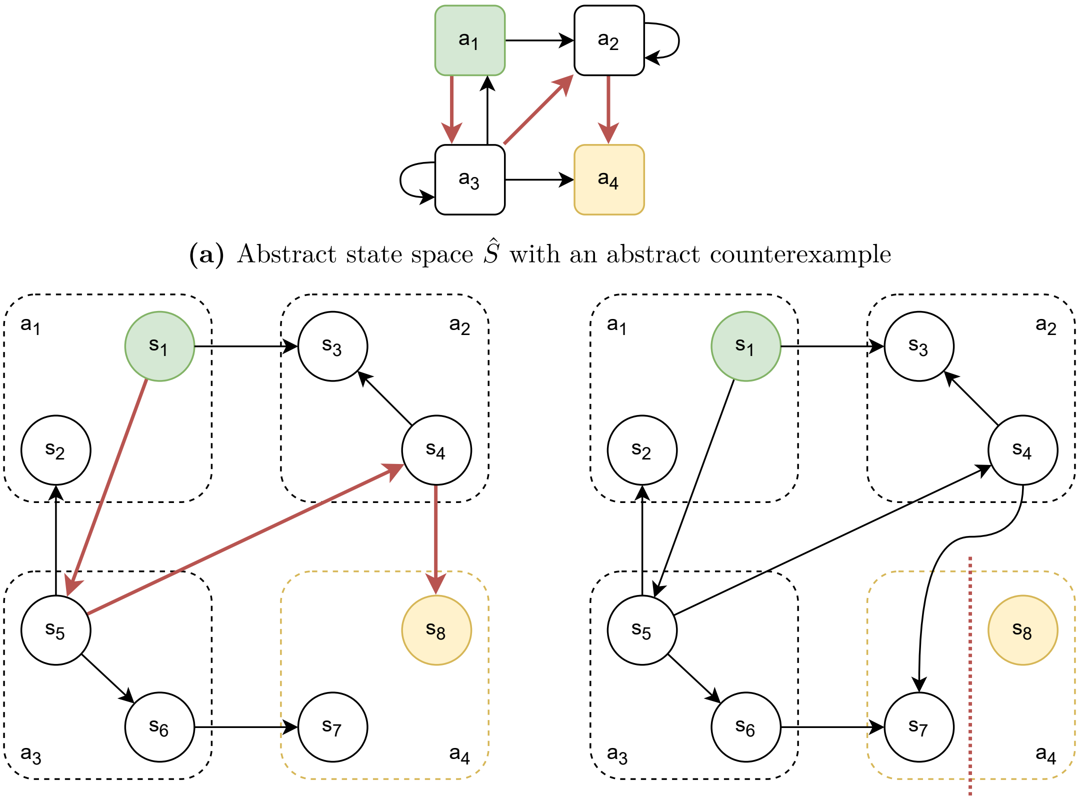
[1] Defining error states as states containing an error location is perfect for error location reachability analysis which is in the main focus of this work. However, for other formal requirements, another interpretation of error states is necessary (c.f. data races in Chapter 5).

In practice, when CEGAR is applied for software verification, information about data flow (e.g., values of variables) turned out to be most beneficial to abstract away [15]. Typical forms of abstraction are *explicit-value abstraction* [13] and *predicate abstraction* [25].

With explicit-value abstraction, the concrete values of certain variables are tracked while other variables are abstracted away. In the refinement step, new variables are added to the set of tracked variables. When evaluating an expression (e.g., a guard condition or the value for an assignment), untracked variables get an *unknown* value, meaning it can be anything from the domain of the variable. If the concrete value of the expression cannot be calculated due to unknown values, the value of the whole expression will be unknown.

Predicate abstraction keeps track of logical predicates about variables (e.g., $x = 1$ and $y > 0$). In the refinement step, a new set of tracked predicates is calculated. When evaluating an expression, the result will be unknown if the tracked predicates do not imply the expression.

The abstraction can be represented formally with an abstraction function [7]. The abstraction function is a function $f : S \to \hat{S}$ (where S is the set of concrete states and $\hat{S}$ is the set of abstract states)[2]. Multiple concrete states can be mapped to the same abstract state. The abstract state space over-approximates the concrete state space. An abstract state s_0' is initial if $f(s_0) = s_0'$ for the initial state s_0 of the concrete state space. If a transition (s_1, α, s_2) is in the concrete state space, there is a transition $(f(s_1), \alpha, f(s_2))$ in the abstract state space. An abstract state e' is an error state if there is a state $e \in S$ such that $f(e) = e'$ and e is an error state of the concrete state space.



(a) Abstract state space $\hat{S}$ with an abstract counterexample

(b) Feasible counterexample in S_1

(c) Spurious counterexample in S_2

Figure 2.4: CEGAR counterexamples

[2] In some cases in practice, a concrete state can be represented by multiple abstract states [15]: the abstraction function then maps to a set of abstract states.

Example 2. *Consider a model checking process where the abstractor provides the abstract counterexample highlighted in Figure 2.4a. This counterexample leads from the abstract initial state a_1 to the abstract error state a_4 in the abstract state space AS. The abstract state space is an over-approximation of the concrete state space. So the refiner has to decide whether the abstract counterexample is feasible or spurious.*

First, let us assume that the concrete state space abstracted by AS is S_1 from Figure 2.4b. 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 state s_1 leading to the error state s_8.

However, S_2 from Figure 2.4c can also be the concrete state space whose abstraction is AS. The counterexample is spurious now, as there is no path from s_1 to s_8 in S_2.

Let f_1 be the abstraction function $f_1 : S_1 \to \hat{S}$. $f_1(s_1) = a_1$, and s_1 is the initial state of S_1, so a_1 is initial in $\hat{S}$. If s_i is within the bounding box of a_j in Figure 2.4b, then $f_1(s_i) = a_j$. s_8 is an error state, so $f_1(s_8) = a_4$ is an abstract error state. Transition $(s_1 \to s_5)$ is in S_1, so the transition $(a_1 \to a_3) = (f_1(s_1) \to f_1(s_5))$ is in $\hat{S}$. Similarly, transition $(s_4 \to s_3)$ is in S_1, so the transition $(a_2 \to a_2) = (f_1(s_4) \to f_1(s_3))$ is in $\hat{S}$.

2.3 Partial Order Reduction (POR)

Generally, the execution order of operations from different threads is unspecified in a multi-threaded program. Thus, when such a program is verified, it is obviously insufficient to check only a single randomly chosen thread interleaving (consider the possible interleavings of the threads in Figure 2.5a: the printed result can be anything from {00, 01, 10, 11}).

A definitely correct approach is to check every possible execution. While it yields an accurate result, it suffers from the problem of combinatorial explosion. The intuitive idea to reduce the number of interleavings to check is that there are independent operations whose order of execution is irrelevant: their swapping (if they are neighbors) does not change the outcome. This way, executions can be grouped into *equivalence classes* [32, 23]. Any element of a class can be transformed into any other execution in the same class by only swapping independent neighbors. Then, it is enough to check only one execution from each equivalence class. This idea can be generalized to transition systems.

2.3.1 Dependency Relation

In the case of transition systems, the dependency relation used to be formulated on a general level [7]:

Definition 4. Let $TS = (S, A, T, I)$ be a deterministic transition system. For $s \in S$, $\alpha, \beta \in \mathit{enabled}(s)$ $(\alpha \neq \beta)$, actions α and β are *independent* in s if:

- $\beta \in \mathit{enabled}(\mathit{post}(s, \alpha))$ and $\alpha \in \mathit{enabled}(\mathit{post}(s, \beta))$, and
- $\mathit{post}(\mathit{post}(s, \alpha), \beta) = \mathit{post}(\mathit{post}(s, \beta), \alpha)$.

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

The first condition means that independent actions can neither disable nor enable each other. The second property states that independent actions are commutative. Sometimes, *dependency of transitions* is used in this report: by the dependency of transitions, dependency of their actions is meant.

It is rather impractical to check this definition of independence. Checking these conditions would require calculating the successor states of s after α and β and after β and α. This is exactly what partial order reduction tries to avoid. Fortunately, actions are program operations when our transition system models a program. Sufficient conditions can be given for two actions to be independent using the semantics of program operations [23]. Intuitively, when speaking about a multi-threaded program, two operations are independent if neither their *control part* nor their *data part* is in conflict. The following conditions formalize this intuition. Two actions α and β are independent if:

- α and β are not the actions of the same process, *and*
- the set of objects that are accessed by α is disjoint from the set of objects accessed by β.

Note: in our case, shared accessed objects (that operations from different processes, threads can access) are global variables, but in general, it could mean any object (e.g., a file). Also, note that special attention is needed at operations that create or destroy a process.

Independence could be defined more sophisticatedly, e.g., by distinguishing *read* and *write* operations on shared objects (two read operations on the same object could be considered independent) [33]. This way, the overall dependency between operations would decrease. At the same time, this work focuses on the basic concepts of partial order reduction and not on such enhancements.

It is easy to check that these conditions are sufficient indeed for two actions to be independent. An action α can only enable or disable another action β if either they are in the same process or α modifies the value of a global variable that β uses in its guard condition. In both cases, the actions are dependent based on the introduced conditions. As for commutativity, the swapping of two actions can only lead to different states if their sets of accessed objects are not disjoint: the actions are dependent according to the introduced conditions, again.

2.3.2 Partial Orders

Definition 5 (Partial Order, Total Order, Linearization). On a set S a relation $R \subseteq S \times S$ is a *partial order* if R is reflexive, antisymmetric, and transitive.

A partial order R is a *total order* if for all $s_1, s_2 \in S$ either $(s_1, s_2) \in R$ or $(s_2, s_1) \in R$.

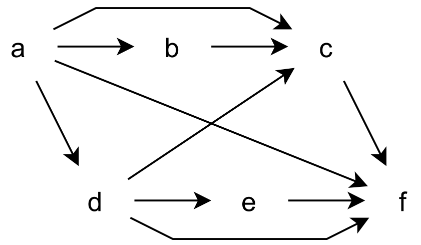
A *linearization* of a partial order R on S is a total order $R' \subseteq S \times S$ such that $R \subseteq R'$. ▪

A partial order R can be visualized by a directed graph whose vertices are the elements of the set S, and there is an edge from $s_1 \in S$ to $s_2 \in S$ if and only if $(s_1, s_2) \in R$.

A concrete execution (also called thread interleaving) of a program can be considered as a total order R on the set of operations where, for all $op_1, op_2 \in OPS, (op_1, op_2) \in R$ iff op_1 is executed before op_2. In the case of multi-threaded programs, a partial order can be associated to an execution using the concept of dependency where the partial order relation consists of the dependent ordered pairs of operations (operations are in execution order in the ordered pair). The concrete execution R is the linearization of this partial order. [23]

```
  Thread T1     |   Thread T2
a: x = 0;       | d: x = 1;
b: a = 1;       | e: b = 1;
c: print(x);    | f: print(x);
```

(a) Basic two-threaded program



(b) Visual representation of the partial order for execution $E = (a, d, b, e, c, f)$.

Figure 2.5: Multi-threaded program with a partial order for an execution

Example 3. *Let's take threads T_1 and T_2 from Figure 2.5a and the execution $E = (a, d, b, e, c, f)$. E is a total order on the set of the six operations.*

Operations a, d, c, f are dependent with each other since they all use the global variable x. Operations of the same thread are dependent by definition. The partial order with the dependent operation pairs can be seen in Figure 2.5b. Execution E is the linearization of this partial order.

$E' = (a, b, d, c, e, f)$ is also the linearization of the same partial order.

2.3.3 Partial Order Reduction Techniques

Executions - or generally transition sequences in a transition system - that are the linearizations of the same partial order yield the same result since dependency is completely "included" in the partial order. That is, partial order is the formalization of the *equivalence class* intuitively used in the introduction of this section. Such an equivalence class is called a Mazurkiewicz trace [32]. Any two transition sequences in a Mazurkiewicz trace can be obtained from each other by successively swapping adjacent independent actions. Therefore, it is sufficient to check a single transition sequence (linearization) from each Mazurkiewicz trace (partial order) in a verification process. This is the basic concept of *partial order reduction.* [23]

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

There are two main approaches to partial order reduction: *static* and *dynamic* POR [7]. In the static version, the model (e.g., the CFA of the program) is analyzed and the reduced state space (or its high-level description) is generated prior to the verification process. The dynamic approach constructs the reduced state space during the model checking. The latter's advantage is that it is not necessary to generate the entire state space, only the relevant part (that is actually needed in the verification). The abstraction-aware partial order reduction algorithm integrated into CEGAR presented in this report is inherently a dynamic approach since it uses on-the-fly information.

Chapter 3

Related Work

Partial order reduction has been a field of active research since the 1990s to this day [40, 23, 36, 7, 22, 1]. Algorithms evolved from basic solutions to proven optimal methods with several further optimizations. In this work, I use an early POR algorithm as a base since the focus is not on implementing a state-of-the-art algorithm but rather on developing a novel approach to the combination of partial order reduction (POR) and counterexample-guided abstraction refinement (CEGAR).

3.1 Traditional Partial Order Reduction (POR) Algorithms

Early partial order reduction methods build on the notion of *stubborn* [40], *ample* [7], *persistent*, and *sleep sets* [23]. These sets are associated with states: such states are subsets of the enabled actions in that state. The reduced state space is generated in a way that, from a state, only enabled actions in its stubborn/ample/persistent set are explored. It is proven that if a deadlock is reachable in the original state space, a deadlock can also be reached in the reduced state space. Therefore, it is sufficient to explore only the reduced state space.

Sleep sets are particularly useful in stateless model checking [24] where the visited states are not remembered. A sleep set is also associated to a state. An action α is put in the sleep set of a state s when we know that α would lead from s to an already explored state. Actions in the sleep sets are not explored. Sleep sets are orthogonal to persistent sets: they are used together to achieve more reduction.

3.2 State-of-the-Art POR Algorithms

Traditional POR algorithms approximated the conflicts between actions statically. Later, a dynamic partial order reduction (DPOR) algorithm was introduced, where the independence of actions is decided dynamically during the exploration [22]. DPOR first takes a (complete) execution, then marks *backtrack points* along this trace where dependency is detected. Actions that might lead to other non-equivalent traces are associated to a backtrack point. These actions have to be explored from the marked state. The algorithm continues to explore the state space until there is any unprocessed backtrack point.

Source DPOR from [1] is a dynamic partial order reduction algorithm that uses *source sets* instead of persistent sets. Each persistent set is a source set, but source sets are strictly smaller in some cases. This way, fewer executions are explored with source sets while reaching an equivalent result to the original problem. The presented source DPOR algorithm uses sleep sets, too.

Optimal DPOR [1] extends the Source DPOR algorithm with a construct called *wakeup tree*, which replaces the backtrack set of actions introduced in DPOR [22]. In simple DPOR, only single actions are added to backtrack sets. Here, action sequences are associated with backtrack points: these are wakeup trees. Exploration is only performed along the associated action sequences from backtrack points. Optimal DPOR is proven to be optimal: the minimal number of interleavings are explored in every case (that is no equivalent executions are explored). Since it has been published, Optimal DPOR has been extended with several enhancements [5, 31].

3.3 Conditional Independence

Initially, the independence relation of actions has been approximated statically by analyzing the transitions in the model [7, 23]. As a result, two actions that are dependent in some contexts will be handled as dependent in all possible contexts. However, several POR algorithms retrieve information from the search context: actions are considered dependent only in certain states under certain conditions [4, 5, 42].

In [42], a *guarded independence relation* is introduced where a condition is associated with each pair of actions meaning that the two actions are independent in any state where the condition holds. As an example, take two actions α and β where α reads the value of variable x while β assigns a value to x in the form of $x := v$. α and β are guarded independent with respect to $x = v$, meaning that α and β are independent in any state where $x = v$ holds (obviously, β does not change x if its value is already v). It could be said that the abstraction-based POR proposed in this work uses a guarded independence relation where the condition for two actions using the same variable x is "variable x is abstracted away in the current abstraction". At the same time, it is computationally simpler to check during the dependency calculation whether a variable is abstracted away. So in the algorithm presented in this work, the condition for guarded independence is only implicitly used.

In [5], an extension of optimal DPOR is presented: optimal DPOR with observers. The independence of actions is conditional to future actions called *observers*. For actions α and β, which both write the shared variable x, γ is an observer if it is a possible future action that reads the value of x. If there is no observer for α and β (i.e., x is unused later), α and β can be considered independent. Also, consider the situation where we have n processes $p_1, p_2, ..., p_n$, each with the single action $x := i$ (for p_i) and a safety requirement on x after joining all processes. The order of processes before the last one is irrelevant since the last process will overwrite the value of x anyway. So instead of $n!$ possible interleavings, it is sufficient to check n (where the last process is different in each trace). Optimal DPOR with observers achieves further reduction in these scenarios. Again, abstraction-based POR could be an extension of observers where any read operation on a variable x that is abstracted away is not an observer of x. Similarly, it would mean a considerable and redundant computational overhead to realize abstraction-based POR using observers compared to the method presented in the next chapter.

Context-sensitive DPOR [4] is another extension of optimal DPOR [1], which uses conditional independence, though implicitly. Instead of associating conditions to action pairs, it checks state equivalence during the state space exploration. Sleep sets are modified so that not only can single actions be added to a sleep set, but also sequences of actions to avoid exploring that sequence. Context-sensitive POR would be capable of recognizing executions that are equivalent only in the current abstraction because this algorithm is defined on a more general level. At the same time, it would only realize that two such executions are equivalent in the "last moment", just before the two traces reach the same state. The proposed abstraction-based POR algorithm knows it when the two traces diverge. Thus, context-sensitive POR has to explore more states to discover the equivalence of executions.

3.4 POR Combined with CEGAR

Some partial order reduction algorithms, such as sleep set techniques, are primarily useful in stateless model checking. (Sleep sets aim to avoid exploring the same state several times: this can be easily achieved in stateful model checking by consulting the list of visited states.) CEGAR is inherently a stateful model checking paradigm, so these methods provide less reduction. On the other hand, other POR algorithms are similarly advantageous in stateful as in stateless model checking, such as a persistent set technique where complete branches of the state space can be ignored.

CPAchecker is an open-source configurable program verification framework that supports several analysis techniques, including CEGAR and partial order reduction [11]. However, the POR algorithm applied in CPAchecker is relatively simple: only thread-local operations are considered independent (where an operation is global if it accesses a global memory location and thread-local otherwise). That is, the application of partial order reduction is orthogonal to CEGAR in CPAchecker.

In [41], an abstraction-based verification (though the *Impact* algorithm, not CEGAR) is combined with a dynamic partial order reduction algorithm. Although they use conditional dependency, it is similar to the guarded independence relation described in [42], and they do not exploit information about the applied abstraction to reduce dependency.

Chapter 4

Partial Order Reduction for Abstraction-Based Verification

This chapter describes how partial order reduction can be integrated into a CEGAR-based model checking algorithm. As the reduction of the state space is done during the verification process, it is a dynamic approach here, even though the used POR algorithm is more similar to static approaches [23] than the dynamic methods [1, 22] in the literature.

The novelty of the proposed algorithm lies in using extra information about the actual abstraction used in CEGAR when applying partial order reduction. This information is only available on-the-fly: that is why the presented algorithm is dynamic. Furthermore, this abstraction-aware extension of POR is orthogonal to the underlying algorithm: any dynamic POR method could be used.

4.1 Combining POR with CEGAR

In CEGAR, instead of the concrete state space of a program, an abstract state space is explored. So, partial order reduction is applied in the abstract state space.

4.1.1 Source Sets

In Section 2.3.3, it has been introduced that partial order reduction techniques work by calculating sufficient subsets of outgoing transitions to explore for each state. For this work, I adapt source sets from [1, 2]. Before defining source sets, a few notations are introduced.

A transition sequence $w = t_1 ... t_k$ is an execution from s, if there are states $s_1, ..., s_k$ and transitions $s \xrightarrow{t_1} s_1$, $s_{i-1} \xrightarrow{t_i} s_i$ (for $1 < i \leq k$) in the transition system. $s \xrightarrow{w} s'$ means that starting from state s, and taking all the transitions in w, state s' is reached. 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. The concatenation of transitions or transition sequences is denoted by $w.t$ or $w.v$: for $w = w_1 ... w_m$, $v = v_1 ... v_n$, and transition t, transition sequence $w.t = w_1 ... w_m t$, and $w.v = w_1 ... w_m v_1 ... v_n$.

Transition sequences w_1 and w_2 are in the same equivalence class (Mazurkiewicz trace) in a state s, if $s' = s''$ for $s \xrightarrow{w_1} s'$ and $s \xrightarrow{w_2} s''$. This is denoted by $w_1 \simeq_s w_2$.

If transition sequences w_1 and w_2 can be completed (by concatenation) to equivalent sequences from state s, it is denoted by $w_1 \sim_s w_2$. That is, $w_1 \sim_s w_2$ if there are transition sequences v_1, v_2 such that $w_1.v_1 \simeq_s w_2.v_2$.

4.1.1.1 Notion of Source Sets

A source set is a subset of enabled actions in a state of a transition system: it is sufficient to explore this subset in a software verification process [2]. This section introduces the exact definitions concerning source sets.

Intuitively, a subset P of enabled actions in a state s is a *source set* if for each execution w from s there is an action $\alpha \in P$ such that the first transition in w that is dependent with α is a transition with α as its action.

Definition 6 (Weak Initials). Let s be a state, and w be a transition sequence from s. For w, the set $WI_s(w)$ of weak initials in s is a set of actions: $\alpha \in WI_s(w)$ iff $\alpha \sim_s w$. ▪

Note, that $\alpha \in \mathit{enabled}(s)$, since $\alpha \sim_s w$ requires that α and w can be executed from s.

The intuitive idea behind weak initials is that in a state s, we can choose an action α for a sequence w from its weak initials, explore a transition with action α and avoid exploring w from s. Then, we can still reach the same state from $\mathit{post}(s, \alpha)$ as we would reach from s by w. This idea is formulated by the following definition of source sets and Theorem 2 in Section 4.1.1.3 from [2].

Definition 7 (Source set). Let s be a state. A set $P \subseteq \mathit{enabled}(s)$ is a *source set* in s if for each transition sequence w from s, $WI_s(w) \cap P \neq \emptyset$. ▪

Example 4. *Let us have the example from Figure 4.1, and let s be the initial state. Let the value of each variable be 0 in s.*

For $w = \alpha\beta\gamma\delta$, $WI_s(w) = \{\alpha, \beta, \delta\}$. To see this, let us check that each element of $WI_s(w)$ can be extended to an execution equivalent with w. For α, w is a trivial choice: it starts with α, and it is equivalent with itself. For β, we can choose $w' = \beta\alpha\gamma\delta$ since $w' \simeq_s w$ (the processes are in the same location after w and w', and the values of all variables are the same). As for δ, it can be extended to $\delta\alpha\beta\gamma$ which is equivalent with w.

Now, let us see $v = \beta\gamma\alpha\delta$: $WI_s(v) = \{\beta, \delta\}$. α is not a weak initial of v in s, because we cannot find a suffix v' for α such that $\alpha.v' \simeq_s v$. After v, the value of variable a is 2. Starting with α, the value of a is always 0 (the initial value of x), no matter what actions are executed after α.

As a consequence $\{\alpha\}$ is not a source set in s, because $WI_s(v) \cap \{\alpha\} = \emptyset$. On the other hand, $\{\beta\}$ or $\{\alpha, \delta\}$ are source sets in s. Unfortunately, to see this using the definition, we would need to check all executions from s whether there is an equivalent execution for each of them starting with an element of our source set. In this small example, we could do that, but in practice, we need a better method to compute source sets.

Thread t_1	Thread t_2	Thread t_3
(α) `a = x;`	(β) `b = 1;`	(δ) `c = 1;`
	(γ) `x = 2;`	

Figure 4.1: Small example to illustrate source sets

4.1.1.2 Computing Source Sets

As we have seen in Example 4, using the definition to calculate source sets means checking all executions from a state. This is exactly what partial order reduction tries to avoid: exploring all executions in the state space. So we need a method for computing source sets that is applicable in practice. The applied algorithm is similar to Overman's algorithm [35, 23]. Before presenting the algorithm that calculates source sets, the following notions are introduced (similar concepts can be found in [2]):

Definition 8 (May-enabled action in a state). An action α is *may-enabled* in a state s, if $\alpha \in \mathit{enabled}(s)$ or α can become enabled after a sequence of transitions from processes other than p_α. ▪

Definition 9 (Future actions). $\mathit{future\_actions}(s, \alpha)$ is a set of actions: $\beta \in \mathit{future\_actions}(s, \alpha)$ 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 $\beta_0 \notin \mathit{enabled}(s)$. ▪

An action $\alpha \notin \mathit{enabled}(s)$ can be *may-enabled* in a state s in two ways:

- The source CFA location of α is the location of p_α in s, but α is disabled for some reason. In practice, this can happen, if α is the next action of p_α and α is a join operation on a process p that has not reached its final location in s (α gets enabled when p terminates); or α has a guard condition $[c]$ that evaluates to false in s (α gets enabled when actions from other processes change the values of variables used by c so that c evaluates to true).
- α is the first action of a process (the source location of α is the initial location of the CFA of p_α) that has not been started as of s (α gets enabled when its process is created and started by another process).

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 compute an over-approximation of $\mathit{future\_actions}(s, \alpha)$ without exploring the state space by analyzing the static model of the program. Initially, the actions of the process p_α of α are collected with a graph search of the CFA of p_α. Another action β that is may-enabled in s can be enabled by an action reached in the CFA of p_α. Then, $\mathit{future\_actions}(s, \beta)$ is called recursively to collect more future actions. Algorithm 1 demonstrates the algorithm for computing *future_actions*.

Algorithm 1: Computing *future_actions*

```
Input: $s$, $\alpha$
Output: $FA$                                   /* Set of future actions */
$p_\alpha \leftarrow$ process of $\alpha$
$FA_{p_\alpha} \leftarrow$ reached actions with a graph search of the CFA of $p_\alpha$ from $\alpha$
$FA \leftarrow \{\}$
foreach may-enabled $\beta$ in $s$, $\beta \notin enabled(s)$, $\beta$ may be enabled by a $\gamma \in FA_{p_\alpha}$ do
    $FA \leftarrow FA \cup future\_actions(s, \beta)$
end
$FA \leftarrow FA_{p_\alpha} \cup FA$
```

On the implementation side, we have the below three cases to check whether the may-enabled action β can be enabled by a reached action γ. If any of them is true, $\mathit{future\_actions}(s, \beta)$ is calculated.

- γ starts the process of β.
- γ terminates its own process p_γ and β is a join operation on p_γ.
- γ writes a variable that β uses in its guard condition.

This computation of *future_actions* based on the static analysis of the model is an over-approximation, because each branch of the CFA is searched in line 2, even though some branches might not be reachable in the state space.

There is a minor nuisance because a process may start another process with the same CFA. This would result in endless recursion in the presented algorithm. Fortunately, this problem can easily be eliminated by passing the already reached actions in the recursive call of *future_actions*: these actions are excluded from the graph search. For the sake of conciseness though, I have neglected these details from the description of Algorithm 1.[1]

With the help of *future_actions*, we can compute source sets. The enabled actions in the current state s ($\mathit{EA} = \mathit{enabled}(s)$) are provided as an input to Algorithm 2 along with the initial actions (IA) that are initially put in the source set(-to-be) P. As long as any new action is added to P, the following is repeated: $\mathit{future\_actions}(s, \alpha)$ is calculated for each $\alpha \in \mathit{EA} \setminus P$. If there is any action $\beta \in \mathit{future\_actions}(s, \alpha)$ that is dependent with an action $\gamma \in P$, α is added to P.

A source set is computed with Algorithm 2 starting from the enabled actions per process.[2] That is, the set of enabled actions of a single process is extended to be a source set with enabled actions from other processes. This is repeated for each process. One source set with minimal size is chosen from the calculated source sets and returned as the final source set of s (this is a simple heuristic for choosing from multiple possible source sets to achieve the most possible reduction).

[1] Alternatively, an iterative approach can be used: in fact, an iterative approach has been implemented. On the other hand, it is easier to present and understand the algorithm in the recursive way.

[2] Source set calculation could start from a single action, but actions of the same process would be added to the set anyway: that extra calculation can be spared.

Algorithm 2: Calculating a Source Set from State s

```
Input: $s$, $EA = enabled(s)$, $IA \subseteq EA$      /* $IA$: initially added actions */
Output: $P$                                 /* Source set containing $IA$ */
$P \leftarrow IA$
$newAdded \leftarrow True$
while $newAdded$ do
    $newAdded \leftarrow False$
    $toAdd \leftarrow \{\alpha : \alpha \in EA \setminus P,\ \exists\beta \in future\_actions(s, \alpha), \exists\gamma \in P$
                                        such that $\beta$ and $\gamma$ are dependent$\}$
    if $toAdd \neq \emptyset$ then
        $P \leftarrow P \cup toAdd$
        $newAdded \leftarrow True$
    end
end
```

Theorem 1. *A set P returned by Algorithm 2 for a state s, $EA = enabled(s)$, and $\forall IA \subseteq EA$ is a source set in s.* ▪

Proof. Let us check the definition of source sets, that is, for each execution from s, one of its weak initials is in P. Let $w = w_1 ... w_n$ be a transition sequence from s. We have two cases:

I. $\exists w_i \in P$ 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 $1 \leq j < f$.

All such w_j is independent with w_f. To see this, assume the opposite: there is a w_d dependent with w_f, 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 2 would have added α to P based on the condition in line 5 (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, $w_f.(w \setminus w_f) \simeq_s w$. This implies that $w_f \sim_s w$, which means by definition that $w_f \in WI_s(w)$. So one of the weak initials of w is in P, indeed.

II. $\nexists w_i \in P$

This supposition implies that all actions in w are independent with all actions in P. Assume the opposite: there is a w_d for some $1 \leq d \leq n$, and $\gamma \in P$ such that w_d and γ are dependent. Reasoning is similar to case I. Since $\nexists w_i \in P$, $w_d \in \mathit{future\_actions}(s, \alpha)$ for some $\alpha \in \mathit{enabled}(s) \setminus P$. Algorithm 2 would have added α to P based on the condition in line 5, which did not happen.

Since all actions in w are independent with all actions in P, any $\alpha \in P$ is a weak initial of w: $\alpha.w \simeq_s w.\alpha$, which implies that $\alpha \sim_s w$. This means by definition that $\alpha \in WI_s(w)$, indeed. So one of the weak initials of w is in P, again. □

Now, we have an algorithm that calculates source sets. The next section explains how source sets can be used for partial order reduction.

4.1.1.3 Source Set Selective Search in CEGAR

In CEGAR, the abstract state space is built by the abstractor in an *expand* operation. Let S denote the states of the abstract state space and $S_E \subseteq S$ the set of expanded states. A not-yet-expanded state $s \in S \setminus S_E$ is chosen based on a search strategy (e.g., BFS, DFS, or A* with some sophisticated heuristics) and the selected state s is expanded. That is, the enabled actions in s are collected, and their targets (if not already in S) are added to the abstract state space as new states, and a new transition is added for each enabled action from s to the new state.[3]

The above way, the abstract state space is fully discovered. That is what POR is about to prevent. The POR algorithm applied here filters the enabled actions and only expands the abstract state space with the filtered subset of enabled actions and their successor

[3] The construction of the abstract state space (the *Abstract Reachality Graph* or ARG exactly) is slightly more complex in CEGAR. States can *cover* each other, and it is unnecessary to expand covered states. Nevertheless, covering does not influence POR since POR works in the *expand* operation. See [21] for more details about covering.

states. This filtered subset is a source set in each state.[4] The abstraction used in CEGAR preserves the CFA locations and actions in the abstract state space, only the values of variables can be abstracted away. Thus, source sets can be directly calculated using the actions of the program (the CFA).

Theorem 2 in [2] proves that if we generate a reduced state space by exploring only source sets of enabled actions in each state, for all executions in the original state space, there will be an equivalent one in the reduced state space. By default, source set selective search works for acyclic state spaces. However, with a minor supplement, cycles can be handled as well. Theorem 2 from [2] is the following (the theorem is proven in [2]).

Theorem 2. Let S be the original state space, and S_R be the reduced 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 in s,
2. for each cycle in S_R, if an action α is enabled in all states of the cycle, then α must be explored from some state of the cycle,

then for each state s in S_R and execution w from s in S, there is an execution w' in S_R such that $w.v \simeq_s w'$ for some transition sequence v. ▪

Note that states reachable from an error state are error states (if we reach an error location in a process p, we may execute further operations on other processes, p remains in the error location since there are no available operations from an error location). So if w leads to an error state, $w.v$ ends in an error state as well. This way, Theorem 2 states that if an error state can be reached in the original state space, an error state (potentially another one) is also reachable in the reduced state space discovered with a source set selective search. Thus, reachability analysis performed in the reduced and the original abstract state space yields equivalent results.

To satisfy the second condition concerning cycles, we have to detect cycles in the state space. Cycles can be detected by performing a depth-first search from the initial state. With DFS, edges of a graph are classified into *tree*, *forward*, *back* and *cross* edges. Each cycle contains a *back edge*. If a back transition starting from state s is encountered in the state space, all enabled actions are explored from s. This guarantees the second condition of Theorem 2: any action that is enabled in all states of a cycle is explored from some state of the cycle.

Back transitions could be detected during the exploration of the state space, on-the-fly. However, it would require a depth-first search of the state space (a similar approach can be found in [23]). To leave the possibility for other search strategies (e.g., BFS), back transitions can be calculated differently. A sufficient method to decide whether a transition $t = (s, \alpha, s')$ is a back transition is the following: if the program operation of α is represented by a back edge in the CFA of the process, t is considered a back transition. Note that states are partly characterized by CFA locations: without a back CFA edge, we could never "get back" to a previously visited state. Furthermore, back edges of the CFA can be calculated once at the beginning of the whole model checking process.

[4] It turns out that the filtered subset of enabled actions is not necessarily a source set in an abstract state. The proof of the soundness of the presented algorithm in Section 4.1.2 will explain this in more detail.

4.1.2 Soundness of the combination of POR and CEGAR

In this section, the soundness of using partial order reduction with CEGAR in the presented way is proven. To prove the correctness of the combination of CEGAR and POR, we have to check that if an error state is reachable in the concrete state space of the program, an abstract error state is reachable in the POR-reduced abstract state space. That is, we have to prove that the composition of the POR and the CEGAR transformation is *error-preserving.*

We do not have to handle the case where no error state can be reached in the concrete state space. In this case, the abstract state space can contain an abstract error state, and the abstractor of CEGAR may find a spurious counterexample in the abstract state space. However, the refiner checks the counterexamples and refines the abstraction if necessary. That is, CEGAR cannot produce a *false positive* (return an unsafe verdict when the program is safe). The algorithm may not terminate, but we have seen in Section 2.2.1 that the model checking problem is undecidable as a mathematical problem.

Theorem 3. Let the abstract state space S_A be the result of an abstraction applied in CEGAR on the concrete state space S. Let the reduced abstract state space S_{AR} be obtained from the original abstract state space S_A by only exploring actions returned by Algorithm 2 from each abstract state (all enabled actions are explored when one of them appears on a back transition).

If an error state is reachable from the initial state of the concrete state space S, an abstract error state can be reached in the reduced abstract state space S_{AR}. ▪

Proof. Technically, the concrete state space is mapped to the abstract state space, then to the reduced state space with POR (1). For the sake of the proof, let us consider the composition of the two transformations in a reversed order (2). That is:

$$\begin{aligned} POR \circ abstraction: &\quad S \xrightarrow{abstraction} S_A \xrightarrow{POR} S_{AR} \quad (1)\\ abstraction \circ POR: &\quad S \xrightarrow{POR} S_R \xrightarrow{abstraction} S_{RA} \quad (2) \end{aligned}$$

The proof proceeds by checking that $S \xrightarrow{POR} S_R$ and $S_R \xrightarrow{abstraction} S_{RA}$ are error-preserving transformations; it is also shown that $S_{RA} \subseteq S_{AR}$. This proves that if an error state is reachable in S, then an abstract error state is reachable in S_{AR} (which is the statement of the theorem).

Let f be the abstraction function of the abstraction $S \xrightarrow{abstraction} S_A$. Let $P_{f(s)}$ denote the set of actions to explore returned by Algorithm 2 for the abstract state $f(s)$.

Let us perform the $S \xrightarrow{POR} S_R$ (theoretical) state space reduction so that the explored actions from a concrete state s is $P_s = P_{f(s)} \cap enabled(s)$.

To see that P_s is a source set in s, note that $enabled(s) \subseteq enabled(f(s))$ since by definition of the abstract state space, if a transition (s, α, s') is in the concrete state space, the transition $(f(s), \alpha, f(s'))$ is in the abstract state space. Now, call Algorithm 2 with P_s as the initial actions input of the algorithm (the other two inputs are s and $enabled(s)$). The dependency relation used by the algorithm is a valid dependency relation in S. Therefore, based on Theorem 1, the returned set P'_s is a source set in s. Let us assume that $P'_s \neq P_s$, i.e., the algorithm has added at least one action $\alpha \in enabled(s) \setminus P_s$. But $\alpha \in enabled(f(s))$ as well, and $P_s \subseteq P_{f(s)}$, so Algorithm 2 would have had to add α to $P_{f(s)}$ when we used the algorithm to calculate $P_{f(s)}$: since $future\_actions(s, \alpha) \subseteq future\_actions(f(s), \alpha)$, if α is selected in line 5 of Algorithm 2 during the calculation of P'_s, α is also selected during the calculation of $P_{f(s)}$. This is contradiction because α has not been added to $P_{f(s)}$. Thus, $P'_s = P_s$, which implies that P_s is a source set for any state $s \in S$, indeed.

So the explored actions in S_R form source sets in each state of S_R: we can obtain S_R from S with a source set selective search. The correctness of a source set selective search is proven in [2], so $S \xrightarrow{POR} S_R$ is an error-preserving transformation.

Let us perform the transformation $S_R \xrightarrow{abstraction} S_{RA}$ with the same abstraction function f. Abstraction is error-preserving: if there is an error state $e \in S_R$, then its abstract state $f(e) \in S_{RA}$ is an abstract error state by definition.

Now, let us see that $S_{RA} \subseteq S_{AR}$, that is any state or transition present in S_{RA} is present in S_{AR} (in other words, if a state s is reachable in S_{RA} from its initial state, s is reachable in S_{AR} from its initial state). Let s_0 be the initial state of S. Then $s_0 \in S_R$, $f(s_0) \in S_{RA}$, $f(s_0) \in S_A$ and $f(s_0) \in S_{AR}$. If a state s' is reachable in S_{RA}, there is a state $s \in S_R$ so that $f(s) = s'$ and s is reachable in S_R. Since $P_s \subseteq P_{f(s)}$ for any state $s \in S_R$, if a state s is reachable in S_R, $f(s)$ is reachable in S_{AR}. We got that if a state s' is reachable in S_{RA}, s' is also reachable in S_{AR}, so $S_{RA} \subseteq S_{AR}$ holds indeed.

Let $E(SP)$ denote the set of error states reachable in a state space SP from its initial state. The statement of the theorem is the following: $\exists e \in E(S) \implies \exists e' \in E(S_{AR})$.

$S \xrightarrow{POR} S_R$ being an error-preserving transformation means $\exists e \in E(S) \implies \exists e' \in E(S_R)$. The abstraction function maps e' to the abstract state $f(e') \in S_{RA}$ which is an error state in S_{RA} (by definition of abstract error states): $f(e') \in E(S_{RA})$. Based on $S_{RA} \subseteq S_{AR}$, $f(e') \in E(S_{AR})$ which proves the theorem. □

Note that the proof does not assume that the used dependency relation of actions is valid in the abstract state space S_A. For certain concrete types of abstraction (e.g., explicit-value abstraction), it could be easily proven that the dependency relation is valid in the abstract state space. Then, we would not need the reversed order of transformations ($abstraction \circ POR$) for the proof because we could simply say that S_{AR} is obtained from S_A with a source set selective search which is proven to be error-preserving. However, without any assumption on the abstraction function, the dependency relation is not necessarily valid in S_A, and S_{AR} is not necessarily the result of a source set selective search of S_A. This way, the proof shows that the above combination of CEGAR and POR is correct independent of the type of the used abstraction.

4.2 Abstraction-Aware Partial Order Reduction

The previous sections of this chapter introduced the combination of a traditional partial order reduction algorithm with a CEGAR-based model checking algorithm. However, this integration is rather loose so far: the point has been identified in CEGAR (i.e., the expansion of the abstract state space) where POR can be applied, but the two algorithms have "no further contact".

In this section, a novel approach of integrating POR with CEGAR is presented where POR uses extra information from the current state of the CEGAR algorithm. I refer to this approach as *abstraction-aware partial order reduction (AAPOR).*

4.2.1 Basic Concept and Motivation

In Section 2.2.2, two common forms of abstraction has been introduced: explicit-value abstraction and predicate abstraction. The information describing an abstraction is called precision. In case of explicit-value abstraction, the precision is the set of tracked variables. The precision of predicate abstraction is the set of tracked predicates.

Let us use the term *abstract variable* for an element of the precision. In explicit-value abstraction, a tracked variable is an abstract variable, while in predicate abstraction, a predicate is an abstract variable. The notation $\chi \in \Pi$ is used to denote that abstract variable χ is present in precision Π (χ is one of the tracked variables in explicit-value abstraction or χ is a tracked predicate in predicate abstraction). Let $orig(\chi)$ denote the set of concrete (original) variables that appear in χ (in explicit-value abstraction, $orig(x) = \{x\}$ if x is a tracked variable; for a predicate $y > z$, $orig(y > z) = \{y, z\}$).

This information, the precision can be used to boost partial order reduction. If a variable x is not present in the current precision, it is unnecessary to consider two actions dependent just because they both use x (if there is no other global variable that they both access) since the value of x is ignored in the current abstraction. With explicit-value abstraction, it is enough to take the tracked variables into consideration when calculating dependency between actions. Similarly, when using predicate abstraction, two actions are only dependent if there is a predicate that has variables from both actions.

Example 5. *Let us have two processes. Let the model checking reach a state s where the only enabled actions are* $\alpha\{x = 2 \cdot z\}$ *and* $\beta\{y = x - 1\}$ *from different processes.*

a. *If we calculate a source set in this state in the traditional way, we need to include both* α *and* β *in our set because they both use the object* x*, so they are dependent regardless of the applied abstraction.*

b. *Let us assume, that we use explicit-value abstraction and the set of tracked variables is currently* $\Pi = \{y, z\}$*. Since* $\nexists\ \chi \in \Pi$ *such that* $x \in orig(\chi)$ *(*x *is not in the precision) and* x *is the only object that both* α *and* β *accesses, we can consider* α *and* β *independent in the current abstraction.*

c. *Now, let us use predicate abstraction and let the set of tracked predicates be* $\Pi = \{y > 2, x + z = 0\}$*. As* $x \in orig(x + z = 0)$ *(there is a predicate about* x *in* Π*),* α *and* β *is considered dependent in this abstraction even with the proposed method. They are also dependent if* $\Pi = \{y > z\}$ *since* $y, z \in orig(y > z)$*, that is the predicate* $y > z$ *uses variables from both* α *and* β*. However, the two actions are independent with precision* $\Pi = \{y > 0, z = 2\}$*.*

The motivation for developing this abstraction-aware POR algorithm is to make fewer actions dependent. By decreasing the dependency in the model, the reducing effect of partial order reduction hopefully increases resulting in better performance.

The introduced concepts are illustrated in a small case study on a complete multi-threaded program with figures about the abstract state spaces in Section 6.1.

4.2.2 Description of the Algorithm

First, a simple version of the algorithm is described, then an extension is explained that makes the proposed algorithm compatible with lazy state space computation [28]. Finally, the correctness of the presented methods is proven in this section.

4.2.2.1 Simple Version

When using a basic version of CEGAR, the abstraction-aware POR algorithm is quite simple. However, some lazy computation can improve CEGAR, which requires further steps to preserve the correctness of AAPOR. This will be explained in detail later in this section. Let us start with the simple version.

The criterion for applying the simple version of AAPOR is to start building the abstract state space from scratch in every iteration of CEGAR. In this case, only the calculation of dependency is different compared to Algorithm 2 presented in Section 4.1. Algorithm 3 is the modified algorithm with differences highlighted.

The algorithm receives the precision Π of the current abstraction as an input. Dependency is calculated in Line 5 with respect to precision Π, which can be formalized as a modification of the sufficient conditions for the independence of actions introduced in Section 2.3.1:

Lemma 1. It is sufficient to determine whether two actions α and β are independent in a state s with precision Π by the following conditions. Actions $\alpha, \beta \in \mathit{enabled}(s)$ are independent in s with respect to Π if:

- α and β are not the actions of the same process, *and*
- there is no abstract variable in the precision in which an object accessed by α and an object accessed by β both appear. ∎

To clarify the grammatically complex wording of the lemma, the following mathematical notation and formula can be used to describe the second condition. Let X_α and X_β be the set of objects accessed by actions α and β. The condition says that $\nexists\ \chi \in \Pi$ such that $\mathit{orig}(\chi) \cap X_\alpha \neq \emptyset \wedge \mathit{orig}(\chi) \cap X_\beta \neq \emptyset$. Lemma 1 will be proven later in Section 4.2.3.1.

Note that the output of Algorithm 3 is only guaranteed to be a valid source set in the current abstraction. In the basic version of POR where the abstraction is not taken into consideration, if the same actions are enabled in a state, any calculated source set is also a source set in any other abstraction. Now, this is not the case. For this, consider the following example.

Algorithm 3: Calculating a Source Set in an Abstraction

```
   Input: $s$, $EA$, $IA \subseteq EA$, $\Pi$          /* $\Pi$:  precision of the abstraction */
   Output: $P$                                /* Source set in this abstraction */
1  $P \leftarrow IA$
2  $newAdded \leftarrow True$
3  while $newAdded$ do
4      $newAdded \leftarrow False$
5      $toAdd \leftarrow \{\alpha \ : \ \alpha \in EA \setminus P,\ \exists \beta \in future\_actions(s, \alpha), \exists \gamma \in P$
                   such that $\beta$ and $\gamma$ are dependent with respect to $\Pi\}$
6      if $toAdd \neq \emptyset$ then
7          $P \leftarrow P \cup toAdd$
8          $newAdded \leftarrow True$
9      end
10 end
```

Example 6. *Let us take case b of Example 5. $\{\alpha\}$ is a source set in s in the abstraction with precision $\{y, z\}$ since α and β are independent in this abstraction.*

Let us assume that in the next refinement step, x is added to the set of tracked variables: the precision becomes $\{x, y, z\}$. $\{\alpha\}$ is not a source set anymore in s with this new precision since α and β are dependent now.

In the simple version of AAPOR where the abstract state space is reexplored from the initial state, this limitation of the validity of a source set does not matter. However, the consequences must be handled in the setting of the next section.

4.2.2.2 Compatibility with Lazy Pruning

The simple version of AAPOR required to build the abstract state space from the ground up. However, CEGAR can be optimized to use parts of the abstract state space built in the previous iteration. The refiner prunes the abstract state space so that the spurious counterexample found in the previous iteration can never be found again. At the same time, it keeps the other part of the abstract state space that *cannot be blamed* for finding the spurious counterexample. This is called lazy pruning [28].

If AAPOR is naively used together with lazy pruning, the result may be incorrect. Consider a situation where the abstract state space is reduced with AAPOR with a precision Π. Let s be a state with a calculated source set P which is a valid source set with Π. The abstractor finds a counterexample which turns out to be spurious. The refiner lazily prunes the abstract state space. Let s not be in the pruned part: it is kept in the abstract state space for the next iteration. Let the precision change so that P is not a valid source set anymore. When the abstractor expands the state space in the next iteration, it does not deal with the preserved part of the state space and explores only from the state(s) where the state space has been pruned. Unfortunately, this is not a source set selective search now, since the explored actions from s, P is not a source set in the new abstraction. That means, the correctness of the algorithm is no longer guaranteed.

In order to preserve that the exploration of the state space is a source set selective search, exploration has to start again from states where the previously calculated source set is not a source set anymore. For this, the source set calculation must be extended: a set of variables is returned along with a source set with the semantics that the returned set of actions is only a source set until none of the returned variables appears in the precision. This way, when a new variable is entered into the precision, the abstractor will know which states to recompute. Fortunately, previous exploration from such a state s can be preserved, only some new set of actions must be explored in addition which complete the no-more-source set to a source set (technically this means that the set of actions already explored from s are given to Algorithm 4 as the *initial actions* input parameter).

More formally, Algorithm 4 (the modified version of Algorithm 3) is used to calculate source sets. It returns the tuple $\{P, X\}$ for a state s and a precision Π, where P is a source set in s until $\forall\, \chi \in \Pi', X \cap orig(\chi) = \emptyset$ for the precision Π' of any later abstraction. X is called the validity set of P. In lines 7-9, for each action α that is not yet in P (and will not be added in that iteration), variables are collected whose presence in the precision would mean that α has to be added to P. At the end of the procedure, X is calculated as the union of variables associated with actions not in the source set. Let $P_{s,i}$ and $X_{s,i}$ denote the sets P and X calculated for state s in iteration i of CEGAR.

Algorithm 4: Calculating a Source Set in an Abstraction with Lazy Pruning

```
Input: $s$, $EA$, $IA \subseteq EA$, $\Pi$
Output: $P$, $X$                    /* $P$ is a source set until $X \cap \Pi' = \emptyset$ */
$P \leftarrow IA$
$newAdded \leftarrow True$
$ignored \leftarrow \{\}$                                      /* Empty hash map */
while $newAdded$ do
    $newAdded \leftarrow False$
    $toAdd \leftarrow \{\alpha \ : \ \alpha \in EA \setminus P, \ \exists \beta \in future\_actions(s, \alpha), \exists \gamma \in P$
                    such that $\beta$ and $\gamma$ are dependent with respect to $\Pi\}$
    foreach $\alpha$ in $EA \setminus (P \cup toAdd)$ do
        $ignored[\alpha] \leftarrow ignored[\alpha] \cup \{x \ : \ \exists \beta \in future\_actions(s, \alpha), \exists \gamma \in P$
                    such that $\beta$ and $\gamma$ are dependent with respect to $\Pi \cup \{x\}\}$
    end
    if $toAdd \neq \emptyset$ then
        $P \leftarrow P \cup toAdd$
        $newAdded \leftarrow True$
    end
end
$X \leftarrow \bigcup_{\alpha \in ignored, \ \alpha \notin P} ignored[\alpha]$
```

Algorithm 5: Process Preserved States after Lazy Pruning

```
Input: $S_{preserved}$, $\Pi$
foreach $s$ in $S_{preserved}$ do
    if $\exists \ \chi \in \Pi_i$ such that $X_{s,i-1} \cap orig(\chi) \neq \emptyset$ then
        $(P_{s,i}, \ X_{s,i}) \leftarrow$ call Algorithm 4($s$, $enabled(s)$, $P_{s,i-1}$, $\Pi$)
        explore each $\alpha \in P_{s,i} \setminus P_{s,i-1}$ from $s$
    else
        $(P_{s,i}, \ X_{s,i}) \leftarrow (P_{s,i-1}, \ X_{s,i-1})$
    end
end
```

In the refinement step of CEGAR, part of the abstract state space from iteration $i - 1$ is preserved according to a lazy pruning technique. Let the preserved states be S' and the refined precision of iteration i be Π_i. The handling of preserved states is shown in Algorithm 5 with the following explanation: from all state $s \in S'$ where $\exists \ \chi \in \Pi_i$ such that $X_{s,i-1} \cap orig(\chi) \neq \emptyset$, exploration restarts. Algorithm 4 gets $P_{s,i-1}$ as the "initial actions" input for such a state. For a state $s' \in S'$ where $\forall \ \chi \in \Pi_i, X_{s,i-1} \cap orig(\chi) = \emptyset$, exploration does not restart from s'.

4.2.3 Correctness of the Presented Methods

In this section, it is proven that the presented algorithms preserve the correctness of model checking, that is the new algorithms yield an equivalent result with the original problem. First, the soundness of the simple version, and then the correctness of the lazy pruning compatible version is proven.

4.2.3.1 Correctness of the Simple Version

The correctness of the simple AAPOR algorithm can be formalized with the following theorem:

Theorem 4. Let the exploration of the abstract state space start from the abstract initial state in every iteration of CEGAR. Let s be a state in the abstract state space reached during the exploration and let the set of explored actions from s be calculated with Algorithm 3. The state space exploration is a source set selective search in every iteration of CEGAR. ▪

The correctness of a source set selective search is already proven as explained in Section 4.1. So if Theorem 4 holds, the simple version of AAPOR is sound. To prove Theorem 4, we prove Lemma 1, then we conclude the theorem with a few more steps. First, let us recall Lemma 1:

Lemma 1. It is sufficient to determine whether two actions α and β are independent in a state s with precision Π by the following conditions. Actions $\alpha, \beta \in \mathit{enabled}(s)$ are independent in s with respect to Π if:

- α and β are not the actions of the same process, *and*
- there is no abstract variable in the precision in which an object accessed by α and an object accessed by β both appear. ▪

In the proof of the lemma, it is shown that the new conditions given in this lemma for the independence of two actions are sufficient indeed in the abstract state space built with the same precision. Since we are dealing with an over-approximation of the dependency relation, where we can safely say that two actions are dependent (even if they are independent in reality), we only have to check the cases where two actions are independent based on the new conditions.

Proof (Proof of Lemma 1). Let $\alpha, \beta \in \mathit{enabled}(s)$, p_α and p_β be the process of α and β, and Π be the current precision. Let X_α and X_β be the set of objects accessed by α and β respectively.

If $p_\alpha = p_\beta$, they are dependent owing to the first condition. Again, if $\exists\ \chi \in \Pi$, such that $X_\alpha \cap \mathit{orig}(\chi) \neq \emptyset$ and $X_\beta \cap \mathit{orig}(\chi) \neq \emptyset$, α and β are dependent based on the second condition of the lemma.

So let us check the definition of independent actions (Definition 4) in the remaining case where $p_\alpha \neq p_\beta$ and $\nexists\ \chi \in \Pi$, such that $X_\alpha \cap \mathit{orig}(\chi) \neq \emptyset$ and $X_\beta \cap \mathit{orig}(\chi) \neq \emptyset$. Here, the conditions of the lemma tell us that α and β are independent. The two criteria in the definition of independence:

- $\beta \in \mathit{enabled}(\mathit{post}(s, \alpha))$ and $\alpha \in \mathit{enabled}(\mathit{post}(s, \beta))$

  Indirectly, let us assume that $\beta \notin \mathit{enabled}(\mathit{post}(s, \alpha))$, that is, β is disabled by α (the case is symmetric for α and β). Since process p_β is at the same location in s and $\mathit{post}(s, \alpha)$ (as $p_\alpha \neq p_\beta$), β can only be disabled if its guard condition evaluates to false in $\mathit{post}(s, \alpha)$. As the guard condition of β is true in s (because $\beta \in \mathit{enabled}(s)$), the evaluation of the guard expression of β is different in s and $\mathit{post}(s, \alpha)$. Consequently, some abstract information (an abstract variable) about variables used by β is changed by α.

The previous statement says that α changes the value of an abstract variable χ (so $X_\alpha \cap orig(\chi) \neq \emptyset$), and the guard condition of β depends on χ (so $X_\beta \cap orig(\chi) \neq \emptyset$). This contradicts our supposition that $\nexists \chi \in \Pi$ such that $X_\alpha \cap orig(\chi) \neq \emptyset$ and $X_\beta \cap orig(\chi) \neq \emptyset$.

- $post(post(s,\alpha),\beta) = post(post(s,\beta),\alpha)$

  Let $s = (l_{p_\alpha}, l_{p_\beta}, ..., d_{\alpha,1}, ..., d_{\alpha,n}, d_{\beta,1}, ..., d_{\beta,m}, ...)$ where l_{p_α} and l_{p_β} are the location of p_α and p_β in s; $d_{\alpha,i}$ and $d_{\beta,j}$ are the values of the abstract variables related to α and β respectively (these abstract variables are disjoint indeed based on our supposition)[5].

  Executing α changes l_{p_α} and may change the values $d_{\alpha,1}, ..., d_{\alpha,n}$ but leaves the locations l_{p_i} (for all other processes $p_i \neq p_\alpha$) and the values of other abstract variables (other than $d_{\alpha,j}$) as they are. The same is true for β, analogically. So:

  $post(s,\alpha) = (l'_{p_\alpha}, l_{p_\beta}, ..., d'_{\alpha,1}, ..., d'_{\alpha,n}, d_{\beta,1}, ..., d_{\beta,m}, ...)$

  $post(s,\beta) = (l_{p_\alpha}, l'_{p_\beta}, ..., d_{\alpha,1}, ..., d_{\alpha,n}, d'_{\beta,1}, ..., d'_{\beta,m}, ...)$

  Since β only uses the abstract variable values $d_{\beta,1}, ..., d_{\beta,m}$ (and obviously only the location of p_β matters for β), as far as β is concerned, s and $post(s,\alpha)$ is equivalent. So taking β from s or $post(s,\alpha)$ will lead to the same location l'_{p_β} and will set the same new values $d'_{\beta,1}, ..., d'_{\beta,m}$ for the related abstract variables. Again, the same is true for α, analogically. Thus:

  $post(post(s,\alpha),\beta) = post(post(s,\beta),\alpha) = (l'_{p_\alpha}, l'_{p_\beta}, ..., d'_{\alpha,1}, ..., d'_{\alpha,n}, d'_{\beta,1}, ..., d'_{\beta,m}, ...)$

Both criteria in the original definition of independence is met so α and β are indeed independent in the supposed case. □

With Lemma 1, the proof of Theorem 4 is immediate:

Proof (Proof of Theorem 4). The abstract state space is built all over again from the initial state in every iteration of CEGAR. Let us take one iteration. The actions to explore are calculated with Algorithm 3 from every state. The way Algorithm 3 calculates dependency between actions is a sufficient over-approximation of the dependency relation of actions based on Lemma 1. As a consequence, the correctness of Algorithm 3 is equivalent with Algorithm 2 whose correctness is explained in Section 4.1.1.2. Thus, the set of returned actions is a source set in that particular state and remain a source set throughout this iteration since the abstraction does not change during an iteration. That is, a source set selective search is performed in every iteration. □

4.2.3.2 Correctness of the Integration with Lazy Pruning

The following theorem proves the correctness of the extended AAPOR algorithm which can be used when the abstract state space is pruned lazily:

Theorem 5. Let s be a state in the abstract state space and Π be the current precision. If s has not been explored before, let the set of explored actions from s and an associated set of variables be calculated with Algorithm 4. If s has been explored previously, let s be processed again with Algorithm 5. The state space exploration performed this way is a source set selective search in every iteration of CEGAR. ▪

[5] The first "..." stands for the locations of other processes and the last "..." signifies the values of other abstract variables that are neither related to α nor β.

Proof. First, observe that Algorithm 4 and Algorithm 3 calculates source sets exactly the same way (the modifications in Algorithm 4 does not affect the returned set P). Thus, the returned set P by Algorithm 4 is a source set in the iteration when it has been calculated as we have seen it for Algorithm 3 in the proof of Theorem 4.

Now, let us take an iteration i of CEGAR with a set of preserved states $S_{preserved}$ from the previous iteration $i-1$ (for $i=0$, $S_{preserved}=\emptyset$) and let Π be the precision of the applied abstraction in iteration i. We show that the set of explored actions is a source set in every state at the end of iteration i. We have the following two cases for a state s:

1. $s \in S_{preserved}$: s is reprocessed with Algorithm 5.

   If $\exists \chi \in \Pi$ such that $X_{s,i-1} \cap orig(\chi) \neq \emptyset$, the set of actions to explore $P_{s,i}$ is recalculated with Algorithm 4 and the actions in $P_{s,i}$ that has not been explored previously are explored from s. So the set of explored actions from s at the end of this iteration is $P_{s,i}$ which is a source set in this abstraction based on our observation at the beginning of this proof.

   If $\nexists \chi \in \Pi$ such that $X_{s,i-1} \cap orig(\chi) \neq \emptyset$, $P_{s,i-1}$ is still a source set in s in this abstraction based on Lemma 1.

2. $s \notin S_{preserved}$: the set of actions to explore $P_{s,i}$ is calculated with Algorithm 4 and all actions in $P_{s,i}$ are explored from s. So the set of explored actions from s at the end of the iteration is $P_{s,i}$ which is a source set in this abstraction based on our observation at the beginning of this proof.

We have seen that the set of explored actions is a source set for each state in the abstract state space at the end of any iteration. So a source set selective search is performed in every iteration of CEGAR. □

4.3 Implementation

I implemented the presented abstraction-aware partial order reduction algorithm into the open-source CEGAR-centric model checking framework Theta [38, 28, 3]. The verification tool is developed by the Fault Tolerant Systems Research Group (FTSRG) at the Department of Measurement and Information Systems (MIT) of our university.[6]

The practical output of my work is a contribution to this open-source project in the form of a GitHub pull request[7] and parts of the code on a development branch[8]. With my contribution, Theta is capable of verifying considerably more concurrent programs (see details in the evaluation chapter, in Section 6.2). Just as in previous years, Theta will participate in SV-COMP [10] later this year. SV-COMP is a software verification competition where verification tools have to verify programs as fast as they can. Hopefully, with the contributed algorithms, Theta will be able to solve much more tasks in the concurrency safety category of the competition, thus ranking much better than in previous years.

[6] https://ftsrg.mit.bme.hu

[7] The pull request is available at: https://github.com/ftsrg/theta/pull/177

[8] The branch is available at: https://github.com/ftsrg/theta/tree/xcfa-refactor

4.3.1 Theta

Theta is a configurable model checking framework which supports several formalisms including C programs as an input model [38, 28, 9]. The core of its model checking algorithm is CEGAR. Theta is designed to perform reachability analysis in a state space (for possibly different interpretations of error states).

The framework has been created with configurability in mind: different abstraction domains and refinement strategies are implemented to compare their performance. Theta can be easily extended to support the verification of new formalisms by implementing a new front-end that can interpret the desired models.

4.3.2 Implementation of Abstraction-Aware POR

Partial order reduction has a role when building the abstract state space, so POR has been implemented in the abstractor component of Theta. More specifically, when the abstract state space is expanded, the enabled actions are collected. There is an interface LTS (standing for labelled transition system) which has a method that can return the enabled actions for a state and a precision. The original implementation of this interface simply returns all enabled actions.

I added two new implementations: one that works according to the traditional POR algorithm (this version does not use the precision for calculating dependency) and another that realizes abstraction-aware POR.

The compatibility with lazy pruning mainly required additions in the refiner component. At the end of the refinement step, each state is marked whose source set is not a source set in the new abstraction. The abstractor is extended so that already explored actions are not processed again in such states.

The implementation of the new algorithms preserve the configurability of Theta: I used dependency injection to add the new algorithms in a manner that can be easily configured, changed or extended.

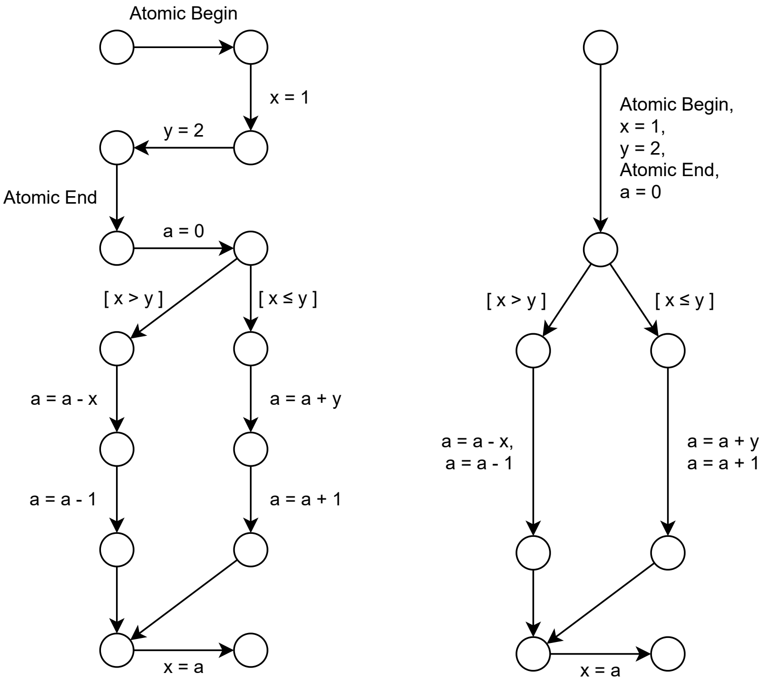
4.3.3 An Optimization - Large-Block Encoding

Exploring a transition in the state space means that an SMT problem has to be solved [28]. It is a costly operation, so we try to minimize necessary exploration during the verification. That is what partial order reduction does. On the other hand, other methods can also reduce the number of transitions in the state space.

Large-block encoding (LBE) achieves this by grouping several actions on the same transition [16]. This way, more complex but less SMT problems have to be solved. Benchmark results in [16] and in Section 6.2 show that this trade-off is well worth it. Actions can be grouped on the same transition based on various methods.

I apply a simple version of LBE, where action groups are formed in a way that any consecutive thread-local operations and operations of atomic blocks are appended after a global operation. (By global operations, I mean operations that use global variables; the rest are thread-local operations.) The semantics of these action groups (lists) are that the actions in it are performed sequentially. Let us illustrate how the implemented version of LBE works on a small example.

```
void func() {
  ATOMIC_BEGIN();
  x = 1;
  y = 2;
  ATOMIC_END();
  int a = 0;
  if(x > y) {
    a -= x;
    a--;
  } else {
    a += y;
    a++;
  }
  x = a;
}
```



(a) Source code **(b)** Single-block encoding **(c)** Large-block encoding

Figure 4.2: Small example to illustrate the presented algorithms

Example 7. *Let us have the C function from Figure 4.2a (the function does not perform any meaningful task).* `ATOMIC_BEGIN()` *and* `ATOMIC_END()` *mark the beginning and the end of an atomic block, respectively.* `x` *and* `y` *are global variables, while* `a` *is local.*

Figure 4.2b is the CFA of the program without LBE (that is, it uses SBE, single-block encoding). Figure 4.2c shows the CFA where LBE is applied. The operations on the first edge were grouped together because the first four operations form an atomic block and `a = 0` *is a thread-local operation. Then, in the two branches, the first operation is global since they use* `x` *or* `y`*, and the second is thread-local, so they can be represented by a single edge. The condition check and the last operation are global operations without any thread-local operations after them, thus, they remain alone.*

This version of LBE works well with partial order reduction. By having at most one global operation per edge, new dependency does not arise. If we group an operation using x and an operation using y on a single transition, it would be dependent with any action that uses any of x and y. So the applied LBE does not counteract partial order reduction.

Chapter 5

Data Race Detection

Data races occur in a concurrent program, when multiple processes or threads access the same shared memory location simultaneously [34]. Since data races lead to undefined behavior, we would like to avoid them most of the time. It can lead to situations where the memory contains a value that was not intended to be written by any operations (see Example 8). In safety-critical systems, such errors are naturally intolerable. As a consequence, the formal verification of the concurrent software of such a system extends to data race detection.

Partial order reduction can be used similarly for data race detection as for error location reachability analysis. However, the formal requirement is different from what we have seen so far in this work, therefore some modifications are inevitable. This chapter introduces data races and the way partial order reduction can be used to improve the performance of data race detection. The soundness of detecting data races this way is proven as well. Evaluation of partial order reduction for data race detection can be found in the next chapter of evaluation.

5.1 Data Races

The definition of data races are taken from the C standard (ISO/IEC 9899:201x §5.1.2.4 paragraph 25) [29]:

Definition 10 (Data Race). The execution of a program contains a *data race* if it contains two conflicting actions in different threads, at least one of which is not atomic, and neither happens before the other. Any such data race results in undefined behavior. ▪

Based on the standard, two actions conflict if one of them modifies a memory location and the other one reads or modifies the same memory location. This is one of the sufficient conditions of dependency introduced in Section 2.3.1.[1] Two actions that cause a data race are referred to as *racing actions*.

From the perspective of model checking, "neither happens before the other" means that there is a state in the state space where the two racing actions are both enabled.

Consider the following example for data races.

[1] The sufficient conditions for dependency introduced in Section 2.3.1 does not distinguish read and write operations, but it could be extended with this condition without any consequences. This condition is only ignored there to follow the implementation.

Initially: `int x = 0;`

Thread t_1	Thread t_2	Thread t_3
(α) `x = 1;`	`ATOMIC_BEGIN;`	`ATOMIC_BEGIN;`
	(β) `x = 512;`	(γ) `x = 2;`
	`ATOMIC_END;`	`ATOMIC_END;`

Figure 5.1: Small example to illustrate data races

Example 8. *Let us have the threads from Figure 5.1.*

The execution $\alpha.\beta$ contains a data race because α and β are racing actions: α and β write the same variable `x`*, one of them is not an atomic operation (note that the assignment of an* int *is not atomic), and neither happens before the other (α and β are both enabled in the initial state: $\beta.\alpha$ is also a possible execution).*

Let `x` *be a 4-byte int. Imagine the situation where the first three bytes are written to* `x` *by α, then β writes all bytes of its value into* `x`*, finally α finishes with its last byte. The value of* `x` *will be 513 which was not intended by either process.*

On the other hand, the execution $\beta.\gamma$ does not contain a data race since both β and γ are atomic operations (even though they access the same variable).

5.2 Partial Order Reduction for Data Race Detection

For verifying that a program contains no data race, the formal requirement is the following: there is no state in the state space where two conflicting (dependent) actions from different threads are enabled and one of them is not atomic. In this formulation, it is assumed without the loss of generality that operations of an atomic block appear as a single action. On the one hand, this can be achieved by using a version of large-block encoding that groups operations of an atomic block on a single edge. On the other hand, when calculating dependency, all variable accesses can be collected in the atomic block (with a simple graph search until the end of the atomic block is reached).

So now, an error state is a state where two racing actions are enabled. We can use a source set selective search to check the existence of such error states. Recall Theorem 2: it states that if we build a reduced state space S_R with a source set selective search of the original state space S, then for each state s in S_R and execution w from s in S, there is an execution w' in S_R such that $w.v \simeq_s w'$ for some transition sequence v. Note that in error location reachability analysis, only error states are reachable from an error state due to the fact that error locations are deadlocks (locations without outgoing edges) in the CFA: if w ends in an error state, then $w.v$ as well. However, non-error states may be reachable from an error state, now. This difference makes the proof of the correctness of using POR for data race detection slightly more complex.

Note: we can assume that racing actions do not have guard conditions. If we have an action with a guard condition, we can first store the value of the condition in a local variable and use the local variable in the guard. So we can replace an action $[cond]$ with two actions $localVar = cond$ and $[localVar]$. This way, data race can occur between actions without a guard condition.

As a consequence of this assumption, if we have an action α that is in a race with another action, α is the single action starting from the source location of α in the CFA of p_α (since branching operations always have guard conditions in a program).

Theorem 6. Let S be the original state space, and S_R be the reduced state space obtained from S by a source set selective search (satisfying the conditions of Theorem 2). Let w be an execution in S from state s in S_R that contains a data race.

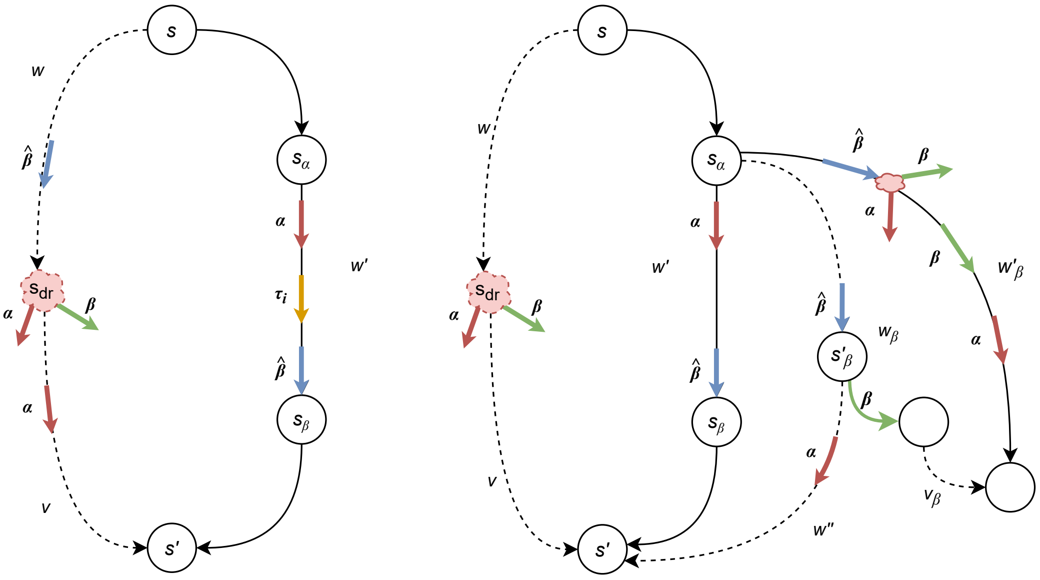
There is also an execution w' in S_R from s that contains a data race. ▪

Proof. Since we use a source set selective search, based on Theorem 2, there is an execution w' in S_R such that $w.v \simeq_s w'$ for some transition sequence v. That is $w.v$ and w' are in the same Mazurkiewicz trace. Let s_{dr} be the state with $s \xrightarrow{w} s_{dr}$ (where the data race occurs), and s' be the state with $s \xrightarrow{w.v} s'$ and $s \xrightarrow{w'} s'$.

Let the processes of α and β be p_α and p_β, and the source location of α and β in the CFA of their processes be l_α and l_β respectively. In s_{dr}, p_α is in l_α, and p_β is in l_β.

Since $w.v$ and w' are in the same Mazurkiewicz trace, the same actions are used (their order may be different). As a consequence, the same CFA locations are reached in w' as in $w.v$. Since l_α is reached in w (e.g., in s_{dr}), p_α is in l_α in some states of w' (same for β). Let l_α be reached first in w' (the case is symmetric, we could choose l_β): let s_α be the state where l_α appears first in w' (that is where α is first enabled in w'), and s_β be the state where l_β appears first. From our assumption, s_α is reached earlier in w' than s_β. It is not assured though, that α is still enabled in s_β (p_α may have a step between s_α and s_β).

If α is still enabled in s_β, we are ready because we have found the data race in S_R. For the rest of the proof, assume that α is not enabled in s_β. This can only happen if α is one of the actions of w between s_α and s_β since α does not have a guard condition (α cannot simply "get disabled"), and it is the only action from l_α (α cannot be bypassed) based on the notes made before the theorem. As $w.v$ and w' in the same Mazurkiewicz trace, α appears in v.



(a) Case I. **(b)** Case II.

Figure 5.2: State space in the proof

Let $\hat{\beta}$ be the action of p_β preceding β in w' (i.e., the target location of $\hat{\beta}$ is l_β, and $\hat{\beta}$ is the action in w' reaching s_β). We have two cases. Figure 5.2 hopefully helps in understanding what is described in the below cases. Transition sequences with a continuous line are in S_R, while dashed lines indicate sequences in S only.

I. α is dependent with $\hat{\beta}$ transitively, that is, there are actions $\tau_1 = \alpha, \tau_2, ..., \tau_n = \hat{\beta}$ in w' in this order such that τ_i is dependent with τ_{i+1} for $1 \leq i < n$.

α starts from l_α, and in $w.v$, α comes after s_{dr}. The target location of $\hat{\beta}$ is l_β, so $\hat{\beta}$ comes before s_{dr} in $w.v$, and by its definition, $\hat{\beta}$ comes before s_β in w'.

Since $w.v$ and w' are in the same Mazurkiewicz trace, the two traces can be obtained from each other by successively swapping adjacent independent actions. In $w.v$, $\hat{\beta}$ precedes α, while in w', they appear in reversed order. To get $w.v$ from w' by successively swapping adjacent actions, we have to swap α and $\hat{\beta}$. For this, we have to swap τ_i with τ_{i+1} at some point (for some $1 \leq i < n$), but they are dependent. So we cannot obtain $w.v$ and w' from each other by successively swapping adjacent independent actions which is contradiction.

II. If the condition of case I. does not hold, we can achieve a new trace w'' with several independent swapping steps (which is still in the same Mazurkiewicz trace with $w.v$ and w') where $\hat{\beta}$ comes before α. Note that we can make these swappings so that only the part of w' after s_α changes, so w'' also passes through s_α. At the target state s'_β of $\hat{\beta}$ in w'', both α and β are enabled.

Unfortunately, w'' is an execution in S, and it is not necessarily explored (not necessarily in S_R). So we have to go on with reasoning.

Consider the execution from s_α to s'_β extended with β: w_β. We are still using a source set selective search, so we can apply Theorem 2 again. Based on the theorem, an execution w'_β is explored from s_α with $w'_\beta \simeq_{s_\alpha} w_\beta.v_\beta$ for some transition sequence v_β. Since w_β contains β but does not contain α, w'_β can only contain α after β (since $w_\beta.v_\beta$ and w'_β are in the same Mazurkiewicz trace and α and β are dependent). This means that α is enabled in all states of w'_β from s_α until β is reached. That is α and β are both enabled in the state before β in w'_β.

So we always reach a state in S_R where α and β are both enabled. □

We have proven that if we use partial order reduction for data race detection, we will find all data races in the reduced state space that are present in the original state space.

5.3 Implementation Notes

On the implementation front, data race detection required a new interpretation of error states. A new predicate has been implemented for deciding whether a state is an error state. This predicate checks whether any two enabled actions in the state are racing actions.

Otherwise, the partial order reduction algorithm (not its abstraction-aware version, only the traditional) implemented in a previous phase of this work for the state space exploration can be used without modifications.

Chapter 6

Evaluation

The implemented algorithms have been evaluated on C programs since THETA supports the parsing of C programs [9] and a large set of benchmark data is available in the form of C programs[1] [10].

6.1 Case Study

In this section the concepts and algorithms presented in the previous chapter are illustrated on a small multi-threaded program.

Let our example be the C program from Figure 6.1a (some operations are labeled for later reference). There are two threads: the main thread m and thread t is created by m. We would like to verify this program: our formal requirement is that no error location is reachable in any execution of our program ($reach\_error()$ indicates the error location). We can quickly tell that the program is unsafe: if $y = 1$ on m is executed between $y = 2$ and the condition check on t, the program reaches the error location. So we anticipate that the result of the model checking will be unsafe (with a counterexample telling us how the error location can be reached). The abstraction-aware version of POR will be used with lazy pruning.

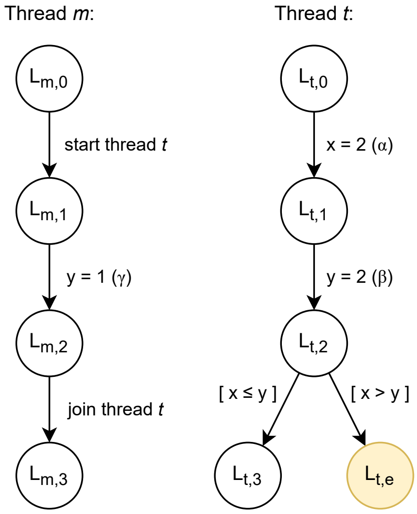
For the model checking, the program is converted to an XCFA (see Figure 6.1b), that is, both threads have its own CFA. (For the sake of simplicity, variable initialization at the beginning and return operations have been removed from the figure.) The error location is highlighted in the CFA of thread t.

Let us use explicit-value abstraction in CEGAR and let the precision be $\{x\}$ in the first iteration, so only the value of variable x is tracked. Figure 6.2a depicts the abstract state space of the first iteration. Rectangles are states and arrows are transitions. The locations of the active processes are shown in a state along with the value of the tracked variables (which is only x in this iteration). The labels of transitions indicate the performed action with the thread of that action. To be concise, $s_{i,j}$ refers to the state where the locations of the threads are $L_{m,i}, L_{t,j}$ ($s_{1,0}$ refers to the state with $L_{m,1}, L_{t,0}$ and s_3 refers to the state with $L_{m,3}$).

[1] gitlab.com/sosy-lab/benchmarking/sv-benchmarks/-/tree/main/c

```
     int x = 0, y = 0;
     void *f(void *arg) {
α:     x = 2;
β:     y = 2;
       if(x > y) reach_error();
       return 0;
     }
     int main() {
       pthread_t t;
       pthread_create(&t, 0, f, 0);
γ:     y = 1;
       pthread_join(t, 0);
       return 0;
     }
```

(a) Source code



(b) XCFA of the program

Figure 6.1: Small example to illustrate the presented algorithms

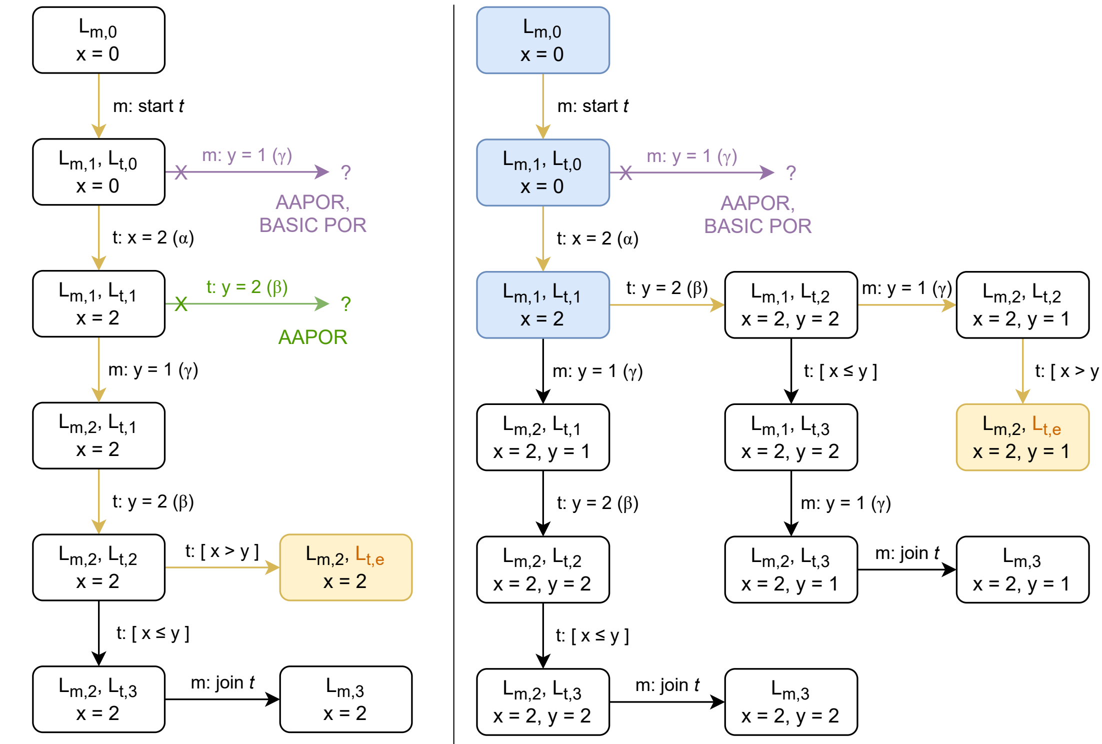
From s_0, the only enabled action is the one that starts thread t. This action forms a (trivial) source set in itself; it is explored from s_0.

In $s_{1,0}$, the enabled actions are $\{\alpha, \gamma\}$. With precision $\{x\}$, valid source sets are $\{\alpha\}, \{\gamma\}$, and $\{\alpha, \gamma\}$. Note that $\{\gamma\}$ is not a source set if we do not consider the abstraction because $\beta \in \mathit{future\_actions}(s_{1,0}, \alpha)$, and β and γ both uses the variable y, so α would have to be added to the source set. On the other hand, when we calculate dependency with the consideration of the precision, β and γ are independent since they do not commonly use any tracked variable. Algorithm 2 returns in $s_{1,0}$ the source sets $\{\alpha\}$ and $\{\alpha, \gamma\}$, while Algorithm 4 returns the source sets $\{\alpha\}$ and $\{\gamma\}$ for the initial actions $\{\alpha\}$ and $\{\gamma\}$ respectively. Then, one of the smallest source sets is chosen: $\{\alpha\}$ can be the choice both with the traditional POR and the abstraction-aware POR algorithm. The validity set returned by Algorithm 4 for this source set is $X = \emptyset$, which means this source set is valid in any abstraction. As the source set $\{\alpha\}$ has been chosen, only α is explored from $s_{1,0}$.

Unexplored transitions are marked with a cut symbol and lead to a question mark to denote that those parts of the abstract state space have not been explored. These transitions are colored with green if only abstraction-aware POR ignores it and with purple if traditional POR ignores it, too. Also, a label indicates the algorithms that ignore the given transition.

In $s_{1,1}$, β and γ are enabled. Traditional POR explores both of them because they both use variable y, so they are dependent. However, abstraction-aware POR explores only one of them, let it be γ. The validity set is $\{y\}$ for this source set: if y is added to the precision, $\{\gamma\}$ will no longer be a source set in $s_{1,1}$. Now, the main thread has to wait until thread t terminates to perform the join operation. In $s_{2,2}$, the guard condition of the actions starting from $L_{t,2}$ evaluate to unknown because y is not tracked, so both of them is enabled. One branch terminates normally, but the other reaches the error location $L_{t,e}$. The transitions of the abstract counterexample leading to the error state found in iteration 1 is highlighted with yellow on Figure 6.2a. This counterexample is given to the refiner to decide whether it is spurious or feasible.

The counterexample turns out to be spurious: if we execute the operations in this order, x will not be greater than y, so the error location is not reachable in fact. The refiner refines



(a) Abstract state space in iteration 1

(b) Abstract state space in iteration 2

Figure 6.2: Abstract state spaces in the model checking

the abstraction by adding y to the set of tracked variables: the new precision is $\Pi = \{x, y\}$. In addition, the refiner prunes the abstract state space lazily. Let the preserved states be $\{s_0, s_{1,0}, s_{1,1}\}$ (highlighted with blue on Figure 6.2b).

In iteration 2, the exploration restarts from $s_{1,1}$ where the abstract state space has been pruned in the refinement step. Besides, the preserved states are processed with Algorithm 5. In s_0, there is no unexplored enabled action, so no more actions have to be explored from there. In $s_{1,0}$, the chosen source set was $\{\alpha\}$ in the last iteration with an empty validity set $X = \emptyset$. Since $X \cap \mathit{orig}(\chi) = \emptyset$ for any abstract variable $\chi \in \Pi$, $\{\alpha\}$ is still a source set in this iteration: we do not have to explore new actions from $s_{1,0}$.

The previously used source set of $s_{1,1}$ has been $\{\gamma\}$ with a validity set $X = \{y\}$. Since we use explicit-value abstraction, $\mathit{orig}(y) = \{y\}$ for the abstract (and also concrete) variable y. $X \cap \mathit{orig}(y) = \{y\}$ which is not an empty set. So $\{\gamma\}$ is not source anymore in $s_{1,1}$, a new source set has to be calculated. Anyway, this is the point where the abstract state space has been pruned in the refinement step, so the exploration would have to start again from this state. Unfortunately, β and γ are dependent in this abstraction, so the only source set in $s_{1,1}$ is $\{\beta, \gamma\}$. It is not surprising, though: all variables are tracked in this iteration, so we do not expect that actions operating on the same variable are judged independent even by AAPOR.

From here, the exploration of the state space continues according to Figure 6.2b. Again, an abstract counterexample is found. The refiner checks whether this counterexample is spurious or feasible. Now, we have a feasible counterexample: if we execute the operations based on the yellow path leading to $s_{2,e}$ in Figure 6.2b, we reach the error location, indeed.

At this point, the verification is complete: the algorithm returned an unsafe verdict and the counterexample found in the last iteration.

6.2 Evaluating on Benchmark C Programs

The implemented partial order reduction algorithms and optimizations have been evaluated on a set of 763 C benchmark programs provided by SoSy-Lab.[2] Programs from the `pthread-*` folders were used. The same benchmarks are used in SV-COMP [10].

6.2.1 Test Configurations

The benchmark tests were carried out with different configurations of Theta. The POR algorithm itself has been tested in three version: POR disabled (`NO_POR`), only traditional POR has been applied (`BASIC`) or abstraction-aware POR has been applied (`AAPOR`). The LBE optimization was either turned on (`LBE`) or turned off (`NO_LBE`). The pruning strategy has been FULL or LAZY. The abstraction domain was explicit-value abstraction (`EXPL`) or Cartesian predicate abstraction (`PRED`). Several combinations of these configurations were tested.

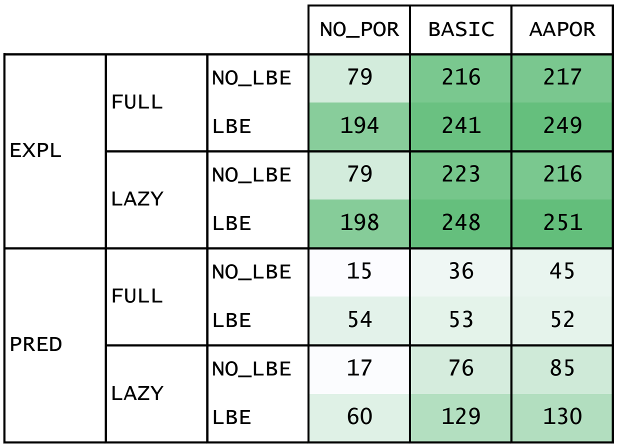
6.2.2 Results

This section shows the results of the benchmark tests. Each test had a time limit of 900 seconds: Theta had this amount of time to perform the model checking and come to a verdict.

6.2.2.1 Number of Solved Tasks

Firstly, as for correctness, the provided results (where Theta could respond before timeout) were practically all correct[3]. The great number of correct results and the absence of incorrect ones confirm that the implemented algorithms work correctly.

As for the performance, Figure 6.3 shows the number of solved tasks (out of 763) by configuration. With my contributed algorithms, Theta is capable of solving 3-8 times more problems depending on the configuration.



			NO_POR	BASIC	AAPOR
EXPL	FULL	NO_LBE	79	216	217
		LBE	194	241	249
	LAZY	NO_LBE	79	223	216
		LBE	198	248	251
PRED	FULL	NO_LBE	15	36	45
		LBE	54	53	52
	LAZY	NO_LBE	17	76	85
		LBE	60	129	130

Figure 6.3: Evolution of the algorithms

[2] gitlab.com/sosy-lab/benchmarking/sv-benchmarks/-/tree/main/c

[3] In fact, a few results are incorrect, but these tasks have been removed from the SV-COMP benchmark set since this benchmark was performed because it turned out that these tasks contain a data race which leads to undefined behavior. See details in commit `fa5078c6` of the `sv-benchmarks` repository.

An important note on the number of solved tasks is that it can largely depend on the portion of benchmark programs that can be parsed by Theta. For example, Theta currently does not support *structs* and some other C language elements. As a result, Theta can only parse 381 programs from this benchmark set.

Figure 6.4 highlights my contributions with explicit-value and predicate abstraction. The 1st (blue) columns in both groups show the results without my contributions. The results in the 2nd (orange) columns were achieved with traditional POR, the 3rd (green) with abstraction-aware POR. In the first three cases, the large-block encoding optimization was turned off; the 4th (red) columns show the results where LBE was applied.

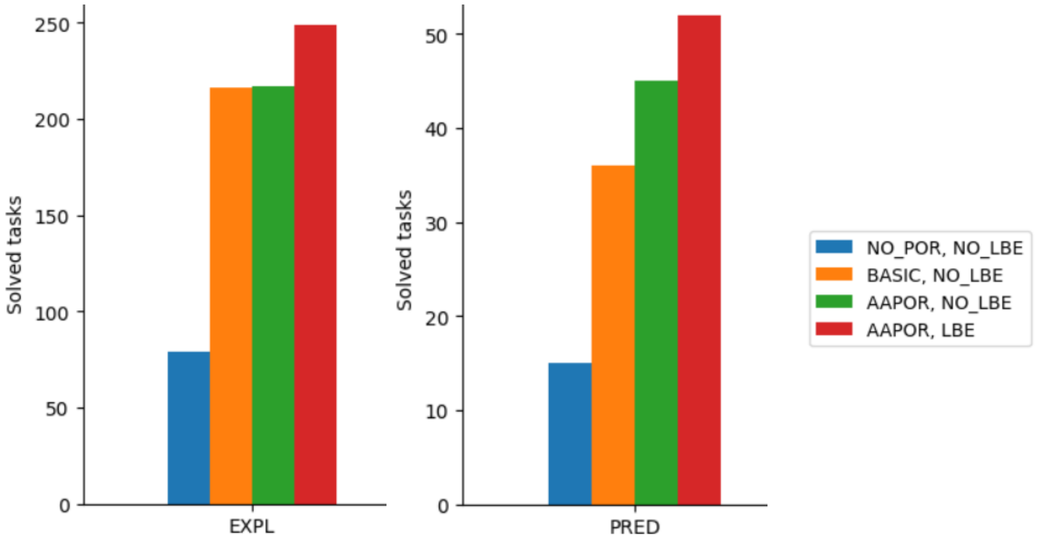


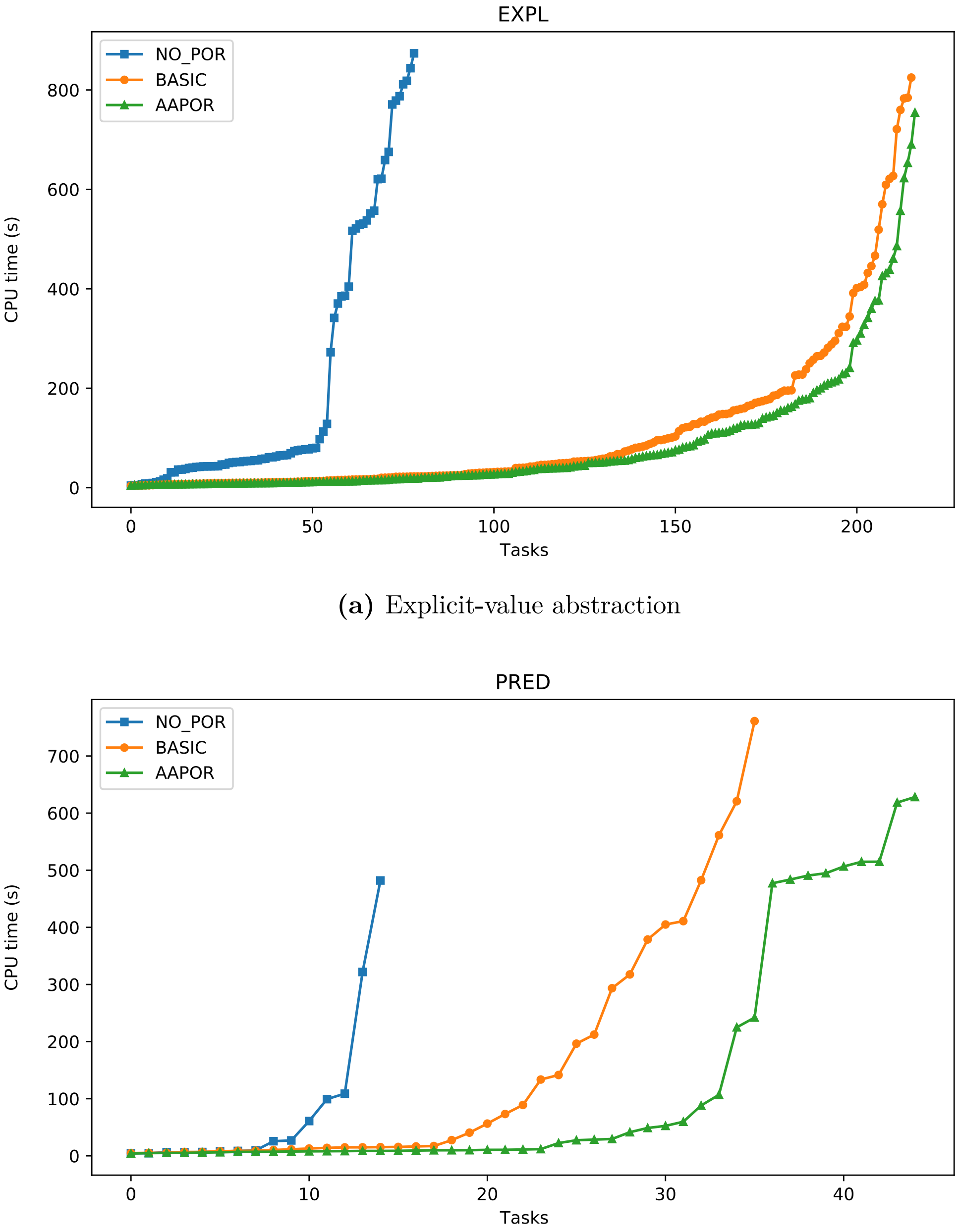
Figure 6.4: Number of solved tasks

6.2.2.2 CPU Time

Exploring the number of solved tasks does not tell us everything about performance as the benchmark programs are not homogeneous. To get a better understanding of the performance of the presented algorithms, let us have a look on the time taken to solve tasks. In this section, I concentrate on the POR algorithms and omit the LBE optimization.

The quantile plots of Figure 6.5 show the CPU time taken to solve the problems. The horizontal axis represents the tasks, while the vertical axis shows the CPU time in seconds that Theta needed to solve the corresponding problem. The problems are sorted based on time to solve: that is why the curves are monotonically increasing. Figure 6.5a and Figure 6.5b plot the CPU time measured in the `EXPL` and `PRED` domains respectively. In these plots, a flatter curve means more tasks solved in less time. These plots confirm that POR considerably improves the performance. Furthermore, though not excessively, abstraction-aware POR outperforms the traditional version of POR.

At first sight, there does not seem to be a considerable difference between traditional and abstraction-aware POR in Figure 6.5a. However, that small gap between the curve of traditional POR (marked with circles, orange) and the curve of AAPOR (marked with triangles, green) means significant difference in the average problem-solving time. In this case, traditional POR solved tasks in 110 seconds on average, while abstraction-aware POR managed it with an average of 87.3 seconds. That is, traditional POR needed 26% more time on average than AAPOR.



(a) Explicit-value abstraction

(b) Predicate abstraction

Figure 6.5: CPU time taken to solve tasks

Exploring an action in the state space is a costly operation because it requires to solve an SMT problem (with the help of the Z3 SMT solver [28]). In the benchmark tests, the number of explored actions has also been counted for each problem. Plotting these data reveals similar characteristics to the CPU time plots. The similarities are illustrated in Figure 6.6. The plots in the first row are the same as in Figure 6.5. The number of explored actions is plotted in the second row. The charts in the same column refer to the same configuration to make the similarities between the CPU time and the number of explored actions easy to see.

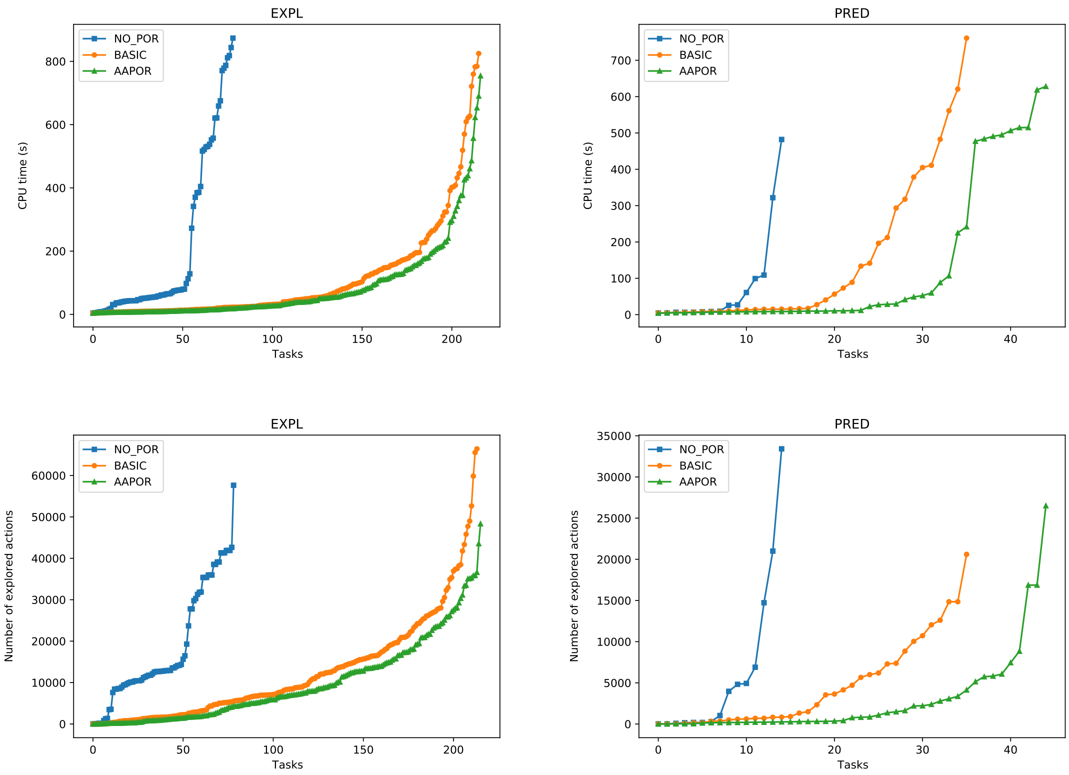


Figure 6.6: CPU time and number of explored actions

6.2.2.3 Memory Usage

The memory usage has also been measured during the verification. An important note on memory usage measurement is that THETA runs on the JVM, which means a considerable noise in the recorded data. For this reason, exact numerical results are not presented in this section. From the available 16GB of memory, less than 3GB has been used for most solved tasks.

Figure 6.7 shows relative memory usage compared to the results with the NO_POR configuration in each group. To make a fair comparison between configurations, only those tasks were considered for this chart in each configuration group which were correctly solved by each configuration in the group. The sum of used memory for these tasks were plotted relative to each other.

The chart shows that my algorithms decrease the memory usage of THETA. This meets our expectations, since as the size of the state space is reduced with POR and LBE, less memory is needed to store the explored states.

Interestingly, AAPOR needed more memory than BASIC POR in the third configuration group. In the forth column as well, AAPOR does not use significantly less memory even though the size of the explored state space is considerably smaller with AAPOR (c.f., Figure 6.6 bottom right chart). Most probably, this may be the result of the small number of tasks (15 and 17) solved by each configuration in those groups.

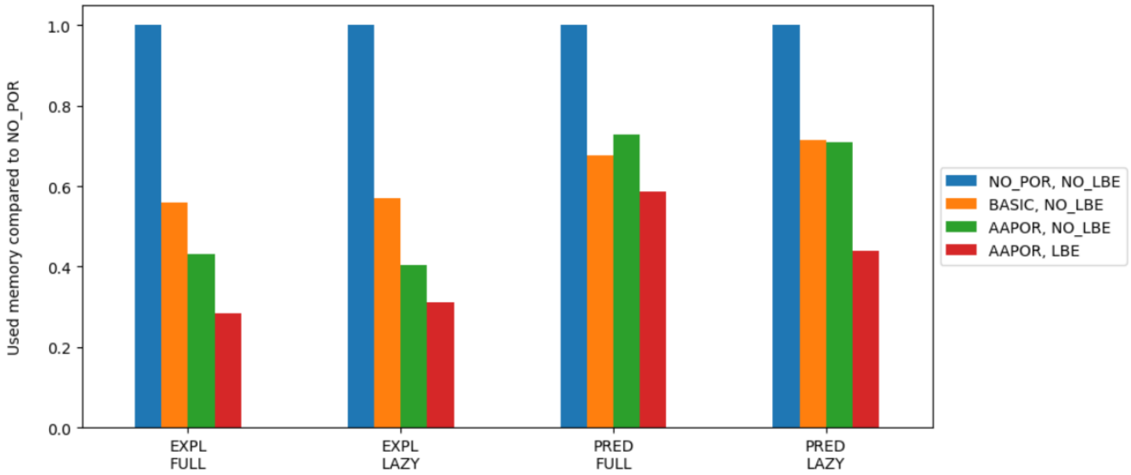


Figure 6.7: Memory usage compared to `NO_POR`

6.2.3 Comparison to Other Verification Tools

In this section, Theta is compared to other verification tools. The comparison is based on the *prerun* results of software verification competition SV-COMP 2023. These prerun tests were performed in November 2022. The real benchmark tests are carried out at the beginning of December: results have not been published at the time of writing, yet.

There are two columns for Theta: the first (with considerably less solved tasks) is the version competing in SV-COMP 2023. Unfortunately, some algorithms have not been implemented in this version concerning POR, so the full potential of my algorithms is not exploited by Theta in this year's SV-COMP. For this reason, I have added the green column for the tasks solved by Theta with my algorithms (POR and LBE).

The number of solved tasks is plotted by tool in Figure 6.8. The blue bars represent the correctly solved tasks, the red bars stand for the incorrect results, and the orange bars for the tasks that were not solved by the tool (due to timeout or a runtime error).

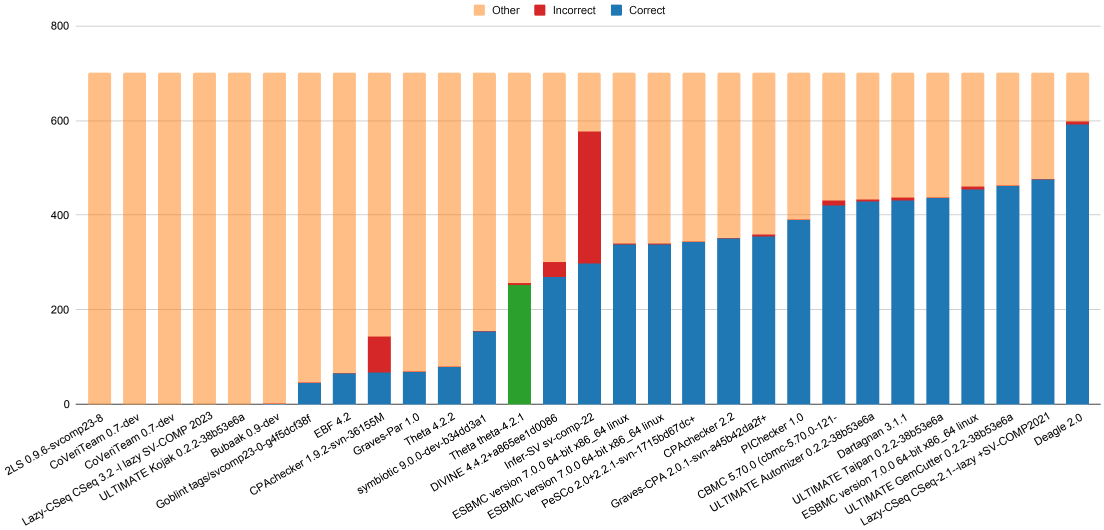


Figure 6.8: Solved tasks by tools in SV-COMP 2022

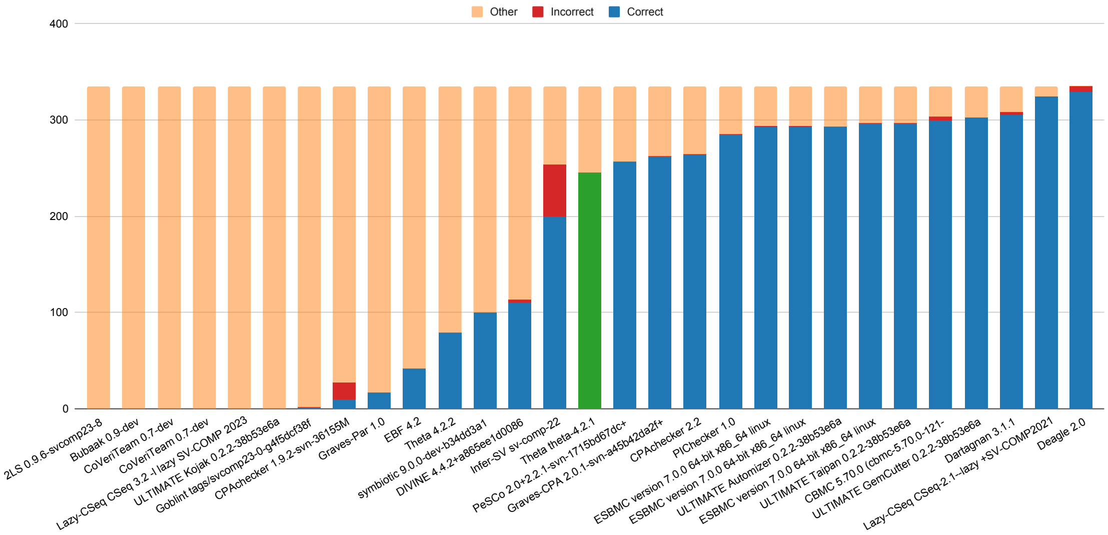


Figure 6.9: Solved tasks by tools in SV-COMP 2022 that Theta can parse

Again, keep in mind that the number of solved tasks by a tool can largely depend on the portion of benchmark programs the tool is capable of parsing. This is about half of the programs in the case of Theta. To compare the reasoning power of tools with Theta, Figure 6.9 only shows the tasks that Theta is capable of parsing.

6.3 Evaluation of Data Race Detection

This section gives a brief overview on the results of data race detection benchmark tests. From a set of 903 programs, Theta can parse 389. No incorrect results were produced. The abstraction domain, and the pruning strategy has been measured with different configurations (with the same values as in the benchmarks of the previous sections), though the results do not vary considerably based on these configurations as Figure 6.10 shows. With POR, Theta can verify about 5.3 times more tasks in this benchmark set.

		NO_POR	BASIC
EXPL	FULL	26	139
	LAZY	25	139
PRED	FULL	25	138
	LAZY	23	139

Figure 6.10: Data race detection results

Figure 6.11 plots the used CPU time by task. POR apparently improved the performance of data race detection. Though there is no difference in the number of tasks solved with explicit-value and predicate abstraction, Theta needed around 20% less time for the verification of these tasks when explicit-value abstraction was used (only counting correctly answered tasks).

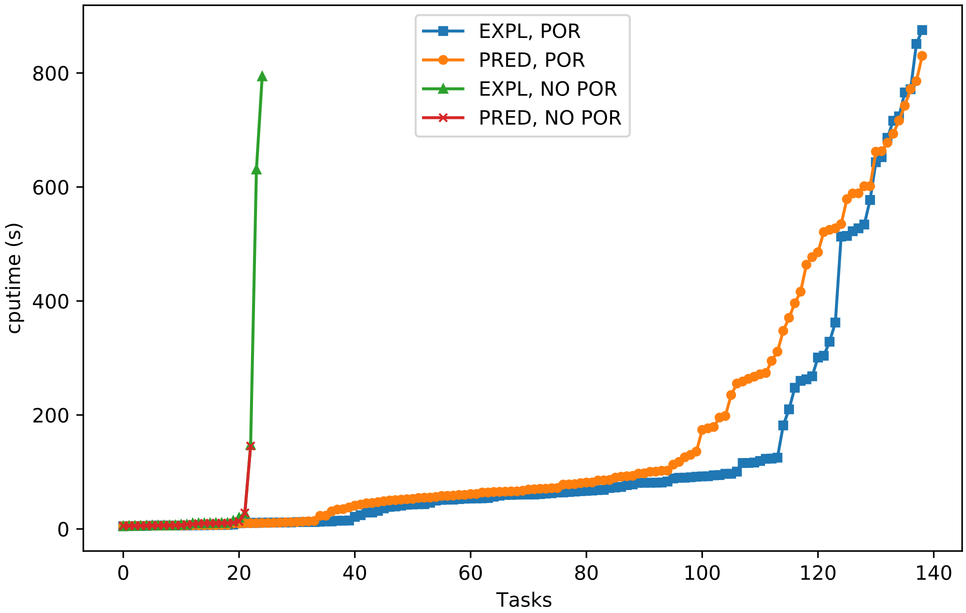


Figure 6.11: CPU time taken for data race detection tasks

6.4 Benchmark Conclusions

Benchmark results show that partial order reduction makes the abstraction-based verification of concurrent software much more efficient. The proposed abstraction-aware POR further improves the performance. Looking at both the number of solved tasks and the CPU time taken for problems to solve, we can conclude that abstraction-aware POR achieves better improvement with predicate abstraction.

Large-block encoding turned out to be a very efficient optimization technique that considerably improved the performance of the verification.

With these results on the SV-COMP benchmark data set, THETA can now be considered a competitive tool in the concurrency safety category of SV-COMP (according to results at SV-COMP 2022 [10]).

Naturally, there are some threats to the validity of the benchmarking, though hopefully, they did not change the results substantially. The tests were performed in a distributed environment of several virtual machines in the BME NIIF cloud[4]. Even though, the VMs had equal resources (16GB of RAM, 8 CPU cores, Ubuntu 20.04 LTS operating system), it cannot be assured that all tasks ran in exactly the same circumstances. Some other operating system tasks in the VMs or some fluctuations in the host environment where the VMs were run could add certain noise to the benchmark results. On the other hand, the tests were carried out with BENCHEXEC, a benchmark execution environment that fulfills the requirements for reliable benchmarking [17]. Furthermore, the tests were performed multiple times which yielded similar results. These factors strengthen the validity of the benchmarks.

[4] https://niif.cloud.bme.hu

6.5 Summary

Software verification is a difficult task where various techniques were introduced to handle data and reduce the complexity yielded by concurrent, multi-threaded software solutions. The following list summarizes my contributions in the scope of this work.

- I introduced a combined abstraction-based software verification approach that reduces the complexity of thread interactions to be explored in concurrent software.
- I proposed a novel partial order reduction algorithm working on the abstract state space representation: I devised a new definition of dependency, that exploits information encoded in the current precision of the abstraction.
- I have shown that the proposed algorithm can be used together with a lazy extension of CEGAR.
- I proved the correctness of the proposed methods.
- I implemented the presented algorithms along with a large-block encoding optimization in the THETA model checking framework.
- I have introduced along with a proof of soundness how partial order reduction can be used to improve the performance of data race detection.
- I performed benchmark tests and analyzed the results.

The proposed algorithms improve the performance of concurrent software verification. With my work, I contributed to the open-source verification tool, THETA. My contribution enables THETA to verify a wider range of concurrent programs from safety-critical systems.

6.6 Future Work

Though the presented algorithms considerably enhance the verification of concurrent programs, it is still a proof-of-concept implementation, so far. The solution could be improved in many ways. Such possibilities for future work are the following.

- A DPOR algorithm (e.g., Source DPOR or Optimal DPOR [1]) could be used as the base of POR instead of the currently applied static approach to the computation of source sets.
- The dependency relation could be further optimized by distinguishing read and write depencencies, where two read operations are independent.

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

List of Figures

Bibliography

[1] Parosh Aziz Abdulla, Stavros Aronis, Bengt Jonsson, and Konstantinos Sagonas. Optimal dynamic partial order reduction. In Suresh Jagannathan and Peter Sewell, editors, *The 41st Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '14, San Diego, CA, USA, January 20-21, 2014*, pages 373–384. ACM, 2014. DOI: `10.1145/2535838.2535845`. URL `https://doi.org/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. In Luca Aceto, Giorgio Bacci, Giovanni Bacci, Anna Ingólfsdóttir, Axel Legay, and Radu Mardare, editors, *Models, Algorithms, Logics and Tools - Essays Dedicated to Kim Guldstrand Larsen on the Occasion of His 60th Birthday*, volume 10460 of *Lecture Notes in Computer Science*, pages 516–536. Springer, 2017. DOI: `10.1007/978-3-319-63121-9\_26`. URL `https://doi.org/10.1007/978-3-319-63121-9_26`.

[3] Zsófia Ádám, Levente Bajczi, Mihály Dobos-Kovács, Ákos Hajdu, and Vince Molnár. Theta: portfolio of CEGAR-based analyses with dynamic algorithm selection (competition contribution). In Dana Fisman and Grigore Rosu, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, volume 13244 of *Lecture Notes in Computer Science*, pages 474–478. Springer International Publishing, Cham, 2022. ISBN 978-3-030-99527-0. DOI: `10.1007/978-3-030-99527-0_34`.

[4] Elvira Albert, Puri Arenas, Maria Garcia de la Banda, Miguel Gómez-Zamalloa, and Peter J. Stuckey. Context-sensitive dynamic partial order reduction. In Rupak Majumdar and Viktor Kuncak, editors, *Computer Aided Verification - 29th International Conference, CAV 2017, Heidelberg, Germany, July 24-28, 2017, Proceedings, Part I*, volume 10426 of *Lecture Notes in Computer Science*, pages 526–543. Springer, 2017. DOI: `10.1007/978-3-319-63387-9\_26`. URL `https://doi.org/10.1007/978-3-319-63387-9_26`.

[5] Stavros Aronis, Bengt Jonsson, Magnus Lång, and Konstantinos Sagonas. Optimal dynamic partial order reduction with observers. In Dirk Beyer and Marieke Huisman, editors, *Tools and Algorithms for the Construction and Analysis of Systems - 24th International Conference, TACAS 2018, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2018, Thessaloniki, Greece, April 14-20, 2018, Proceedings, Part II*, volume 10806 of *Lecture Notes in Computer Science*, pages 229–248. Springer, 2018. DOI: `10.1007/978-3-319-89963-3\_14`. URL `https://doi.org/10.1007/978-3-319-89963-3_14`.

[6] David Baelde, Stéphanie Delaune, and Lucca Hirschi. Partial order reduction for security protocols. *CoRR*, abs/1504.04768, 2015. URL `http://arxiv.org/abs/1504.04768`.

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

[8] Levente Bajczi. Application of counterexample-guided abstraction refinement on concurrent programs. Technical report, Budapesti Műszaki és Gazdaságtudományi Egyetem, Villamosmérnöki és Informatikai Kar, 2021. URL `https://tdk.bme.hu/VIK/DownloadPaper/Ellenpeldaalapu-absztrakcio-finomitas`. Scientific Students' Association Report.

[9] Levente Bajczi, Zsófia Ádám, and Vince Molnár. C for yourself: Comparison of front-end techniques for formal verification. In *2022 IEEE/ACM 10th International Conference on Formal Methods in Software Engineering*. IEEE, 2022. DOI: `10.1145/3524482.3527646`.

[10] Dirk Beyer. Progress on software verification: SV-COMP 2022. In Dana Fisman and Grigore Rosu, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, pages 375–402, Cham, 2022. Springer International Publishing. ISBN 978-3-030-99527-0.

[11] Dirk Beyer and Karlheinz Friedberger. A light-weight approach for verifying multi-threaded programs with cpachecker. In Jan Bouda, Lukás Holík, Jan Kofron, Jan Strejcek, and Adam Rambousek, editors, *Proceedings 11th Doctoral Workshop on Mathematical and Engineering Methods in Computer Science, MEMICS 2016, Telč, Czech Republic, 21st-23rd October 2016*, volume 233 of *EPTCS*, pages 61–71, 2016. DOI: `10.4204/EPTCS.233.6`. URL `https://doi.org/10.4204/EPTCS.233.6`.

[12] Dirk Beyer and M. Erkan Keremoglu. Cpachecker: A tool for configurable software verification. In Ganesh Gopalakrishnan and Shaz Qadeer, editors, *Computer Aided Verification - 23rd International Conference, CAV 2011, Snowbird, UT, USA, July 14-20, 2011. Proceedings*, volume 6806 of *Lecture Notes in Computer Science*, pages 184–190. Springer, 2011. DOI: `10.1007/978-3-642-22110-1\_16`. URL `https://doi.org/10.1007/978-3-642-22110-1_16`.

[13] Dirk Beyer and Stefan Löwe. Explicit-value analysis based on CEGAR and interpolation. *CoRR*, abs/1212.6542, 2012. URL `http://arxiv.org/abs/1212.6542`.

[14] 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`. URL `https://doi.org/10.1007/s10009-007-0044-z`.

[15] Dirk Beyer, Thomas A. Henzinger, and Grégory Théoduloz. Configurable software verification: Concretizing the convergence of model checking and program analysis. In Werner Damm and Holger Hermanns, editors, *Computer Aided Verification, 19th International Conference, CAV 2007, Berlin, Germany, July 3-7, 2007, Proceedings*, volume 4590 of *Lecture Notes in Computer Science*, pages 504–518. Springer, 2007. DOI: `10.1007/978-3-540-73368-3\_51`. URL `https://doi.org/10.1007/978-3-540-73368-3_51`.

[16] Dirk Beyer, Alessandro Cimatti, Alberto Griggio, M. Erkan Keremoglu, and Roberto Sebastiani. Software model checking via large-block encoding. In *Proceedings of 9th International Conference on Formal Methods in Computer-Aided Design, FMCAD 2009, 15-18 November 2009, Austin, Texas, USA*, pages 25–32. IEEE, 2009. DOI: `10.1109/FMCAD.2009.5351147`. URL `https://doi.org/10.1109/FMCAD.2009.5351147`.

[17] 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. URL https://doi.org/10.1007/s10009-017-0469-y.

[18] Per Bjesse. What is formal verification? *SIGDA Newsl.*, 35(24):1–es, dec 2005. ISSN 0163-5743. DOI: 10.1145/1113792.1113794. URL https://doi.org/10.1145/1113792.1113794.

[19] 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. URL https://doi.org/10.1145/876638.876643.

[20] Edmund M. Clarke, William Klieber, Milos Novácek, and Paolo Zuliani. Model checking and the state explosion problem. In Bertrand Meyer and Martin Nordio, editors, *Tools for Practical Software Verification, LASER, International Summer School 2011, Elba Island, Italy, Revised Tutorial Lectures*, volume 7682 of *Lecture Notes in Computer Science*, pages 1–30. Springer, 2011. DOI: 10.1007/978-3-642-35746-6_1. URL https://doi.org/10.1007/978-3-642-35746-6_1.

[21] Mihály Dobos-Kovács. On the verification of safety-critical embedded software systems. Master's thesis, Budapest University of Technology and Economics, Budapest, dec 2021. URL https://diplomaterv.vik.bme.hu/en/Theses/Kritikus-beagyazott-szoftverek-verifikacios.

[22] Cormac Flanagan and Patrice Godefroid. Dynamic partial-order reduction for model checking software. In Jens Palsberg and Martín Abadi, editors, *Proceedings of the 32nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2005, Long Beach, California, USA, January 12-14, 2005*, pages 110–121. ACM, 2005. DOI: 10.1145/1040305.1040315. URL https://doi.org/10.1145/1040305.1040315.

[23] 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. URL https://doi.org/10.1007/3-540-60761-7.

[24] Patrice Godefroid. Model checking for programming languages using verisoft. In Peter Lee, Fritz Henglein, and Neil D. Jones, editors, *Conference Record of POPL'97: The 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, Papers Presented at the Symposium, Paris, France, 15-17 January 1997*, pages 174–186. ACM Press, 1997. DOI: 10.1145/263699.263717. URL https://doi.org/10.1145/263699.263717.

[25] Susanne Graf and Hassen Saïdi. Construction of abstract state graphs with PVS. In Orna Grumberg, editor, *Computer Aided Verification, 9th International Conference, CAV '97, Haifa, Israel, June 22-25, 1997, Proceedings*, volume 1254 of *Lecture Notes in Computer Science*, pages 72–83. Springer, 1997. DOI: 10.1007/3-540-63166-6_10. URL https://doi.org/10.1007/3-540-63166-6_10.

[26] Orna Grumberg, Edmund M. Clarke, and Doron A. Peled. Model checking. In *International Conference on Foundations of Software Technology and Theoretical Computer Science; Springer: Berlin/Heidelberg, Germany*, 1999.

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

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

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

[30] Java SE 8 Edition. The Java Language Specification. Language specification, Sun Microsystems, May 2015. URL `https://docs.oracle.com/javase/specs/jls/se8/html/index.html`.

[31] Bengt Jonsson, Magnus Lång, and Konstantinos Sagonas. Awaiting for godot: Stateless model checking that avoids executions where nothing happens. In *Proceedings of the 22nd Conference on Formal Methods in Computer-Aided Design – FMCAD 2022*, pages 284–293. TU Wien Academic Press, October 2022. URL `https://doi.org/10.34727/2022/isbn.978-3-85448-053-2_35`.

[32] Antoni W. Mazurkiewicz. Trace theory. In Wilfried Brauer, Wolfgang Reisig, and Grzegorz Rozenberg, editors, *Petri Nets: Central Models and Their Properties, Advances in Petri Nets 1986, Part II, Proceedings of an Advanced Course, Bad Honnef, Germany, 8-19 September 1986*, volume 255 of *Lecture Notes in Computer Science*, pages 279–324. Springer, 1986. DOI: `10.1007/3-540-17906-2\_30`. URL `https://doi.org/10.1007/3-540-17906-2_30`.

[33] Jeroen Meijer, Gijs Kant, Stefan Blom, and Jaco van de Pol. Read, write and copy dependencies for symbolic model checking. In Eran Yahav, editor, *Hardware and Software: Verification and Testing - 10th International Haifa Verification Conference, HVC 2014, Haifa, Israel, November 18-20, 2014. Proceedings*, volume 8855 of *Lecture Notes in Computer Science*, pages 204–219. Springer, 2014. DOI: `10.1007/978-3-319-13338-6\_16`. URL `https://doi.org/10.1007/978-3-319-13338-6_16`.

[34] Robert H. B. Netzer and Barton P. Miller. What are race conditions? some issues and formalizations. *LOPLAS*, 1(1):74–88, 1992. DOI: `10.1145/130616.130623`. URL `https://doi.org/10.1145/130616.130623`.

[35] William T. Overman and Stephen D. Crocker. Verification of concurrent systems: Function and timing. In Carl A. Sunshine, editor, *Protocol Specification, Testing and Verification, Proceedings of the IFIP WG6.1 Second International Workshop on Protocol Specification, Testing and Verification, Idyllwild, CA, USA, 17-20 May, 1982*, pages 401–409. North-Holland, 1982.

[36] Doron A. Peled. Ten years of partial order reduction. In Alan J. Hu and Moshe Y. Vardi, editors, *Computer Aided Verification, 10th International Conference, CAV '98, Vancouver, BC, Canada, June 28 - July 2, 1998, Proceedings*, volume 1427 of *Lecture Notes in Computer Science*, pages 17–28. Springer, 1998. DOI: `10.1007/BFb0028727`. URL `https://doi.org/10.1007/BFb0028727`.

[37] Tian Huat Tan, Yang Liu, Jun Sun, and Jin Song Dong. Verification of orchestration systems using compositional partial order reduction. In Shengchao Qin and

Zongyan Qiu, editors, *Formal Methods and Software Engineering - 13th International Conference on Formal Engineering Methods, ICFEM 2011, Durham, UK, October 26-28, 2011. Proceedings*, volume 6991 of *Lecture Notes in Computer Science*, pages 98–114. Springer, 2011. DOI: 10.1007/978-3-642-24559-6_9. URL https://doi.org/10.1007/978-3-642-24559-6_9.

[38] Tamás Tóth, Ákos Hajdu, András Vörös, Zoltán Micskei, and István Majzik. Theta: a framework for abstraction refinement-based model checking. In *2017 Formal Methods in Computer Aided Design (FMCAD)*, pages 176–179. IEEE, 2017.

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

[40] Antti Valmari. Stubborn sets for reduced state space generation. In Grzegorz Rozenberg, editor, *Advances in Petri Nets 1990 [10th International Conference on Applications and Theory of Petri Nets, Bonn, Germany, June 1989, Proceedings]*, volume 483 of *Lecture Notes in Computer Science*, pages 491–515. Springer, 1989. DOI: 10.1007/3-540-53863-1_36. URL https://doi.org/10.1007/3-540-53863-1_36.

[41] Björn Wachter, Daniel Kroening, and Joël Ouaknine. Verifying multi-threaded software with impact. In *Formal Methods in Computer-Aided Design, FMCAD 2013, Portland, OR, USA, October 20-23, 2013*, pages 210–217. IEEE, 2013. URL https://ieeexplore.ieee.org/document/6679412/.

[42] Chao Wang, Zijiang Yang, Vineet Kahlon, and Aarti Gupta. Peephole partial order reduction. In C. R. Ramakrishnan and Jakob Rehof, editors, *Tools and Algorithms for the Construction and Analysis of Systems, 14th International Conference, TACAS 2008, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2008, Budapest, Hungary, March 29-April 6, 2008. Proceedings*, volume 4963 of *Lecture Notes in Computer Science*, pages 382–396. Springer, 2008. DOI: 10.1007/978-3-540-78800-3_29. URL https://doi.org/10.1007/978-3-540-78800-3_29.