

TDK-dolgozat

Csanád Telbisz

2024.



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

Efficient Certifiable Verification of Concurrent Software

Scientific Students' Association Report

Author:

Csanád Telbisz
VI. évfolyam

Advisors:

Levente Bajczi,
PhD student

Dániel Szekeres,
PhD student

dr. András Vörös,
associate professor

2024.

Contents

Kivonat	1
Abstract	2
1 Introduction	3
2 Preliminaries	7
2.1 Satisfiability Modulo Theories	7
2.2 Weak Memory Models	8
2.3 Partial Orders	9
2.4 Related Work	10
2.4.1 Verification of Concurrent Software with Partial Orders	10
2.4.2 Validation of Verification Results	11
3 Ordering Consistency Methods	12
3.1 Verification with Partial Orders	12
3.1.1 Symbolic Encoding of Multi-threaded Programs	13
3.1.2 Ordering Consistency Theory	14
3.2 Automatic Conflict Detection	16
3.2.1 Over-Approximation of the Happens-Before Relation	16
3.2.2 Bounded Cycles in the Potential Happens-Before Relation	20
4 Correctness Witnesses and Validation	21
4.1 Overview of the Approach	21
4.2 Validation Process	21
4.2.1 Boolean Abstraction of Ordering Consistency Theory Formulae	22
4.2.2 Formal Statement of the Validation Problem	22
4.3 Checking Conflict Clauses	24
4.4 Limitations of the Approach	27
5 Experimental Evaluation	28

5.1	Evaluating Automatic Conflict Detection	28
5.1.1	Research Questions and Experiment Setup	28
5.1.2	Experiment Results	29
5.2	Evaluating Validation with Conflict Clauses	31
5.2.1	Research Questions and Experimental Setup	31
5.2.2	Experiment Results	32
5.3	Threats to Validity	33
6	Conclusion	35
	Acknowledgements	36
	Bibliography	37

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 teszt-lefedettség elérése még nagyobb kihívást jelent, a naiv verifikációs technikák pedig gyakran gyakorlatilag használhatatlanná válnak.

Számos ígéretes modellellenőrzési technika létezik a párhuzamos programok formális verifikációjára, melyek a programot és az ellenőrzött követelményt SMT (Satisfiability Modulo Theories) problémaként kódolják az érvelést az SMT megoldók erejére bízva. Ezeknek a többszálú szoftverek verifikációjára szolgáló eljárásoknak a lényege, hogy a párhuzamos szálak eseményeinek (változóhozzáféréseinek) konzisztens sorrendezését igyekeznek biztosítani. Olyan tulajdonságokat gyűjtenek, melyek a párhuzamos viselkedés megsértéséhez vezetnek: ezzel kizárják az érvénytelen program lefutásokat a lehetséges hibához vezető végrehajtások halmazából. Noha az SMT megoldók sok szempontból hatékonyak, mégis csak általános célú eszközök, így számos lehetőség van doménspecifikus optimalizálásra. Dolgozatomban egy ilyen kiegészítést javaslok, mely képes megtalálni az ütemezési következetlenségeket és ezzel bővíteni a kódolt SMT formulát még az SMT megoldó eljárás megkezdése előtt. A megoldó keresési tere ezáltal jelentősen csökkenthető, a verifikációs teljesítmény pedig jelentősen javítható.

A formális verifikáció eredményének közlésével egyetemben bizonyos matematikai bizonyíték szolgáltatása és ezen verifikációs tanúsítvány validálása szintén aktív kutatási terület. A többszálú programok helyességbizonyítékaira vonatkozó koncepciók gyakorlati alkalmazhatósága ugyanakkor meglehetősen korlátozott. Munkámban ismertetek egy új módszert a párhuzamos programok verifikációjához szükséges bizonyítékok megadására és validálására. A megközelítés a fenti ütemezési konzisztencia biztosításán alapuló verifikáció alapgondolatából indul ki. Bemutatom, hogy az érvénytelen végrehajtásokat jellemző tulajdonságok hogyan használhatók a program helyességének bizonyítékaként, és hogyan lehet ezeket validálni.

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 often become practically infeasible as a result of this complexity.

Lately, promising model checking approaches have been developed for the formal verification of concurrent programs that encode the program and the verified property as a Satisfiability Modulo Theories (SMT) problem leaving the reasoning to the strength of SMT solvers. The idea of these methods for concurrent software verification is to ensure that events (variable accesses) of concurrent threads are consistently ordered. Properties are collected that characterize invalid scheduling of concurrent threads and used to exclude invalid program executions. While SMT solvers are optimized in many ways, they remain general purpose tools leaving the room for domain-specific optimization. I propose such an optimization that can find scheduling inconsistencies and strengthen the encoding formula before starting the SMT solving procedure. This way, the solver search space can be greatly reduced and the overall verification performance can be significantly improved.

Attaching a certificate of correctness to the result of formal verification and the validation of these verification witnesses have been intensely studied. However, practical applicability of these concepts concerning correctness proofs of concurrent software is still very limited. In this work, I propose a novel way to provide and validate correctness witnesses for concurrent program verification. The approach is based on the core concept of event ordering consistency-based verification. I show how those properties characterizing invalid executions can be used as witnesses for the correctness of the program and how they can be validated.

Chapter 1

Introduction

The presence of computerized solutions in industrial systems and in everyday life highlights the need for reliable software. In safety critical systems where faults are intolerable, guaranteed - mathematical - correctness is often required which cannot be offered by conventional verification techniques such as testing. Formal software verification - a method that can mathematically prove safety guarantees for software - has been a field of active research in the last few decades [29, 16, 41, 1]. Today, multi-core processors are available for various targets, from personal computers to safety-critical systems. In a critical system, the increased computing capacity of a multi-core processor may add extra resources to the critical functionalities. Nonetheless, functionally correct behavior is still crucial.

In formal software verification, verified properties are typically reachability criteria (whether a certain error state is reachable by any execution of the program), memory-safety (no memory leak or memory handling issue), or termination (whether all executions of the program terminate). In the scope of this work, reachability criteria are considered exclusively. Model checking is a formal verification technique where properties are verified by analyzing the state space of the program [29]. Generally, the input of a model checking algorithm is a *model* and a *formal requirement* (or *specification*). The output is a verdict: the model is either *safe* or *unsafe*. The input model in the scope of this paper is a multi-threaded program. For reachability analysis, specific points of the verified program are marked as unsafe by using *assertions* as formal requirements. If an assertion fails in any possible program execution, the reachability criterion is said to be violated.

Verification with Partial Orders. Formal verification methods take a program and a safety property and try to prove or refute that the property holds for all possible program executions. The verification of concurrent software has always been a challenge due to the great number of possible thread interleavings. For concurrent programs, the aim of the verification is to find a program execution (a specific interleaving of concurrent instructions) that violates the safety property of the verification; or to prove that such execution does not exist. Efficient algorithms for model checking multi-threaded programs often involve reasoning about *happens-before* relations which define a partial order of concurrent program instructions [3, 2]. The program is symbolically encoded along with some scheduling constraints based on the possible happens-before relations. The encoding is often expressed as a Satisfiability Modulo Theories (SMT) formula. The key concept of these bounded model checking approaches is to use the models (variable assignments) provided by the SMT solver to analyze possible partial orders and prevent scheduling inconsistencies that may arise in these models during the verification. A scheduling inconsistency corresponds to a cycle in the happens-before relation, since a valid execution of concurrent

threads must have a linearization of instructions. Any model found by the SMT solver whose interpretation leads to a happens-before cycle does not represent a valid execution of the multi-threaded program. Inconsistencies or conflicts in the model can be converted into a conflict clause that can be used to exclude invalid program executions when looking for an execution that violates the requirement of the verification [45, 32].

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

My work aims to considerably reduce the model search space of the SMT solver by constraining the encoding formula. I achieve this by analyzing the program structure and possible partial orders of program instructions, and collect possible scheduling inconsistencies, i.e., cycles that may arise in the happens-before relations. Figure 3.2 shows simple structures of partial orders that lead to a cycle in the happens-before relation, and therefore represent an invalid partial order of concurrent program events. Conflict clauses formulated from these conflicts are appended to the encoding formula. My contribution enables to greatly improve the performance and scalability of the verification. I formulate a general framework for reasoning with possible partial orders of concurrent program instructions. Then, I propose a novel approach and a specific algorithm that looks for possible conflicts of bounded size. On one hand, applying a bound keeps a limit on the size of the encoding formula, on the other hand, my evaluation shows that a great proportion of invalid behavior is already excluded with the small bound.

While most works try to reduce the size of the encoding formula [45, 32], my approach performs early computations and instead of trying to minimize the size of the formula, preserves some useful information about partial orders that accelerates the SMT solving procedure. My method is still lazy in the sense that basic scheduling constraints are omitted similarly to typical lazy approaches [32, 43]. On the other hand, I inject some eagerness by encoding conflicting partial orders that cannot occur together in a valid program execution. I select these eager constraints in a way that is useful for the SMT solver which instead of putting an overhead on the decision procedure, accelerates it considerably.

While reading the literature and analyzing algorithms in published papers, I discovered a theoretical problem with a published verification algorithm described in [43]. Since the problem is complex, its presentation is technical and does not fit in the limits of this paper. We wrote a paper about the problem and proposed corrections with my supervisors and submitted it to a prestigious international conference¹. The submitted paper is attached as a supplement material. As a consequence of this discovered problem, the problematic algorithm is insufficient for model checking concurrent programs under sequential consistency. I propose a way to circumvent the limitations of the algorithm by extending with other techniques.

Certified Verification. Software verification algorithms and verifier implementations tend to be very complex, thus prone to contain bugs which reduces the reliability of the given proofs and limits the application and spread of formal verification techniques in the industry. An emerging trend in recent years is to develop validation algorithms and

¹The name of the conference is omitted due to the ongoing double-blind review at the time of the submission of this work.

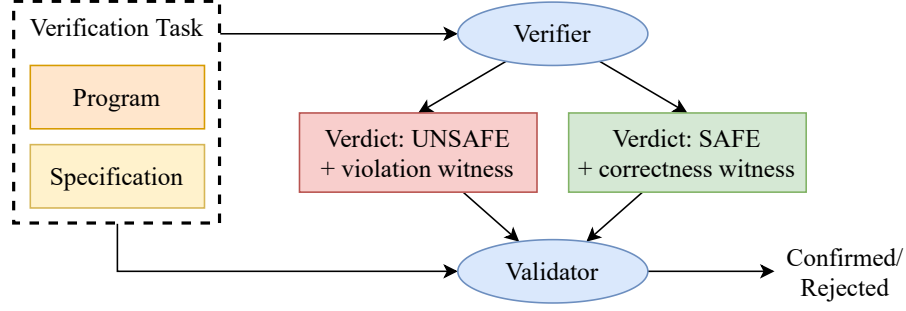


Figure 1.1: Formal Verification with Witness Validation.

tools in parallel to verifiers to check the verification verdicts of verifier tools [11, 15]. In a typical workflow, the verifier gets a program and a safety property as inputs and produces a verdict whether the program satisfies the property. Furthermore, the verifier is also required to produce a *witness* that can be useful for reestablishing the proof of the verifier. In case of an unsafe program, a *violation witness* has to be given (e.g., an error trace leading to the erroneous state or a failing test case). For safe programs, some kind of proof, a *correctness witness* is expected (e.g., loop invariants [12]). Typically, it is easier to produce witnesses for a property violation. The validator (which is ideally a completely independent implementation) takes the same verification task as input accompanied by the witness produced by the verifier. The validator has to confirm or refute the verdict of the verifier. SV-COMP, a competition on software verification recently started hosting a validation track where verdicts and witnesses produced by the verifiers in the competition have to be validated [10]. Furthermore, a correct verification verdict only increases the score of a verifier if it is validated by a validator.

In a multi-threaded program, the extra complexity due to concurrent behavior involves even more complicated algorithms [1, 38, 3] which highlights the need for validation of verification results even more. On the other hand, there are practically no existing techniques providing and validating proofs of correctness for concurrent programs. This is showcased by the fact that there is no proper validator for safe results in the concurrency category of SV-COMP [10]. This work aims to provide methods for the validation of safe concurrent programs with certain properties. I show how the conflict clauses found during verification can be used as witnesses for the correctness of a program.

Contributions. My work includes the following contributions:

- I discovered a theoretical problem with a published algorithm as mentioned above. The problem description and detailed fixes are attached as a supplement material. I formulate a sound verification algorithm that does not suffer from this problem in Section 3.1. While this proposed fix builds on existing ideas, the combination of these concepts in the way that I describe can be considered a novel contribution.
- I propose an optimization to partial order-based verification by automatically discovering scheduling conflicts. The method is presented and proven to be sound in Section 3.2. This contribution is also submitted to an international conference (another one) also undergoing review right now.
- I propose a way to provide useful information about concurrent behavior in a verification witness facilitating certified verification of multi-threaded programs.

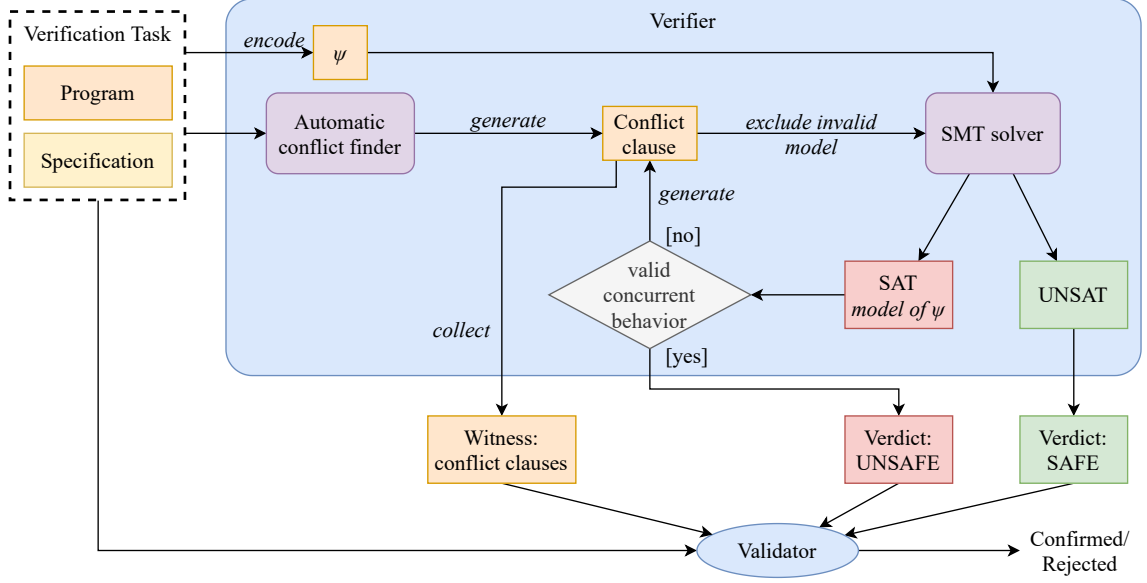


Figure 1.2: Formal Verification with Witness Validation.

- I created an implementation of my theoretical contributions as an open-source extension of the verification tool THETA [44]. My implementation achieves the best result among state-of-the-art bounded model checkers in terms of the number of solved problems on a large set of benchmark programs.

Figure 1.2 summarizes the approach with my contributions. The verification task is encoded into an SMT formula which is given to the SMT solver. If the formula is unsatisfiable, then the program is safe. Otherwise, we have to check whether the model returned by the SMT solver represents a valid execution of concurrent instructions. My first contribution concerns this decision. If the model given by the SMT solver represents a valid program execution indeed, then the program is unsafe. Otherwise, a conflict clause is generated which is given to the SMT solver to exclude this inconsistent model. This process is repeated until a safe or an unsafe verdict is reached. My second contribution corresponds to the *Automatic conflict finder* box: my goal is to automatically² discover conflicting situations and generate conflict clauses before giving the verification formula to the SMT solver. My third contribution is the extension of this scheme with a validation phase: I propose to use the conflict clauses found during verification as part of the witness, and I show how this information can be used effectively by the validator.

The paper is structured as follows: in Chapter 2, a brief description of necessary preliminary knowledge is given. I present the verification algorithm and the proposed optimization in Chapter 3. The method to provide and validate witnesses is described in Chapter 4. Finally, I evaluate the presented approaches in Chapter 5.

²The word *automatic* only refers to the fact that no SMT solver is needed for this approach. The decision diamond in Figure 1.2 is also automatic in the sense that it does not require human interaction.

Chapter 2

Preliminaries

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

2.1 Satisfiability Modulo Theories

The *satisfiability modulo theories* (SMT) problem is the decision problem of determining the satisfiability of a first-order logic formula given some background theories [22, 8]. For example, deciding whether the formula $(x < y - 1 \wedge x = y + 1) \vee x > -2 * y$ has any real solutions is an SMT problem with linear arithmetic as a theory. SMT is expressive enough to be useful when formalizing a wide range of mathematical or engineering problems. On the other hand, SMT is well-defined and constrained which makes it possible to have efficient algorithms to tackle SMT problems. Therefore, SMT is often used as a base of model checking algorithms [8, 3, 43].

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

Several algorithms exist for SMT solving, one traditional method is DPLL(T) [24, 8] which we use to present the concepts in this paper. Other SMT solving procedures often share similar ideas (e.g., conflict clauses as described below), so the method presented in this paper can be adapted to other techniques as well. The DPLL(T) algorithm initially takes the Boolean abstraction $\mathcal{B}(\Phi)$ of the input SMT formula Φ , where a Boolean abstraction means that each symbol in the original formula is simply interpreted as a propositional logic symbol forgetting about the theory (theory formulae become propositional variables). The Boolean abstraction of the above linear arithmetic example formula could be $(p \wedge q) \vee r$ where the propositional logic variables p , q , and r represent the theory formulae $x < y - 1$, $x = y + 1$, and $x > -2 * y$, respectively. A definition of Boolean abstraction tailored to the needs of this work is introduced in Section 4.2.1.

DPLL(T) uses a SAT solver to check the satisfiability of $\mathcal{B}(\Phi)$. If it is unsatisfiable, then Φ is necessarily unsatisfiable, too. However, if the SAT solver finds a model M of $\mathcal{B}(\Phi)$, it is not guaranteed to be a valid model of Φ . A theory solver must be used to determine

whether M is consistent with the axioms of the theory. If an inconsistency is found, the theory solver generates a *conflict clause* c in propositional logic which represents a violation of theory axioms. This way $\mathcal{B}(\Phi) \wedge \neg c$ cannot have the same model M anymore, i.e., the same type of inconsistency is avoided in the future. This procedure can be repeated until either $\mathcal{B}(\Phi)$ augmented with the conflict clauses becomes unsatisfiable or a model is found which is consistent with the theory axioms. In the first case Φ is unsatisfiable, while in the latter case, it is satisfiable with the found model.

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

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

2.2 Weak Memory Models

This work does not really focus on weak memory models, but the accurate theoretical presentation of the paper requires some references to weak memory models. Therefore, we need a few words to have a basic understanding of the concept of memory models. To improve efficiency, most processors apply various kinds of memory reordering. A memory model describes what types of memory or instruction reordering is allowed for a given processor [5]. A weaker memory model allows more types of reordering. The verification problem under a specified memory model $\mathcal{M}$ asks whether there is a sequence of program instructions violating the safety requirement that conforms to $\mathcal{M}$ by respecting the allowed types of instruction reordering. For the presentation of this paper, the only relevant aspect of memory models is the following: if a sequence of instructions is a valid program execution under a memory model $\mathcal{M}_1$, the same sequence is also a valid execution under all memory models $\mathcal{M}_2$ such that $\mathcal{M}_2$ is strictly weaker than $\mathcal{M}_1$ (where strictly weaker means that all types of reordering allowed by $\mathcal{M}_1$ is also allowed by $\mathcal{M}_2$). Therefore if a program is proven safe under the (strictly) weaker memory model $\mathcal{M}_2$, it is also safe under the stronger $\mathcal{M}_1$.

The sequential memory model does not allow any kind of memory reordering. Generally, memory models are often defined via a happens-before relation which specifies for each pair of concurrent instructions in a declarative manner if one must happen before the other or not; that is, if their order is important or not: in the latter case, they may be reordered [30]. Since my work is based on reasoning with happens-before relations, I give a detailed introduction in the next section.

2.3 Partial Orders

In this work, I use an event-based representation of multi-threaded programs [3]. Each memory access is represented by an *event* e characterized by the memory address (which variable or memory location is accessed), access type (*read* or *write*), and a guard condition grd_e (an event is enabled if its guard condition is true). The concurrent behavior of an execution is represented by a *happens-before* relation $\prec$ on events. Intuitively, $e_1 \prec e_2$ means that e_1 must happen before e_2 in any possible program execution where data flows the same way (read events get the same values from the same write events). An execution is *valid* (or consistent) if there is no cycle in $\prec$ ($\nexists e$ such that $e \prec e$) [2], in other words, if there is a possible linearization (or sequentialization) of events respecting $\prec$. The following relations can relate the events of the program (a detailed example of partial orders is given later in Chapter 3, see Figure 3.1):

- The *program order* $\prec_{po}$ is a total order of events of the same thread. It is the order of events respecting the order of instructions in the program code. A thread creation or join event is also related to the first/last event of the thread in $\prec_{po}$ as expected.
- The *read-from* relation $\prec_{rf}$ relates a write event w to a read event r (written $w \prec_{rf} r$) if r reads the value written by w . An important property of this relation is that if $w \prec_{rf} r$ then there cannot be another write event w' to the same memory address between w and r . Otherwise, r could not read the value written by w as it would have been overwritten by w' . This way, we can derive the following two relations.
- A *write-serialization* order (often called a *coherence order* [2] or *store order* [46]) relates two writes $w' \prec_{ws} w$ if there is a read r such that $w \prec_{rf} r$ and w' precedes r . In such a case, w' cannot happen between w and r , so it must precede w as well.
- A *from-read* order relates a read and a write event $r \prec_{fr} w'$ if there is a write w such that $w \prec_{rf} r$ and w' happens after w . Similarly, w' cannot happen between w and r , so it must happen after r as well.

Some partial orders (e.g., from-read) are derived from other partial orders, therefore we need formal rules for derivations, and an extra rule for transitivity. I present an axiomatic system that formally describes the happens-before relation. In fact, I formalize the happens-before relation of *weak sequential consistency* (wSC) [46] which is somewhat weaker than the sequential consistency (SC) memory model. The reason for using the weaker happens-before relation is that wSC is generally easier to handle: deciding whether a program execution conforms to wSC can be done in polynomial time [46] while (surprising as it may seem) the same decision problem for SC is NP-complete [27]. Since my approach discovers cycles in the happens-before relation, I can use that of wSC instead of SC: any cycle in the happens-before relation of wSC can also be found in the happens-before relation of SC as SC is strictly stronger than wSC [46].

Let us define formally the *happens-before* relation and the axioms needed for its definition.

Definition 1 (Happens-Before Relation $\prec$). $e_1 \prec e_2$ holds for events e_1 and e_2 if:

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

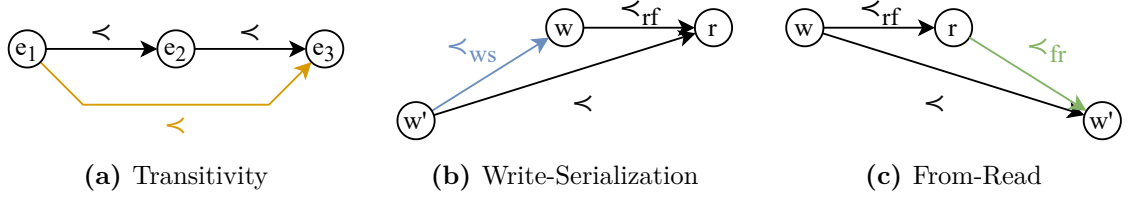


Figure 2.1: Deriving Happens-Before Relations

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

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

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

$$(w \prec_{rf} r) \wedge (w' \prec r) \wedge \text{grd}_{w'} \Rightarrow (w' \prec w).$$

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

$$(w \prec_{rf} r) \wedge (w \prec w') \wedge \text{grd}_w \Rightarrow (r \prec w').$$

I will sometimes emphasize the used derivation step by writing $\prec_{ws}$ or $\prec_{fr}$ for orders derived by Axiom 2 or Axiom 3, respectively. Figure 2.1 visually depicts the derivation rules coloring the derived partial order in each case. Program order and read-from orders come directly from the program code or from the data-flow of an execution, so they cannot be derived from other partial orders, and thus, do not need axioms.

2.4 Related Work

First, I explore the main directions of concurrent software verification concentrating on partial order-based methods, then I give a brief overview of validation methods.

2.4.1 Verification of Concurrent Software with Partial Orders

Many algorithms have been developed for concurrent software verification: initially, most of them were based on exploring the state space or possible steps of the program [40, 23, 1, 38, 28]. In the 2010s, *Alglave et al.* introduced partial order-based symbolic approaches for model checking multi-threaded programs [3]. Early works using partial orders typically include all scheduling constraints in the encoding formula. Each memory event is associated with a clock value, and scheduling constraints are formulated in *integer difference logic*. Alternative solutions use the theory of partial strings for describing partial order constraints [36, 37]. Later, several improvements have been developed for these approaches lazily encoding the scheduling constraints. *Yin et al.* developed an abstraction-refinement method for gradually refining the encoding formula starting without scheduling constraints [45]. *He et al.* proposed a dedicated theory solver integrated into the SMT solver Z3 [21] for reasoning about scheduling constraints [32]. The theory solver is able to perform value propagation and to detect conflicts based on the partial variable assignment of the SMT solver. They also reduce the size of the encoding formula as some constraints can be replaced by reasoning in the theory solver. An improved *pre-ventive* propagation algorithm is introduced by *Sun et al.* [43]. Their algorithm is able to

detect conflicts even before the SMT solver reaches a partial variable assignment implying the conflict; the theory solver prevents conflicting variable assignments by propagating the necessary preventive clauses. While many of these algorithms assume a sequential memory model, extending partial order-based reasoning to weak memory models has long been researched as well [2, 19, 43]. Recently, *Haas et al.* proposed a theory with a theory solver for specifying and handling general memory consistency models [30] with several optimizations [31]. The presented approaches serve as the base algorithms of successful bounded model checkers for concurrent software [33, 45, 20].

Using a dedicated theory solver already significantly constrains the SMT solver during exploring the search space of the formula for a satisfying model [32, 43]. However, the SMT solver may still explore redundant branches of the search space and backtrack unnecessarily with these techniques, since the conflicts are only detected (or prevented) when a corresponding partial variable assignment is reached by the SMT solver. Even the preventive technique [43] is only one step ahead: it can only infer one missing partial order for a conflict, so the other elements of the happens-before relation leading to a cycle still needs to be justified by a corresponding variable assignment. On the other hand, our proposed technique prevents many conflicts right from the beginning. Thus, we can say that our method is even more preventive than the algorithm described in [43]. As our experiments reveal, the clauses obtained from these simple conflicts are often completely instructive for the SMT solver, and only a few or no further propagation steps have to be performed by the theory solver.

Reasoning with over- or under-approximations of happens-before relations have been used by several works [4, 25, 31]. These approximations are useful for enhancing the encoding formula, though these enhancements typically try to reduce the formula by removing unnecessary or lazily computable constraints. Our approach on the other hand adds new constraints to the formula.

2.4.2 Validation of Verification Results

Basic forms of verification witnesses have been used for a long time for safety violations in the form of providing error traces of an unsafe program. Some verifiers can provide instrumented code [42] or generate tests that can reproduce errors [35, 7]. The concept of providing proofs or correctness certificates has also been researched in the last decades [34, 39]. These approaches mainly try to identify invariants of the program, however many of them suffer from the impractically large proof sizes [26]. A standard format for violation witnesses has been first introduced in 2015 [11]. Later, it was extended to support correctness witnesses [13]. This way, verifiers and validators that support this format can easily exchange information about the verified program. Though many concepts can be generalized to the validation of concurrent programs [15], currently there are no specific concurrency-related witness concepts and widely applicable algorithms and validators for safe concurrent programs [10].

Chapter 3

Ordering Consistency Methods

I present the verification algorithm in this chapter along with my improvements. First, I describe the base verification algorithm in Section 3.1 which is a combination of several existing concepts from previous works in the literature [3, 32, 43, 46]. Finally, I present my novel contribution to improve the scalability of the verification approach in Section 3.2.

3.1 Verification with Partial Orders

The verification algorithm is a bounded model checking algorithm requiring a loop-free concurrent program and a safety property. Programs with loops can be handled by unrolling loops: complete loop unrolling can be applied in many cases, while a bounded unrolling may be necessary in some case. In the latter case, the proof of safety is valid up to the loop unroll bound [41]. The program and the property are symbolically encoded into an SMT formula: some constraints come straight from the program instructions, while scheduling constraints ensure that concurrent behavior is properly modeled. A theory solver is integrated into the SMT solver ensuring that scheduling constraints are satisfied.

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

I adapt and combine the methods and ideas of *He et al.* [32], *Sun et al.* [43] and *Zennou et al.* [46] for verifying multi-threaded programs under sequential consistency being recent and efficient partial order-based approaches. While reasoning about program executions conforming to the sequential memory (SC) model is NP-complete [27], *Zennou et al.* observe that reasoning is polynomial for weak sequential consistency (wSC), a slightly weaker memory model than SC [46]. The idea is to reason efficiently under wSC first. If the program is found safe under wSC, it is also safe under the stronger SC model. If an execution violating the safety property is found that is valid under wSC, it must be checked if it is also valid under SC. The algorithm for reasoning under wSC is taken from *Sun et al.* [43]. Checking SC validity is done by applying the encoding of *He et al.* [32] for determining the existence of a total order on write events (being the only difference between SC and wSC).

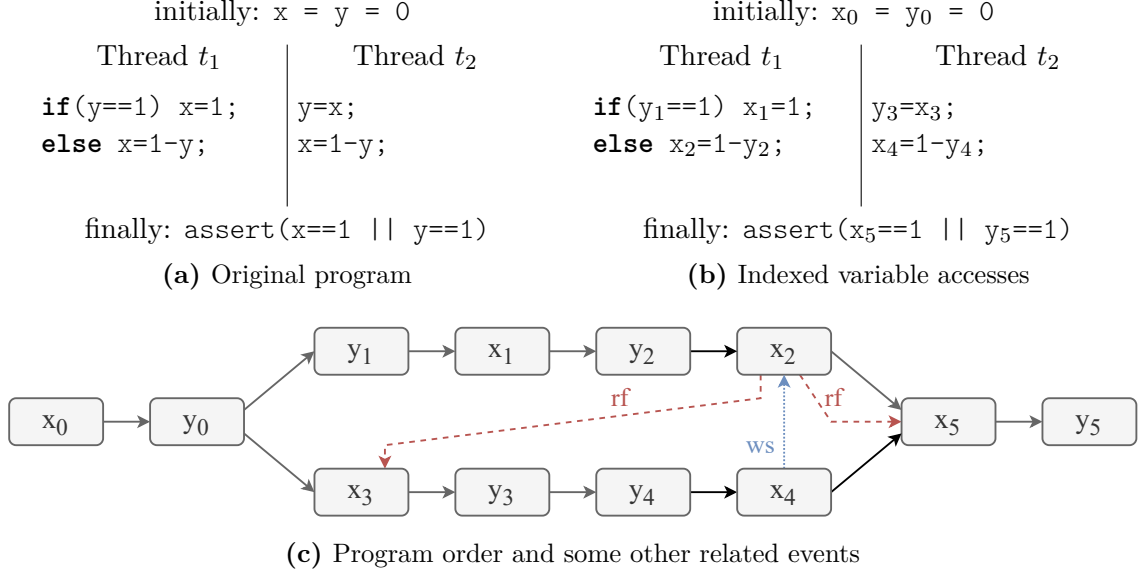


Figure 3.1: Running example

3.1.1 Symbolic Encoding of Multi-threaded Programs

For the symbolic encoding of the program, each variable access is converted to a uniquely indexed version of the variable to have a unique reference for each access¹. Therefore, each event has a unique indexed variable: I write w_{x_i} or r_{x_i} for the write or read event (or e_{x_i} in general) on the indexed variable x_i . By abusing the notation, I simply refer to $grd_{e_{x_i}}$ as grd_{x_i} .

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

As an example, take the simple program from Figure 3.1a with two threads. Figure 3.1b shows the program where variable accesses are marked with unique indices. For example, the guard of the write event w_{x_1} is $grd_{x_1} := (y_1 = 1)$, while $grd_{x_3} := true$. The value assignment of the program is as follows:

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

The program order constraints are encoded into ρ_{po} based on the order of instructions per each thread. In case of branches, we can simply put the branches one after another in the total order since only one branch can be enabled in any valid execution anyway. Furthermore, thread creation and join operations are also reflected in ρ_{po} by for example relating events of the creator thread before the thread creation to events of the created thread. The program order is represented in Figure 3.1c with solid black edges².

To encode read-from constraints, a new Boolean variable $rf_{i,j}^x$ is introduced for each pair of events w_{x_i} and r_{x_j} such that r_{x_j} may read the value written by w_{x_i} . If $rf_{i,j}^x$ is true, then r_{x_j} gets its value from w_{x_i} (i.e., $w_{x_i} \prec_{rf} r_{x_j}$), therefore x_i must be the same value as x_j and both events must be enabled. Formally, we have $rf_{i,j}^x \Rightarrow x_i = x_j \wedge grd_{x_i} \wedge grd_{x_j}$ which is

¹Some works refer to this form as concurrent static single assignment [32] which name is slightly misleading as static single assignment originally means a compilable form of the program with fresh copies of variables per write access [18]. The applied form cannot be compiled and has fresh copies per each access.

²The program order relation is in fact the transitive closure of the solid black edges in Figure 3.1c.

called the *RF-Val* constraint [32]. Also, if a read event is enabled then it must read its value from a write event which leads us to the *RF-Some* constraint: $grd_{x_j} \Rightarrow rf_{i_1,j}^x \vee \dots \vee rf_{i_n,j}^x$ assuming that r_{x_j} can read from $w_{x_{i_1}}$, ..., or $w_{x_{i_n}}$. We also have *RF-Ord* which connects the *rf* variables and the $\prec_{rf}$ relation: $rf_{i,j}^x \Leftrightarrow w_{x_i} \prec_{rf} r_{x_j}$. The above constraints are encoded in the formula ρ_{rf-val} , $\rho_{rf-some}$, and ρ_{rf-ord} respectively.

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

In case where a violating program execution is found by the first stage of the verification, we also need to encode write serialization constraints to check SC conformity. For this purpose, similarly to read-from constraints, a new Boolean variable $ws_{i,j}^x$ is introduced for each pair of same-memory write events. If $ws_{i,j}^x$ is true, then w_{x_i} happens-before w_{x_j} . Formally, we have the following constraints: $ws_{i,j}^x \Rightarrow grd_{x_i} \wedge grd_{x_j}$ called *WS-Cond*; $grd_{x_i} \wedge grd_{x_j} \Rightarrow ws_{i,j}^x \vee ws_{j,i}^x$ called *WS-Some*; and $ws_{i,j}^x \Rightarrow w_{x_i} \prec_{ws} w_{x_j}$ called *WS-Ord* [32].

Finally, we have a correctness property to encode. Since we are looking for safety violations, the correctness property is negated to obtain an error property. The focus of this paper is the verification of error reachability: this way, the error formula ρ_{err} is the guard condition of the marked error in the program. This is often expressed as an assertion in the program: the error formula is the negation of the asserted expression (in conjunction with the guard of branches needed to reach the assert statement). The error property in the example is the negation of the asserted expression, that is, $\neg(x_4 = 1 \vee y_3 = 1)$.

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

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

where ψ_{ssa} only contains expressions that standard SMT solvers can handle, while ψ_{ord} contains the ordering formulae that need a specific theory solver (note the $\prec$ symbols in ψ_{ord} which is not a symbol of a usual SMT theory). The whole formula then becomes $\Psi = \psi_{ssa} \wedge \psi_{ord}$. If Ψ is satisfiable, then a satisfying model gives a valid execution that leads to an error. However, if Ψ is unsatisfiable, there are no valid executions where the error property is also met, thus the program is safe [32]. I use the formulae without the parts in brackets for the first check under wSC. If it turns out to be unsatisfiable, then the program is safe under both wSC and SC. However, if it is satisfiable, I extend the formulae with the bracketed parts to check under SC.

3.1.2 Ordering Consistency Theory

For the theory solver that can handle ψ_{ord} , we need to define a set of axioms for propagation rules and conflict detection as we have seen in Section 2.1. Fortunately, I have already defined Axiom 1, Axiom 2, and Axiom 3 which can be used as the axioms of the theory solver. Performing propagation and conflict detection with these axioms allows us to check if there is any execution violating the safety property that is valid under the wSC model [46, 43]. If the constraints for a total order of writes is also included in the encoding formula in the potential second stage of the verification algorithm, then these axioms can be used to check the existence of a violating execution under SC [32].

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

The theory solver has three main components: theory propagation, consistency checking, and conflict clause generation. Theory propagation is applied when a new ordering variable rf is assigned *true* by the SMT solver. Then the theory axioms are used to derive all possible new orders. As theory propagation is not crucial for the main contribution of this paper, I refrain from going into further details regarding the propagation algorithm. I refer the interested reader for several possible propagation algorithms described in [32, 43]. Consistency checking is also straightforward: we have to check whether $e \prec e$ holds for any event e in the current partial variable assignment after propagating event orders. An order $e \prec e$ is equivalent to a self-loop in the event graph, so consistency checking amounts to checking the presence of a self-loop in the event graph. As conflict clause generation is crucial for the methods presented in this work, I elaborate a bit more on that.

If an inconsistency is found by the theory solver, it must generate a conflict clause that prevents this inconsistency. Therefore, the theory solver stores derivation reasons for each derived happens-before order as first-order logic formulae [43]. The following definition defines reasons of orders using the notation of the corresponding axioms where needed:

Definition 2. The reasons for different types of orders are defined as:

- $reason(e_1 \prec_{po} e_2) := true$, since these orders come from the program structure and must be satisfied at all times,
- $reason(w_{x_i} \prec_{rf} r_{x_j}) := rf_{i,j}^x$, that is the corresponding Boolean variable,
- $reason(w_{x_i} \prec_{ws} w_{x_j}) := ws_{i,j}^x$, that is the corresponding Boolean variable,
- the reason for an order derived by Axiom 1 is:

$$reason(e_1 \prec e_3) := reason(e_1 \prec e_2) \wedge reason(e_2 \prec e_3),$$
- the reason for an order derived by Axiom 2 is:

$$reason(w' \prec_{ws} w) := reason(w \prec_{rf} r) \wedge reason(w' \prec r) \wedge grd_{w'},$$
- the reason for an order derived by Axiom 3 is:

$$reason(r \prec_{fr} w') := reason(w \prec_{rf} r) \wedge reason(w \prec w') \wedge grd_{w'}.$$
 ▪

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

As an example, consider a partial variable assignment corresponding to Figure 3.1c in our running example: where $rf_{2,5}^x$ and $rf_{2,3}^x$ have been assigned *true*. The solver can now perform a write-serialization derivation on the events w_{x_4} , w_{x_2} , and r_{x_5} since the preconditions of Axiom 2 hold: $w_{x_2} \prec_{rf} r_{x_5}$, $w_{x_4} \prec_{(po)} r_{x_5}$, and w_{x_4} is always enabled. So the order $w_{x_4} \prec_{ws} w_{x_2}$ is inferred and added to the event graph as Figure 3.1c also demonstrates. Then, we can find a loop in the event graph, which conflict is also found as a self-loop (e.g., $w_{x_2} \prec w_{x_2}$) by the theory solver after applying the transitivity axiom

a few times. Then, the following conflict clause is generated:

$$\begin{aligned} & \text{reason}(w_{x_2} \prec_{rf} r_{x_3}) \wedge \text{reason}(r_{x_3} \prec_{po} w_{x_4}) \wedge \text{reason}(w_{x_4} \prec_{ws} w_{x_2}) = \\ & \text{reason}(w_{x_2} \prec_{rf} r_{x_3}) \wedge \text{true} \wedge (\text{reason}(w_{x_2} \prec_{rf} r_{x_5}) \wedge \text{reason}(w_{x_4} \prec_{po} r_{x_5}) \wedge \text{grd}_{x_4}) = \\ & rf_{2,3}^x \wedge rf_{2,5}^x \end{aligned}$$

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

3.2 Automatic Conflict Detection

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

It has been introduced in Section 3.1.2 how the theory solver finds conflicts, inconsistencies in the scheduling constraints. The analysis of conflicts found in real programs revealed that these conflicts are dominantly simple in the sense that the self-loops found in the event graph are typically derived from only a few other relations. Having such simple conflicts motivates a pre-processing algorithm that can easily find these conflicts without the computational overhead of the SMT solving procedure. The idea of my proposed algorithm is to explore the program structure and retrieve possible conflicts of bounded size in the number of derivation axioms necessary to obtain them.

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

3.2.1 Over-Approximation of the Happens-Before Relation

At a pre-processing stage, we cannot speak about existing conflicts since they arise during the SMT solving procedure when *rf* variables get assigned by the SMT solver, new happens-before relations are derived by the ordering consistency theory solver using the axioms of Section 3.1.2, and self-loops are found in the event graph. Before starting the SMT solving, the values of *rf* variables are unknown since they are variables. Therefore, no pairs of events are *rf*-related, axioms have not been applied, and actual conflicts have not been found up to the point when my proposed algorithm is executed. Instead, the algorithm deals with a *potential rf* relation and tries to find possible conflicts. I use the $<$ symbol for *potential* happens-before relations³ to highlight the semantic difference from the real happens-before relation. I also define this potential relation similarly to the original happens-before relation. First, I introduce the potential program order and potential read-from relations:

³Some related works use the *may* prefix for the over-approximation of certain relations or sets, or some alternative terminology [31]. Instead, I opted for the *potential* prefix to avoid the slightly ambiguous meaning of *may* and the grammatically dreadful *may-happens-before* name.

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

Definition 3 (Potential Happens-Before $<$). $e_1 < e_2$ holds for events e_1 and e_2 if:

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

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

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

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

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

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

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

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

Example 1. To illustrate the difference between $<$ and $\prec$, take our running example from Figure 3.1. Events x_2 and x_3 are *rf*-related ($x_2 \prec_{rf} x_3$) in the execution $x_0, y_0, y_1, y_2, \textcolor{red}{x_2}, \textcolor{red}{x_3}, y_3, y_4, x_4, x_5, y_5$ (with x_1 disabled). However, $x_2 \not\prec_{rf} x_3$ in the trace $x_0, y_0, y_1, \textcolor{red}{x_3}, y_3, y_2, \textcolor{red}{x_2}, y_4, x_4, x_5, y_5$. Since x_3 may read the value written by x_2 (as in the first case), $x_2 <_{rf} x_3$ universally.

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

Even though some elements of the potential happens-before relation might not materialize as an element of the real happens-before relation of a valid execution, the found potential happens-before loops cannot occur in any valid execution of the program. Thus, we can use these loops similarly to infer conflict clauses and supply the negations of these conflict clauses to the SMT solver as assertions. To convert the found loops in $<$ to propositional

formulae, reasons can be defined similarly to Definition 2. I intentionally omit the *ws* variables and the explicit encoding of write serialization constraints in this section to keep the approach on a more general level: this way we only discover loops that are conflicting with both the wSC and the SC models.

Definition 4. The reasons for different types of the potential happens before relation are:

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

This definition is indeed the same syntactically for $<$ as Definition 2 for $\prec$. Note that even guard conditions are used similarly and they are not omitted from the reason formulae even though guards are omitted from the axioms for $<$. This is due to the semantic difference of *reasons* used in the two contexts. For the original happens-before relation, $reason(e_1 \prec e_2)$ refers to the preconditions that induce $e_1 \prec e_2$. For the potential happens-before relation, $reason(e_1 < e_2)$ does not refer to the preconditions inducing $e_1 < e_2$; it refers to the preconditions that make the potential $e_1 < e_2$ materialize as a real $e_1 \prec e_2$. The following lemma formalizes this connection between *reasons* for $<$ and the original happens-before relation. The subsequent theorem states the soundness of my approach.

Lemma 1. If $reason(e_1 < e_2)$ is true in a model of the formula, then $e_1 \prec e_2$ can be derived in this model. ▪

Proof. I prove the lemma by structural induction on the definition of *reason*. For this proof, it is useful to imagine $reason(e_1 < e_2)$ as a tree structure (I refer to it as the *reason tree*⁴) where $reason(e_1 < e_2)$ is the root node, each non-leaf node is a *reason* of a pair of $<$ -related events, the children of a node are given by Definition 4, and leaf nodes are first-order terms:

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

Looking at Definition 4 of *reasons* for the potential happens-before relation, we can observe that $reason(e_1 < e_2)$ is in fact a large logical *and* expression of the leaf terms of its *reason tree*. If $reason(e_1 < e_2)$ is *true*, then all leaf terms must also be *true* due to the nature of the *and* operator. Thus, we can start from the leaves of the *reason tree* and prove by induction for each (non-leaf) node $reason(f_1 < f_2)$ that $f_1 \prec f_2$ can be derived. Reaching the root of the *reason tree* proves that $e_1 \prec e_2$ can be derived.

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

⁴Similar trees called derivation trees are defined in Section 4.3. Due to the semantic difference between *reasons* for potential and original happens-before relations, I do not call trees in this proof derivation trees.

1. The node is $reason(f_1 <_{po} f_2)$:
 $reason(f_1 <_{po} f_2)$ is always true by definition, and $<_{po} = \prec_{po}$ implies $f_1 \prec f_2$.
2. The node is $f_1 <_{rf} f_2$ with $f_1 = w_{x_i}$ and $f_2 = r_{x_j}$:
 $reason(f_1 <_{rf} f_2) = rf_{i,j}^x$ by definition. The leaf $rf_{i,j}^x$ is true in the model based on our observation for leaves which means that $f_1 \prec_{rf} f_2$ (c.f., RF-Ord constraints).

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

1. $reason(f_1 < f_2)$ with $f_1 < f_2$ derived by Axiom 4 as the transitive closure of $f_1 < f'$ and $f' < f_2$:
We know that $f_1 \prec f'$ and $f' \prec f_2$ can be derived by the induction hypothesis. Applying the transitivity axiom Axiom 1 for the original happens-before relation immediately gives that $f_1 \prec f_2$.
2. $reason(f_1 <_{ws} f_2)$ with $f_1 = w'$ and $f_2 = w$, and $f_1 <_{ws} f_2$ derived by Axiom 5 from $w <_{rf} r$ and $w' < r$ for some read event r :
Again, we know that $w \prec_{rf} r$ and $w' \prec r$ from the $<_{rf}$ base case and the induction hypothesis. We also know that the leaf child $grd_{w'}$ is also *true* based on our observation above. This way, Axiom 2 gives that $w' \prec_{ws} w$, that is, $f_1 \prec f_2$.
3. $reason(f_1 <_{fr} f_2)$. The case is completely analogous with the previous case. $\square$

Theorem 1. Let Ψ be the encoding formula of a program and a safety property, and let E be a set of events e such that $e < e$.

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

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

For the other way, first I observe that no valid execution satisfies $reason(e < e)$ for any self-loop $e < e$. To see this assume that we have a valid execution which satisfies $reason(e < e)$. Lemma 1 implies that $e \prec e$ can be derived in this model. Thus, we reach a contradiction since no valid execution can have a loop in its (original) happens-before relation.

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

Theorem 1 basically says that no valid execution satisfies reasons of self-loops in the potential happens-before relation, therefore the negations of these reasons are properties that generally characterize all possible program executions. Thus, any algorithm that finds any kind of cycles in the potential happens-before relation and uses these conflicts to strengthen the encoding formula as explained above is safe to use and does not change the verification outcome. I formalize such an algorithm in the next section. The algorithm is constrained to find simple conflicts to have a bound for its complexity and a bound for the size of the extended encoding formula which is desirable in practice. However, different and more complicated algorithms looking for complicated cycles could also be used theoretically, and their correctness would also be guaranteed by Theorem 1.

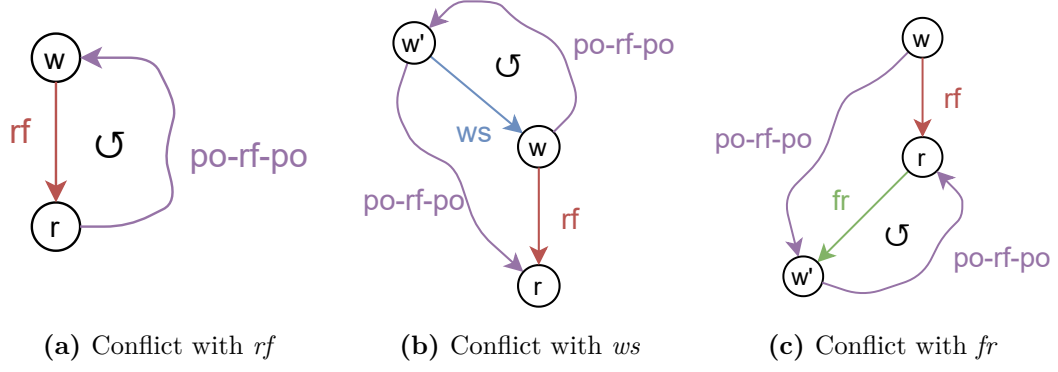


Figure 3.2: Conflicts with basic structures

Algorithm 1: Finding simple conflicts

```

1  $conflicts \leftarrow \{\}$ 
2 foreach  $w <_{rf} r$  do
3   foreach  $w' <_{rf} r'$  do
4     if  $r' <_{po} w$  and  $r <_{po} w'$  then
5        $conflicts \leftarrow conflicts \cup \{w <_{rf} r <_{po} w' <_{rf} r' <_{po} w\}$  // Fig. 3.2a
6   foreach  $w'$  that may write the same memory location as  $w$  do
7     if  $w' <_{po-rf-po} r \wedge w <_{po-rf-po} w'$  then
8        $conflicts \leftarrow conflicts \cup \{w' <_{ws} w <_{po-rf-po} w'\}$  // Fig. 3.2b
9        $conflicts \leftarrow conflicts \cup \{r <_{fr} w' <_{po-rf-po} r\}$  // Fig. 3.2c

```

3.2.2 Bounded Cycles in the Potential Happens-Before Relation

My proposed algorithm looks for potential conflicts, loops in the potential happens-before relation with the basic structures depicted in Figure 3.2. A $po-rf-po$ edge means that it could be replaced by at most one rf and optional po edges before or after this rf . Formally, $e_1 <_{po-rf-po} e_2$ if $e_1 <_{po} e_2$ or there are events e'_1, e'_2 such that $e_1 <_{po} e'_1 <_{rf} e'_2 <_{po} e_2$. To put it simple, the algorithm looks for potential conflicts that may be derived from an rf , a ws , or an fr and another rf or some po relations. Algorithm 1 describes the algorithm used to find potential conflicts before SMT solving. I abuse the notation by writing $w' <_{ws} w <_{po-rf-po} w'$ for example to denote the self-loop $w' < w'$.

Looking at Algorithm 1 or the visualization from Figure 3.2, it is straightforward that Algorithm 1 indeed finds self-loops in $<$. Thus, it is safe to use this algorithm for optimization based on Theorem 1. The impact of the algorithm on verification performance is revealed in Chapter 5.

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

Chapter 4

Correctness Witnesses and Validation

The setup in this chapter consists of a verifier and a validator as described in Chapter 1. In this section, I present how a verifier can provide correctness witnesses and how a validator can validate and use this information in the validation process.

4.1 Overview of the Approach

The input of the verifier is still a concurrent (loop-free) program and a property (namely a reachability property that no error location in the program can be reached). Only safe tasks are interesting for the scope of this chapter as it deals with correctness witnesses. The verifier performs the verification of the program based on the encoding and algorithms described in Chapter 3. Basically, the validator gets the same inputs, however, it also receives some extra information, the witness as input. It has to confirm that the program is indeed safe and accept or refute the witness as a proof for the correctness of the program. The validation process is presented in Section 4.2 and Section 4.3.

The basic concept of the approach is to use the generated conflict clauses from the verification process as the witness for the correctness of the program. Then, this witness, i.e., the set of conflict clauses is provided to the validator as an input. This has its advantages for both the verifier and the validator. First, it does not pose any further computational burden on the verifier, as technically, it simply has to log the generated conflict clauses. Second, this information is extremely useful for the validator, since it does not have to go all the way through the SMT solving procedure (with frequent backtracking from deep branches of the search space) as the conflicts are already provided at the beginning. Naturally, the validator must not simply believe this information coming from the verifier, it has to check it. The rest of the section explains exactly what information has to be checked and how by the validator.

4.2 Validation Process

This section presents the validation process by formally motivating and stating the validation problem. For this, we need a definition of Boolean abstraction.

4.2.1 Boolean Abstraction of Ordering Consistency Theory Formulae

Section 2.1 has given a high-level overview of SMT solving procedures. Even though several background theories might be necessary for solving the encoding formula of a program as defined in Section 3.1.1 (due to potential arithmetic expressions or arrays in the program), I consider the ordering consistency theory as the only important theory from the verification perspective. Ordering consistency theory expressions are simple (being of the form $e_1 \prec e_2$) and do not overlap or interact in any way with other theories. Therefore, I formulate Boolean abstraction by only focusing on the ordering consistency theory without the loss of generality: naturally, it could be generalized to all theories.

Definition 5. The Boolean abstraction $\mathcal{B}(\varphi)$ of a formula φ is obtained by replacing all ordering consistency theory expressions in φ with a fresh Boolean variable. $\cdot$

As an example, $\mathcal{B}(y_3 = x_3 \wedge (r_{x_3} \prec_{po} w_{y_3}) \wedge (w_{x_4} \prec_{ws} w_{x_2}))$ may be $y_3 = x_3 \wedge v_1 \wedge v_2$ where v_1 and v_2 are introduced instead of the theory expressions. Importantly, $\mathcal{B}(\varphi)$ does not contain theory expressions by definition, therefore its satisfiability can be decided without the theory solver (if we had proper Boolean abstraction abstracting the expressions of every theory, it would return a pure SAT formula decidable by simply a SAT solver). The SMT solving procedure iteratively checks the satisfiability of the Boolean abstraction of the program's encoding formula Ψ conjoined with the conflict clauses produced by the ordering consistency theory solver. The conflict clauses also do not contain ordering consistency theory expressions, so this satisfiability check is independent of the theory solver. Thus, if Ψ is unsatisfiable, the result is reached at a point when the conventional SMT formula $\mathcal{B}(\Psi) \wedge \bigwedge_{c \in C} \neg c$ is unsatisfiable with C being the set of conflict clauses found by the ordering consistency theory solver. The conjunction of negations of conflict clauses replaces the information carried by the theory expressions and hidden by Boolean abstraction.

Note that propositional logic connectives can be lifted from the Boolean abstraction as Boolean abstraction only replaces theory expressions: e.g., $\mathcal{B}(\varphi_1 \wedge \varphi_2) = \mathcal{B}(\varphi_1) \wedge \mathcal{B}(\varphi_2)$.

4.2.2 Formal Statement of the Validation Problem

Formally, let C be the set of conflict clauses produced by the verifier. That is, C is a set of reasons, first-order formulae defined in Definition 2. From the program being safe, using the encoding formula Ψ of the program we know that $\mathcal{B}(\Psi) \wedge \bigwedge_{c \in C} \neg c$ is unsatisfiable. To simplify the notation, I use $\tilde{C}$ for $\bigwedge_{c \in C} \neg c$. We can now formulate the following lemma:

Lemma 2. If $\mathcal{B}(\Psi) \wedge \tilde{C}$ is unsatisfiable, then $\psi_{ssa} \wedge \tilde{C}$ is unsatisfiable. $\cdot$

Proof. Recall that $\Psi = \psi_{ssa} \wedge \psi_{ord}$ (c.f., Section 3.1.1). Using this substitution, we obtain $\mathcal{B}(\Psi) \wedge \tilde{C} = \mathcal{B}(\psi_{ssa} \wedge \psi_{ord}) \wedge \tilde{C} = \mathcal{B}(\psi_{ssa}) \wedge \mathcal{B}(\psi_{ord}) \wedge \tilde{C} = \psi_{ssa} \wedge \mathcal{B}(\psi_{ord}) \wedge \tilde{C}$. Note that the Boolean abstraction only replaces ordering consistency theory formulae in this simplified formalization, and ψ_{ssa} does not contain any ordering formulae.

It is also easy to show that if $\psi_{ssa} \wedge \tilde{C}$ has a model M , then M can be trivially extended to a model of $\psi_{ssa} \wedge \mathcal{B}(\psi_{ord}) \wedge \tilde{C}$. To see this, consider $\mathcal{B}(\psi_{ord}) = \mathcal{B}(\rho_{po}) \wedge \mathcal{B}(\rho_{rf-ord})$ obtained by using the definition of ψ_{ord} and the distributivity of the Boolean abstraction with respect to the $\wedge$ symbol. If we apply Boolean abstraction to $\rho_{po} = (e_{11} \prec_{po} e_{12}) \wedge \dots \wedge (e_{n1} \prec_{po} e_{n2})$ (conjunction of all po-related event pairs), we get $\mathcal{B}(\rho_{po}) = v_{\{e_{11} \prec_{po} e_{12}\}} \wedge \dots \wedge v_{\{e_{n1} \prec_{po} e_{n2}\}}$ as the conjunction is part of propositional logic and every theory expression is abstracted

to a Boolean variable. As these new abstract variables do not appear anywhere else in $\psi_{ssa} \wedge \mathcal{B}(\psi_{ord}) \wedge \tilde{C}$, they can be assigned *true* in the extended model without contradiction. Similarly, applying the abstraction to $\rho_{rf-ord} = (rf_1 \Leftrightarrow w_1 \prec_{rf} r_1) \wedge \dots \wedge (rf_m \Leftrightarrow w_m \prec_{rf} r_m)$, we obtain $\mathcal{B}(\rho_{rf-ord}) = (rf_1 \Leftrightarrow v_{w_1 \prec_{rf} r_1}) \wedge \dots \wedge (rf_m \Leftrightarrow v_{w_m \prec_{rf} r_m})$. Again, the new variables only appear once in the whole formula, so each can be assigned the value of the corresponding *rf* variable it is equivalent with. Thus we indeed obtained a model (an extension of M) for $\psi_{ssa} \wedge \mathcal{B}(\psi_{ord}) \wedge \tilde{C}$.

That is, if $\psi_{ssa} \wedge \tilde{C}$ is *satisfiable*, then $\psi_{ssa} \wedge \mathcal{B}(\psi_{ord}) \wedge \tilde{C}$ is *satisfiable*, too. Conversely, if $\psi_{ssa} \wedge \mathcal{B}(\psi_{ord}) \wedge \tilde{C}$ is *unsatisfiable*, then $\psi_{ssa} \wedge \tilde{C}$ is also *unsatisfiable*. As we proved that $\mathcal{B}(\Psi) \wedge \tilde{C}$ being unsatisfiable implies the unsatisfiability of $\psi_{ssa} \wedge \mathcal{B}(\psi_{ord}) \wedge \tilde{C}$, this concludes the proof. $\square$

Based on the lemma, the validator can check the satisfiability of the formula $\psi_{val} := \psi_{ssa} \wedge \tilde{C}$ as it must be unsatisfiable to confirm the verifier's safe verdict. If ψ_{val} is satisfiable, then the verdict and the witness as the output of the verifier is refuted. This is a strict approach to validation. In practice, validators are often implemented to be tolerant in the sense that they accept partial witnesses and try to validate the verdict with that limited information [15]. In our case, this means that if ψ_{val} turns out to be satisfiable, the validator may also compute ψ_{ord} and decide the satisfiability of $\psi_{val} \wedge \psi_{ord}$ with a similar solving procedure as the one that the verifier uses in our setting.

It is an advantage of the strict approach that ψ_{val} does not contain any ordering theory formulae which means that any standard SMT solver can be used to check its satisfiability (even if the solver does not support the ordering consistency theory) reducing the shared algorithms and implementations of the verification and the validation process even more. On the other hand, SMT solvers are typically widely used and thus, well tested, so it may not be too important to use a different solver for the validation. A more tolerant version of the validator can also be used with other verifiers that might use different verification algorithms. Those verifiers may be able to provide some clauses with the required structure though they may not provide all the conflict clauses necessary to make ψ_{val} unsatisfiable.

Lemma 2 is enough if we aim to refute the verdict of a verifier. However, most often, we would like to be convinced about the safety of the verified program. The following lemma establishes that the unsatisfiability of ψ_{val} implies the safety of the program:

Lemma 3. If ψ_{val} is unsatisfiable, then the program is safe. $\cdot$

Proof. For ease of reference, I introduce the following notation for the set of *valid executions* satisfying a formula φ : $tr(\varphi)$.

The conflict clauses in C characterize invalid executions with a loop in the ordering relation, that is, $tr(c) = \emptyset$ for each $c \in C$. Thus appending the negation of a conflict clause to ψ_{ssa} only excludes invalid executions. Thus, if there is a valid execution satisfying ψ_{ssa} , it also satisfies $\psi_{val} = \psi_{ssa} \wedge \tilde{C}$; and it trivially holds vice versa. So $tr(\psi_{ssa}) = tr(\psi_{val})$. Also, $tr(\Psi) = tr(\psi_{ssa} \wedge \psi_{ord}) \subseteq tr(\psi_{ssa})$ trivially again since Ψ is more restrictive than ψ_{ssa} . So we obtained $tr(\Psi) \subseteq tr(\psi_{ssa}) = tr(\psi_{val}) = \emptyset$ since ψ_{val} is unsatisfiable. So there is no valid execution satisfying Ψ , that is, no valid execution that leads to error (remember that the error property is part of Ψ). $\square$

Theorem 2. The program is safe if and only if ψ_{val} is unsatisfiable. $\cdot$

Proof. It follows directly from Lemma 2 and Lemma 3 based on Figure 4.1. $\square$

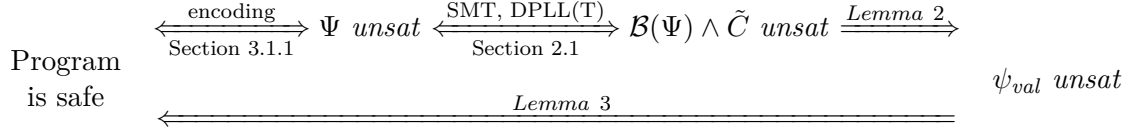


Figure 4.1: Proof structure for Theorem 2

While the set of conflict clauses C are supplied by the verifier, the validator can easily compute the formula ψ_{ssa} itself based on the program structure. This way, the validator indeed has the formula ψ_{val} .

4.3 Checking Conflict Clauses

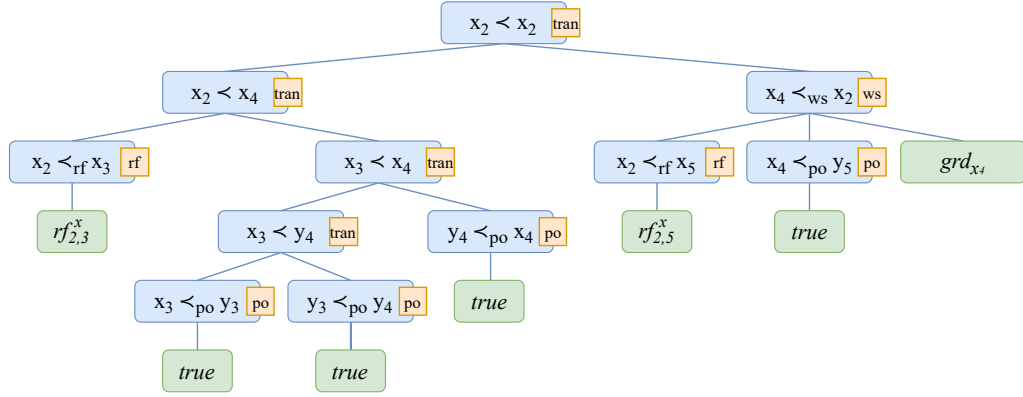
Since the verifier’s algorithm or implementation may contain bugs, the validator must not accept the conflict clauses without checking their validity. To eliminate all trust factor towards the verifier, it must be checked if all given conflict clauses indeed characterize invalid executions with a loop in the ordering relation.

This is not a trivial task if we simply take the raw conflict clauses. Consider the example from the end of Section 3.1.2. The conflict clause $rf_{2,5}^x \wedge rf_{2,3}^x$ does not explicitly show us the (self-)loop in the event graph. To find it, we would have to apply the theory axioms to infer the orders leading to the self-loop. This would be somewhat much cheaper than doing consistency checking during the verification. But it would be best to avoid even this computation. This can be achieved by a minor modification.

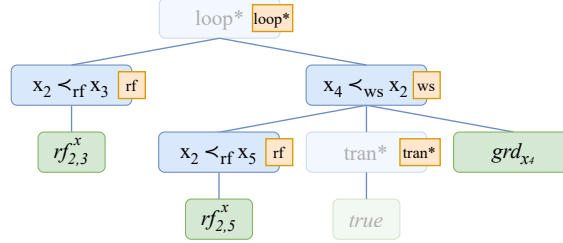
An inconsistency is found by the ordering theory solver in the form of a self-loop $e \prec e$ in the event graph whereupon $reason(e \prec e)$ is generated as a conflict clause. Expanding $reason(e \prec e)$ based on Definition 2 naturally gives us a tree structure that I will refer as the derivation tree. The conflict clause is the conjunction of leaves of the derivation tree. Figure 4.2a depicts derivation tree of the inconsistency found at the end of Section 3.1.2. To solve our issue with conflict clause validation, the verifier must output the full derivation tree of the conflict clause and not only the final propositional formula. This is again not a huge burden for the verifier as it has to explore the derivation tree to generate the conflict clause anyway. Based on this detailed information of conflicts the validator can easily validate the conflict clauses. To avoid an unnecessarily verbose witness, po relations can be omitted as well as nodes of tree corresponding to transitivity axiom application. Furthermore, an order derived by transitivity can also be wiped from the node as it can be easily deduced from the context. These are not really restrictions, as the validator can still easily check the conflict clause based on the reduced derivation tree. Figure 4.2b shows the reduced version of Figure 4.2a. To highlight the conciseness of these conflict clauses, a serialized version of the conflict is shown in Figure 4.2c which could actually be the output of a real tool. The following definition formalizes (reduced) derivation trees:

Definition 6 (Derivation Tree). Leaves of the tree are propositional logic formulae. An internal (non-leaf) node of a derivation tree represents a derivation step. An internal node consists of a derivation method and a happens-before order $e_1 \prec e_2$. The derivation method can be one of the following:

- po , if $e_1 \prec_{po} e_2$,
- rf , if $e_1 \prec_{rf} e_2$



(a) Full derivation tree



(b) Reduced version of Figure 4.2a

[RF(x#2, x#3), WS(RF(x#2, x#5), [], x#4)]

(c) Example of serialized (printed) version of the conflict by a potential real tool

Figure 4.2: Representations of a conflict clause for the program in Figure 3.1

- *tran*, if $e_1 \prec e_2$ is derived by Axiom 1,
- *ws*, if $e_1 \prec_{ws} e_2$ is derived by Axiom 2,
- *fr*, if $e_1 \prec_{fr} e_2$ is derived by Axiom 3,
- *tran**, if $e_1 \prec e_2$ is derived by multiple consequent applications of Axiom 1,
- *loop**, which is used instead of *tran** in the root node.

The root node must have $e_1 = e_2$, while $e_1 \neq e_2$ must hold in other internal nodes. *tran** and *loop** nodes must not have a *tran** child. ■

The happens-before order of *loop** and *tran** nodes can be omitted from the witness to reduce the size of the witness. I use the notation *method(order)* to represent an internal node textually: e.g., $po(e_1 \prec_{po} e_2)$ or $loop^*(\cdot)$ in case of an omitted order. For leaves, I write *leaf*(φ). I also introduce some predicates to make the algorithm more readable:

- $is_po(e_1, e_2)$ holds iff e_1 occurs before e_2 in the program.
- $is_rf(w, r)$ holds iff w and r are write and read events of the same variable.
- $po_ordered(nodes, from, to)$ is a bit more complicated. The predicate holds for a set of derivation tree nodes if there is an ordering of elements of the set where the second event of each node is po-related to the first event of the subsequent node in

Algorithm 2: Conflict clause derivation tree validation

```

1 Procedure valid(node)
2   children  $\leftarrow$  set of children nodes of node
3   switch node do
4     case  $po(e_1 \prec_{po} e_2)$  do
5       | return is_po( $e_1, e_2$ )
6     case  $rf(w \prec_{rf} r)$  do
7       | return is_rf( $w, r$ )
8     case  $tran(e_1 \prec e_2)$  do
9       | if children =  $\{m_1(e_1 \prec e'), m_2(e' \prec e_2)\}$  then
10        | | return valid( $m_1(e_1 \prec e')$ )  $\wedge$  valid( $m_2(e' \prec e_2)$ )
11     case  $ws(w' \prec_{ws} w)$  do
12       | if children =  $\{rf(w \prec_{rf} r), m(w' \prec r), leaf(grd_{w'})\}$  then
13        | | return is_rf( $w, r$ )  $\wedge$  valid( $m(w' \prec r)$ )
14     case  $fr(r \prec_{fr} w')$  do
15       | if children =  $\{rf(w \prec_{rf} r), m(w \prec w'), leaf(grd_{w'})\}$  then
16        | | return is_rf( $w, r$ )  $\wedge$  valid( $m(w \prec w')$ )
17     case  $tran^*(e_1 \prec e_2)$  do
18       | if po_ordered(children,  $e_1, e_2$ ) then
19        | | return  $\bigwedge_{c \in children} \text{valid}(c)$ 
20     case  $loop^*(\cdot)$  do
21       | if po_loop_ordered(children) then
22        | | return  $\bigwedge_{c \in children} \text{valid}(c)$ 
23   return false
  
```

the ordered list. Formally, we have:

$$\begin{aligned}
 po_ordered(nodes, from, to) \Leftrightarrow \\
 \exists list = [m_1(e_{1,1} \prec e_{1,2}), \dots, m_n(e_{n,1} \prec e_{n,2})] \text{ such that } nodes = \{c | c \in list\} \wedge \\
 is_po(from, e_{1,1}) \wedge is_po(e_{n,2}, to) \wedge is_po(e_{i,2}, e_{i+1,1}) \text{ for each } 1 \leq i < n
 \end{aligned}$$

- $po_loop_ordered(nodes) \Leftrightarrow \exists e \text{ event such that } po_ordered(nodes, e, e).$

Now, we arrive to Algorithm 2 that describes how conflict clauses or rather derivation trees can be validated. The input of the algorithm is a derivation tree: the *valid* procedure is called with the root node of the tree. The algorithm basically recursively checks each node in the tree if the derivation is valid at the node. Since the validator has to statically analyze the program structure anyway to build ψ_{ssa} , it is again not a real computational overhead to build a data structure for a quick querying of the *is_po* predicate during conflict clause validation. The following theorem states the correctness of Algorithm 2:

Theorem 3. Let r be the root of a derivation tree T , and c be the conjunction of formulae in leaf nodes of T . Then *valid*(r) implies that no valid program execution satisfies c . $\blacksquare$

Proof. The theorem can be proven by showing that checking the validity of a derivation step is correct. Then by induction on the structure of the derivation tree it follows that r represents an order derived by valid derivation steps. Since the derived order of r is a self-loop $e \prec e$, and the event graph of a valid execution cannot contain a self-loop, the conflict clause c of the derivation tree can only be satisfied by invalid executions.

The derivation step cases are mostly straightforward (such as if $is_po(e_1, e_2)$ holds then $e_1 \prec_{po} e_2$ also holds) so I skip most of the cases except $tran^*$ which is a bit complicated ($loop^*$ is analogous). The derivation of a node $tran^*(e_1 \prec e_2)$ is valid if all of its children are validly derived (which is guaranteed by the induction hypothesis) and if $po_ordered(children, e_1, e_2)$ holds for the children nodes. This predicate implies that there is an ordering of the children nodes $[m_1(e_{1,1} \prec e_{1,2}), \dots, m_n(e_{n,1} \prec e_{n,2})]$ where $is_po(e_{i,2}, e_{i+1,1})$ for each $1 \leq i < n$. Since $is_po(e_{i,2}, e_{i+1,1})$ implies $e_{i,2} \prec_{po} e_{i+1,1}$ and $e_{i,1} \prec e_{i,2}$ holds for each i as being the derived order of the i th item in the list of children, we obtain that $e_{1,1} \prec e_{1,2} \prec_{po} e_{2,1} \prec \dots \prec_{po} e_{n,1} \prec e_{n,2}$. From the definition of the $po_ordered$ predicate, we also know that $e_{1,1} = e_1$ and $e_{n,2} = e_2$ which means that applying the transitivity axiom multiple times on the obtained chain of orders will result in $e_1 \prec e_2$ in the end. Thus, the derivation of $e_1 \prec e_2$ is valid indeed. $\square$

4.4 Limitations of the Approach

The approach inherits its main limitation from bounded model checking, i.e., it can only be applied to loop-free programs or programs that can be safely unrolled (i.e., a bound for the number of iterations is statically known); otherwise, we can only speak about a bounded proof of safety. First, there are of course loop-free programs as demonstrated by the number of programs in the benchmark tests that does not contain loops or could be safely unrolled (see Chapter 5). Second, the ordering consistency-based algorithm could be used together with other techniques that eliminate loops: for example a technique could be used to generate loop invariants [12, 15], and then the loop can be replaced by a non-deterministic assignment of each variable written in the loop and an assumption of the loop invariant. This idea can be especially useful in a verification-validation workflow when the verifier might provide the necessary loop invariants. The difficult part is again concurrency, as it is more complicated to find loop invariants when global memory is accessed in the loop. Thus, the limitation remains, but as loop invariant techniques evolve, so widens the applicability of this algorithm.

Another aspect that I have not investigated in this work is the exchange of conflict clauses between verifiers and validators on a practical level. In my experiments, the verifier serialized the derivation trees of conflict clauses to a file which was then parsed by the validator. The tricky point could be event identification: what events are referred by the serialized derivation trees. Here in a simplified scenario, I built on the fact that the same deterministic parsing and pre-process steps were applied by the verifier and the validator when parsing the input program to an event graph. This way, without extra information, the same event graph was obtained with the same names and variable indices.

Naturally, this is not an assumption that we can make in a real scenario with different tools. However, the situation is very similar as to how loop invariants or other pieces of information are provided in witnesses in practice [6] where exact code location references are used. Each variable access in the code can be annotated in the witness (using exact code location references) by the associated event id(s)¹ used in the derivation trees. This of course increases the size of the witness but enables the tools to exchange this information. Furthermore, it could be required that sequential blocks of code naturally increment event ids in order to considerably reduce the necessary event id association entries in the witness. However, the exact format for event identification should be decided by the software verification community, so I refrain from proposing a detailed solution to the issue here.

¹The potential plural is motivated by unrolled loops: several events could be associated to the same variable access in the program code.

Chapter 5

Experimental Evaluation

In this chapter, I evaluate the efficiency of my theoretical contributions. First, I evaluate the automatic conflict detection approach in Section 5.1 followed by the evaluation of the presented validation approach Section 5.2. Finally, I discuss the threats to the validity of my experiments in Section 5.3.

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

5.1 Evaluating Automatic Conflict Detection

The goal of my experiments is to analyze the impact of strengthening the encoding formula by generated conflicts described in Section 3.2 on the overall verification performance. To have a broader image of how my algorithm helps existing techniques, I implemented two different methods for partial order-based verification. My first implementation resembles the approach of [45] where the initial encoding formula does not contain all scheduling constraints and the formula is gradually refined when the SMT solver finds a conflicting assignment. My second implementation follows the scheme described in Section 3.1 with a dedicated ordering consistency theory solver integrated into Z3 [21] via user propagators [17]. I refer to these versions as BASIC and PROP (short for propagator), respectively. I implemented¹ these proof-of-concept solutions in the THETA verification framework which could already parse C programs into control-flow automata [44].

5.1.1 Research Questions and Experiment Setup

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

RQ1.1 How many (simple) conflicts are found by our algorithm and how does this reduce the number of conflicts occurring during the decision procedure?

RQ1.2 What is the impact of further constraining the encoding formula on the size of the SMT solver model search space?

¹<https://github.com/csanadtelbisz/theta/tree/xcfa-oc-ws>

RQ1.3 What is the computational overhead of finding conflicts? In turn, how is the SMT solving time and the overall verification time reduced?

RQ1.4 How does the performance of our solution compare to the state-of-the-art?

I executed different configurations of THETA over the subset of input programs written in C from the concurrency safety reachability category benchmark suite² of SV-COMP [10] (715 tasks) that is parsable by THETA and supported by the partial-order based analysis: 544 tasks in total. The configurations are the two baseline implementations (BASIC and PROP), and the optimized versions (BASIC+OPT and PROP+OPT). For RQ1.4, I compare my solution to DEAGLE [33], a state-of-the-art bounded model checker built on CBMC based on the preventive propagation algorithm of [43], winner of the concurrency category of SV-COMP 2023 [9]. DEAGLE is shown to outperform other bounded model checkers, therefore I only compare to DEAGLE [43, 9]. I apply loop unwinding for programs with loops as a standard technique in bounded model checking. If the number of loop iterations can be statically decided, the loop is unwinded according to this number. Otherwise, loops are unrolled up to two iterations (in such cases, only a bounded proof of safety may be obtained). This loop unwinding tactic allows a fair comparison with DEAGLE applying the same method [33].

5.1.2 Experiment Results

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

RQ1.1 An average of 1125.7 conflicts are detected by our proposed method per task on the subset of commonly solved tasks of all configurations. Out of this, there are 514.5 conflicts with the structure of Figure 3.2a and 611.2 conflicts following Figure 3.2b or Figure 3.2c on average (note that we have the same number of Figure 3.2b and Figure 3.2c conflicts based on Algorithm 1). Strengthening the encoding formula with these conflict clauses found before starting the SMT solving procedure greatly reduces the number of conflict clauses produced by the decision procedures (see Table 5.1): the number of refinement clauses produced by the BASIC configurations is reduced by an average of 59.3% while the number of propagated conflict clauses by PROP is reduced by 41.1%. Note that more conflicts are generated the new way (simple conflicts discovered by my method and the conflicts generated by the decision procedures) than the original decision procedures produce, however this verbosity positively affects the performance of SMT solving.

²<https://gitlab.com/sosy-lab/benchmarking/sv-benchmarks>

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

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

RQ1.2 Various statistic data provided by Z3 can give us a rough estimate of the impact of my algorithm on the solver search space. The last columns of Table 5.1 show the average number of decision points, performed (unit and binary) propagation operations and necessary restarts. These numbers refer to inner Z3 operations (i.e., not the number of conflict clause propagations by PROP for instance). For the exact interpretation of these concepts, I refer the interested reader to a book on SMT [8] or the Z3 code base³. For the purposes of this evaluation, it is enough to see that the number of these important operations are substantially reduced indicating that the search space becomes much smaller when my algorithm is used. These values are reduced by 70-85% for the BASIC algorithm and by 20-45% for the PROP algorithm. Note that PROP working with a theory solver already achieves a huge reduction in the search space compared to BASIC (which is no surprise based on the paper of Sun et al. [43]); my proposed algorithm can still perform a significant further reduction for PROP.

RQ1.3 The runtime overhead of my conflict generation algorithm is not substantial, often negligible: it is around 0.6 seconds per task on average which means less than 5% of all CPU time. In turn, the time of the SMT solving procedure and thus the overall verification time is significantly reduced: using my optimization the SMT solver is 64.9% faster in case of the BASIC algorithm, and 51.4% faster in case of PROP. The overall CPU time of the verification is reduced by a significant 41.7% for BASIC and 30.7% for PROP on average. The CPU times are also demonstrated in the scatter plots of Figure 5.1 where the majority of points are under the diagonal indicating the performance improvement of my proposed technique.

RQ1.4 Firstly, I note that I do not intend to compete with DEAGLE in terms of absolute performance (CPU time) as my proof-of-concept implementation runs on the JVM while DEAGLE is built on native CBMC code optimized over decades. Having said that, my evaluation shows that my solution with the best configuration (PROP algorithm working with similar principles as DEAGLE extended with my optimization) could verify as much tasks as DEAGLE: 520 tasks. Levelling with a verifier with a highly optimized codebase clearly shows the potential of my contributions. Furthermore, my solution could produce bounded proofs for two more tasks where the tasks are in fact unsafe beyond the loop unroll bound. DEAGLE is shown to outperform other state-of-the-art bounded model checkers and no other published results surpass DEAGLE [43, 9, 10] which implies that my solution also outperforms other existing tools. It is also important to note that DEAGLE is limited to verification under *weak sequential consistency* (see the supplement material) while my solution is suitable for the stronger and more useful *sequential consistency* memory model.

³<https://github.com/Z3Prover/z3>

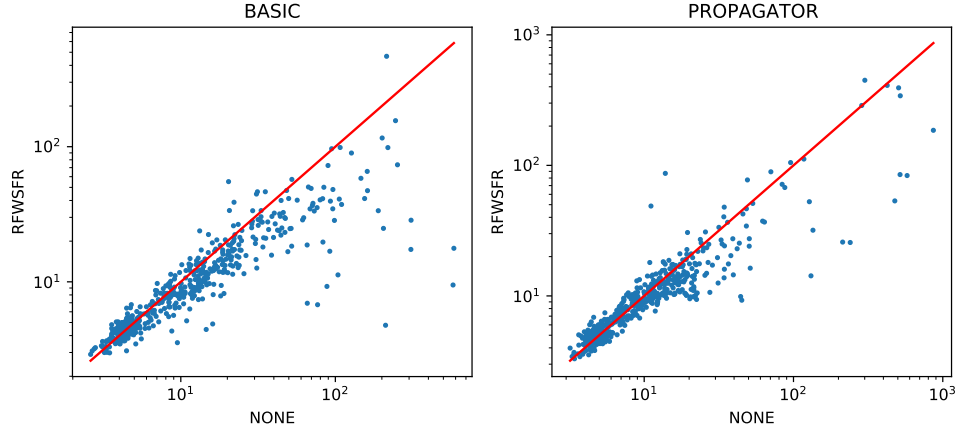


Figure 5.1: CPU time improvement of the optimized verification

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

5.2 Evaluating Validation with Conflict Clauses

In the second part of my experiment, I evaluate the performance of the ordering consistency theory-based validator for concurrent programs presented in Chapter 4. The verifier is the same implementation as for the previous experiments. Furthermore, I also implemented a validator in the THETA verification tool [44] based on the explained principles in Chapter 4. I use the two implemented versions of the verifier (BASIC and PROP). The advantage of the PROP approach is that it can detect inconsistency early, thus avoiding a lot of backtracking in the SMT solver, so it scales better for the verification as revealed in the evaluation of the previous section. On the other hand, the BASIC approach can be used with any arbitrary SMT solver. Furthermore, when we expect no inconsistency to be found (i.e., in most cases in the validator), this approach does not waste time on detecting inconsistency on partial models. In a permissive mode (see Section 4.2 for details), the validator can use the same implementations as the verifier for further checking the safety of the program.

5.2.1 Research Questions and Experimental Setup

To evaluate the presented algorithm, I aim to answer the following research questions concerning important aspects of my novel algorithm:

RQ2.1 Can the validator validate the witness and thus confirm the correctness of the program?

RQ2.2 How much performance improvement is achieved by the validator using the additional information from the witnesses compared to the verifier?

RQ2.3 What is the computational overhead of validating conflict clauses?

In my experiments, I executed different configurations of THETA over a set of safe input programs from the same benchmark dataset as in the earlier parts of the evaluation. I executed the two verifier configurations (BASIC and PROP) to produce witnesses and to

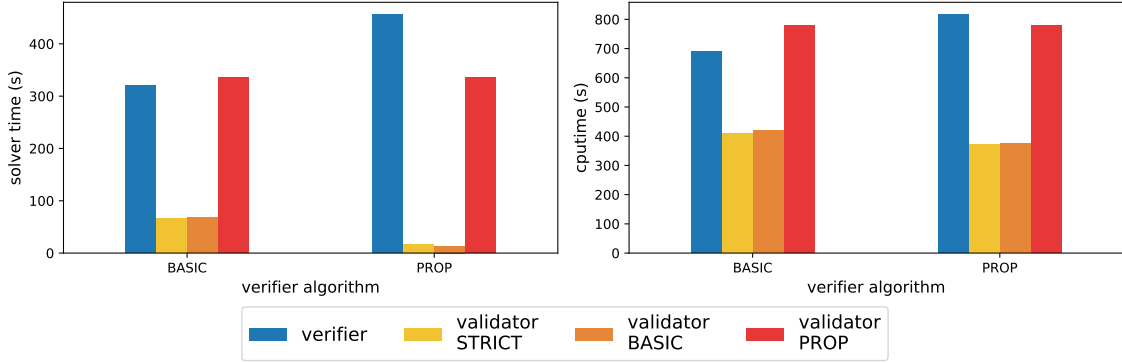


Figure 5.2: Comparison of verifier and validator performances

Metric	Verifier algorithm: BASIC				Verifier algorithm: PROP			
	Verifier	Validator STRICT	Validator BASIC	Validator PROP	Verifier	Validator STRICT	Validator BASIC	Validator PROP
Verified/ validated	87	87	87	87	91	90	90	90
Generated conflicts	18236	-	0	10228	28024	-	0	4753
SMT solver time (s)	320	66	68	336	457	16	14	255
Validation time (s)	-	2.86	2.93	3.02	-	3.51	3.28	3.68
CPU time (s)	689	408	420	779	818	373	375	711

Table 5.2: Different metrics of the evaluation of the validation approach

have a baseline performance. I tested three validator methods: two permissive validators that use the two verifier decision procedures and a strict version (STRICT) that simply inputs the validation formula to an SMT solver without an ordering theory solver. So I executed 6 validator configurations: each validation method for validating the witnesses of both verifier methods.

5.2.2 Experiment Results

In the concurrency safety benchmark suite, there are 347 safe tasks for the error reachability property. Out of this, my proof-of-concept implementation supports 137 programs (the rest contains unsupported language elements) of which 40 programs contain loops that cannot be safely unrolled. So the experiment is technically performed on the remaining 97 tasks. No configuration provided wrong results. Table 5.2 shows results of the benchmark tests. For a fair comparison, data has been aggregated for each metric (except for the number of solved/validated tasks) for the 87 tasks that was verified and validated by all configuration. A dash indicates that the respective metric is not applicable (or zero by definition) for that configuration. Let us have the answers to the research questions:

RQ2.1 First, all generated conflict clauses could be validated by the validator configurations, so derivation trees indeed provide enough information to reestablish the scheduling inconsistencies. Also, except for one case, all validation could confirm the verification result based on the witness. In that case, there were a lot of conflict

clauses and fewer clauses were serialized than propagated (probably due to some buffer limit) which caused the STRICT validator to reject the witness and the other two validator configurations to timeout.

RQ2.2 We can observe based on Table 5.2 and Figure 5.2 that the validation process took significantly less time when not the PROP configuration is used by the validator (which is expected to be less efficient in this setting than other configurations as observed previously; this is due to trying to find conflicts early as confirmed by the number of extra generated conflict clauses which is not a winning strategy during validation). Also not surprisingly, STRICT is slightly better than BASIC as BASIC also builds the initial event graph even though no further conflict clauses were generated. Thus, the SMT solver time was reduced to 20.6% and 3.1% of the verification solver time for BASIC and PROP verification procedures respectively. The greater reduction for PROP is the consequence of the number of generated conflicts being much more in this case as seen in Table 5.2 as well. The overall CPU time is not reduced to this extent as parsing the program and building the initial event graph is also a considerable part of CPU time: still it is reduced to 59.2% and 45.6% for BASIC and PROP respectively. This suggests that the presented approach can achieve greater reduction in complex programs where parsing takes less significant part of the verification.

RQ2.3 Conflict clause validation times are also visible in Table 5.2. Though these times are not completely negligible, it is clearly not a significant part of validation. Validation time correlates with the number of generated conflict clauses: thus we have around 3 seconds of aggregated validation for clauses generated by the BASIC, and around 3.5 seconds for PROP which are less than 1% of all CPU time of validation.

It is also interesting to investigate whether the witnesses are human-readable. Of course, actual readability heavily depends on the applied format how derivation trees are serialized. So I only make a few remarks on this point from a higher level of abstraction. The number of generated conflict clauses is usually too much for human processing in the general case, the average being 210 and 322 clauses per program for BASIC and PROP respectively. On the other hand, the number of serialized nodes of a single (reduced) derivation tree is typically between 1-5 which can be processed by a human reader. Furthermore, typically having small conflicts confirms the validity of my bounded conflict generation method described in Section 3.2.2. Thus, single conflicts in the witnesses may provide meaningful and processable intuition to humans of the correctness of the program.

In conclusion, the evaluation confirms that the proposed way to store some information about concurrent behavior in a verification witness can be efficiently used during validation. It is also revealed that the user propagator-based method is not useful during validation, however, simple approaches without user propagators are suitable for the task.

5.3 Threats to Validity

Limitations of the presented validation approach have been discussed in Section 4.4. Otherwise, the validity of my experiments may be influenced by the following factors.

Internal validity. Consistency and accuracy of the experiments were insured by using the BenchExec framework [14]. I executed my experiments on virtual machines in a cloud computing platform. Therefore, external factors such as loads on other virtual machines

of the host environment and shared resources (such as disks) may have slightly influenced the results. Slow IO operations and differences between printing out and reading-parsing of derivations trees may have caused a minor difference between verifier and validator configurations.

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

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

Furthermore, I would like to note that in my comparative evaluation I used the binary of DEAGLE submitted to SV-COMP 2024 [10]. I also tried to reproduce the results by building the binary of DEAGLE from its source code⁴ but I failed to get the same results as several incorrect verdicts (both false positives and negatives) were produced by this version. However, if DEAGLE is indeed only capable of the slightly worse results, that only affects the placement of my solution among state-of-the-art techniques positively. Thus, this is only a positive threat to the validity of my evaluation.

⁴<https://github.com/thufv/Deagle>

Chapter 6

Conclusion

In my work, I advance the state-of-the-art in partial order-based certified verification of concurrent programs in several aspects. I presented a revised verification algorithm based on existing ideas. I showed different ways to make use of scheduling inconsistencies.

First, I presented a novel optimization based on potential scheduling inconsistencies. I reason with an over-approximation of happens-before relations on the level of the whole program which allows to collect properties characterizing all possible program executions. My algorithm strengthens the program encoding formula by finding scenarios conflicting with concurrent behavior and converting these into first-order clauses. Due to limiting the algorithm to discover conflicting situations of bounded size, it does not pose a serious computational overhead on building the encoding formula. The presented approach can be conveniently integrated into existing methods to reduce the search space of SMT solvers and accelerate the decision procedure. Thus, the presented algorithm effectively combines characteristics of eager and lazy approaches to verification with partial orders. Evaluation reveals significant boost in verification performance and shows that my solution achieves the best results among state-of-the-art verification tools. Therefore, my proposed algorithm can serve as a useful enhancement to other state-of-the-art program verifiers as well reasoning with partial orders.

Second, I argued that these scheduling inconsistencies can be useful information in a verification witness for concurrent programs, and proposed a way to validate this information.

In the future, I plan to thoroughly analyze the trade-off between allowing the algorithm to discover more complicated conflicts and the computational overhead of the discovering algorithm plus the increased size of the encoding formula. I also plan to extend my approach to general weak memory models (weaker than weak sequential consistency). As for the validation method, I plan to present my contributions to the software verification community and intend to incorporate the proposed concurrency-related witness information into standardized witness formats.

Acknowledgements

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

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

Bibliography

- [1] Parosh Aziz Abdulla, Stavros Aronis, Bengt Jonsson, and Konstantinos Sagonas. Optimal dynamic partial order reduction. pages 373–384. ACM, 2014. DOI: 10.1145/2535838.2535845.
- [2] Jade Alglave, Luc Maranget, Susmit Sarkar, and Peter Sewell. Fences in weak memory models (extended version). *Formal Methods Syst. Des.*, 40(2):170–205, 2012. DOI: 10.1007/S10703-011-0135-Z.
- [3] Jade Alglave, Daniel Kroening, and Michael Tautschnig. Partial orders for efficient bounded model checking of concurrent software. volume 8044 of *Lecture Notes in Computer Science*, pages 141–157. Springer, 2013. DOI: 10.1007/978-3-642-39799-8_9.
- [4] Jade Alglave, Daniel Kroening, Vincent Nimal, and Daniel Poetzl. Don’t sit on the fence - A static analysis approach to automatic fence insertion. volume 8559 of *Lecture Notes in Computer Science*, pages 508–524. Springer, 2014. DOI: 10.1007/978-3-319-08867-9_33.
- [5] Mohamed Faouzi Atig, Ahmed Bouajjani, Sebastian Burckhardt, and Madanlal Musuvathi. On the verification problem for weak memory models. pages 7–18. ACM, 2010. DOI: 10.1145/1706299.1706303.
- [6] Paulína Ayaziová, Dirk Beyer, Marian Lingsch-Rosenfeld, Martin Spiessl, and Jan Strejček. Software verification witnesses 2.0. *Proc. SPIN. Springer*, 2024.
- [7] Levente Bajczi, Zsófia Ádám, and Zoltán Micskei. Concurrentwitness2test: Test-harnessing the power of concurrency (competition contribution). volume 14572 of *Lecture Notes in Computer Science*, pages 330–334. Springer, 2024. DOI: 10.1007/978-3-031-57256-2_16.
- [8] Clark W. Barrett, Roberto Sebastiani, Sanjit A. Seshia, and Cesare Tinelli. Satisfiability modulo theories. volume 336 of *Frontiers in Artificial Intelligence and Applications*, pages 1267–1329. IOS Press, 2021. DOI: 10.3233/FAIA201017.
- [9] Dirk Beyer. Competition on software verification and witness validation: SV-COMP 2023. volume 13994 of *Lecture Notes in Computer Science*, pages 495–522. Springer, 2023. DOI: 10.1007/978-3-031-30820-8_29.
- [10] Dirk Beyer. State of the art in software verification and witness validation: SV-COMP 2024. volume 14572 of *Lecture Notes in Computer Science*, pages 299–329. Springer, 2024. DOI: 10.1007/978-3-031-57256-2_15.
- [11] Dirk Beyer, Matthias Dangel, Daniel Dietsch, Matthias Heizmann, and Andreas Stahlbauer. Witness validation and stepwise testification across software verifiers. pages 721–733. ACM, 2015. DOI: 10.1145/2786805.2786867.

- [12] Dirk Beyer, Matthias Dangl, and Philipp Wendler. Boosting k-induction with continuously-refined invariants. volume 9206 of *Lecture Notes in Computer Science*, pages 622–640. Springer, 2015. DOI: 10.1007/978-3-319-21690-4_42.
- [13] Dirk Beyer, Matthias Dangl, Daniel Dietsch, and Matthias Heizmann. Correctness witnesses: exchanging verification results between verifiers. pages 326–337. ACM, 2016. DOI: 10.1145/2950290.2950351.
- [14] 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.
- [15] Dirk Beyer, Matthias Dangl, Daniel Dietsch, Matthias Heizmann, Thomas Lemberger, and Michael Tautschnig. Verification witnesses. *ACM Trans. Softw. Eng. Methodol.*, 31(4):57:1–57:69, 2022. DOI: 10.1145/3477579.
- [16] Per Bjesse. What is Formal Verification? *SIGDA Newsl.*, 35(24):1–es, dec 2005. ISSN 0163-5743. DOI: 10.1145/1113792.1113794.
- [17] Nikolaj S. Bjørner, Clemens Eisenhofer, and Laura Kovács. Satisfiability modulo custom theories in Z3. volume 13881 of *Lecture Notes in Computer Science*, pages 91–105. Springer, 2023. DOI: 10.1007/978-3-031-24950-1_5.
- [18] Ron Cytron, Jeanne Ferrante, Barry K. Rosen, Mark N. Wegman, and F. Kenneth Zadeck. An efficient method of computing static single assignment form. pages 25–35. ACM Press, 1989. DOI: 10.1145/75277.75280.
- [19] Hernán Ponce de León, Florian Furbach, Keijo Heljanko, and Roland Meyer. Portability analysis for weak memory models. PORTHOS: one tool for all models. volume 10422 of *Lecture Notes in Computer Science*, pages 299–320. Springer, 2017. DOI: 10.1007/978-3-319-66706-5_15.
- [20] Hernán Ponce de León, Florian Furbach, Keijo Heljanko, and Roland Meyer. Dartagnan: Bounded model checking for weak memory models (competition contribution). volume 12079 of *Lecture Notes in Computer Science*, pages 378–382. Springer, 2020. DOI: 10.1007/978-3-030-45237-7_24.
- [21] Leonardo Mendonça de Moura and Nikolaj S. Bjørner. Z3: an efficient SMT solver. volume 4963 of *Lecture Notes in Computer Science*, pages 337–340. Springer, 2008. DOI: 10.1007/978-3-540-78800-3_24.
- [22] Leonardo Mendonça de Moura and Nikolaj S. Bjørner. Satisfiability modulo theories: introduction and applications. *Commun. ACM*, 54(9):69–77, 2011. DOI: 10.1145/1995376.1995394.
- [23] Matthew B. Dwyer and Lori A. Clarke. Data Flow Analysis for Verifying Properties of Concurrent Programs. pages 62–75. ACM, 1994. DOI: 10.1145/193173.195295.
- [24] Harald Ganzinger, George Hagen, Robert Nieuwenhuis, Albert Oliveras, and Cesare Tinelli. DPLL(T): fast decision procedures. volume 3114 of *Lecture Notes in Computer Science*, pages 175–188. Springer, 2004. DOI: 10.1007/978-3-540-27813-9_14.
- [25] Natalia Gavrilenko, Hernán Ponce de León, Florian Furbach, Keijo Heljanko, and Roland Meyer. BMC for weak memory models: Relation analysis for compact SMT encodings. volume 11561 of *Lecture Notes in Computer Science*, pages 355–365. Springer, 2019. DOI: 10.1007/978-3-030-25540-4_19.

- [26] Elahieh Ghassabani, Andrew Gacek, and Michael W. Whalen. Efficient generation of inductive validity cores for safety properties. pages 314–325. ACM, 2016. DOI: 10.1145/2950290.2950346.
- [27] Phillip B. Gibbons and Ephraim Korach. Testing shared memories. *SIAM J. Comput.*, 26(4):1208–1244, 1997. DOI: 10.1137/S0097539794279614.
- [28] Patrice Godefroid. Model Checking for Programming Languages using Verisoft. pages 174–186. ACM Press, 1997. DOI: 10.1145/263699.263717.
- [29] Orna Grumberg, Edmund M. Clarke, and Doron A. Peled. Model checking. 1999.
- [30] Thomas Haas, Roland Meyer, and Hernán Ponce de León. CAAT: consistency as a theory. *Proc. ACM Program. Lang.*, 6(OOPSLA2):114–144, 2022. DOI: 10.1145/3563292.
- [31] Thomas Haas, René Pascasl Maseli, Roland Meyer, and Hernán Ponce de León. Static analysis of memory models for SMT encodings. *Proc. ACM Program. Lang.*, 7(OOPSLA2):1618–1647, 2023. DOI: 10.1145/3622855.
- [32] Fei He, Zhihang Sun, and Hongyu Fan. Satisfiability modulo ordering consistency theory for multi-threaded program verification. pages 1264–1279. ACM, 2021. DOI: 10.1145/3453483.3454108.
- [33] Fei He, Zhihang Sun, and Hongyu Fan. Deagle: An smt-based verifier for multi-threaded programs (competition contribution). volume 13244 of *Lecture Notes in Computer Science*, pages 424–428. Springer, 2022. DOI: 10.1007/978-3-030-99527-0_25.
- [34] Thomas A. Henzinger, Ranjit Jhala, Rupak Majumdar, George C. Necula, Grégoire Sutre, and Westley Weimer. Temporal-safety proofs for systems code. volume 2404 of *Lecture Notes in Computer Science*, pages 526–538. Springer, 2002. DOI: 10.1007/3-540-45657-0_45.
- [35] Hyoung Seok Hong, Insup Lee, Oleg Sokolsky, and Sung Deok Cha. Automatic test generation from statecharts using model checking. In *Workshop on Formal Approaches to Testing of Software (FATES)*, pages 15–30, 2001.
- [36] Alex Horn and Jade Alglave. Concurrent kleene algebra of partial strings. *CoRR*, abs/1407.0385, 2014. URL <http://arxiv.org/abs/1407.0385>.
- [37] Alex Horn and Daniel Kroening. On partial order semantics for sat/smt-based symbolic encodings of weak memory concurrency. volume 9039 of *Lecture Notes in Computer Science*, pages 19–34. Springer, 2015. DOI: 10.1007/978-3-319-19195-9_2.
- [38] Jeff Huang. Stateless model checking concurrent programs with maximal causality reduction. pages 165–174. ACM, 2015. DOI: 10.1145/2737924.2737975.
- [39] Alexei Iliasov. Generation of certifiably correct programs from formal models. pages 43–48. IEEE Computer Society, 2011. DOI: 10.1109/WOSOCER.2011.14.
- [40] Doron A. Peled. Ten Years of Partial Order Reduction. volume 1427 of *Lecture Notes in Computer Science*, pages 17–28. Springer, 1998. DOI: 10.1007/BFb0028727.
- [41] Ishai Rabinovitz and Orna Grumberg. Bounded model checking of concurrent programs. volume 3576 of *Lecture Notes in Computer Science*, pages 82–97. Springer, 2005. DOI: 10.1007/11513988_9.

- [42] Herbert Rocha, Raimundo S. Barreto, Lucas C. Cordeiro, and Arilo Dias Neto. Understanding programming bugs in ANSI-C software using bounded model checking counter-examples. volume 7321 of *Lecture Notes in Computer Science*, pages 128–142. Springer, 2012. DOI: 10.1007/978-3-642-30729-4_10.
- [43] Zhihang Sun, Hongyu Fan, and Fei He. Consistency-preserving propagation for SMT solving of concurrent program verification. *Proc. ACM Program. Lang.*, 6(OOPSLA2): 929–956, 2022. DOI: 10.1145/3563321.
- [44] Tamás Tóth, Ákos Hajdu, András Vörös, Zoltán Micskei, and István Majzik. THETA: a Framework for Abstraction Refinement-Based Model Checking. pages 176–179, 2017. ISBN 978-0-9835678-7-5. DOI: 10.23919/FMCAD.2017.8102257.
- [45] Liangze Yin, Wei Dong, Wanwei Liu, and Ji Wang. On scheduling constraint abstraction for multi-threaded program verification. *IEEE Trans. Software Eng.*, 46(5): 549–565, 2020. DOI: 10.1109/TSE.2018.2864122.
- [46] Rachid Zennou, Mohamed Faouzi Atig, Ranadeep Biswas, Ahmed Bouajjani, Constantin Enea, and Mohammed Erradi. Boosting sequential consistency checking using saturation. volume 12302 of *Lecture Notes in Computer Science*, pages 360–376. Springer, 2020. DOI: 10.1007/978-3-030-59152-6_20.