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

Application of Counterexample-Guided Abstraction Refinement on Concurrent Programs

Scientific Students' Association Report

Author:

Levente Bajczi

Advisor:

Dr.Vince Molnár

2021

Contents

Kivonat

A formális szoftververifikáció aktívan kutatott alterülete a többszálú programok hatékony kezelésének problémája. A többmagos processzorok biztonságkritikus rendszerekben való megjelenésével egyre nagyobb szükség van olyan robusztus, a biztonságosság bizonyítására alkalmas megoldásokra, amelyek képesek figyelembe venni a több mag jelentette megnövekedett komplexitást. Ezen új kihívások legfőbb oka a megosztott adatokhoz történő párhuzamos hozzáférés, ami újfajta hibalehetőségek (pl. memória- vagy cache-inkonzisztencia) révén előre nem látható, potenciálisan katasztrofális hibához vezető problémákat okozhatnak a szoftverben és a hardverben.

A szoftververifikáció terén eddig publikált kutatások többnyire a probléma egy részhalmazát célozták, leszűkítve arra az esetre, amikor minden adathozzáférés szigorúan szekvenciális. Ezen megoldások nem alkalmasak a gyenge rendezésű hardver-szoftver rendszerek analízisére, melyeket azonban széles körben alkalmaznak a hatékonyság növelése érdekében. A tudományos világ csak az elmúlt néhány évben kezdett el új, általánosabban alkalmazható, memóriamodell-alapú megoldások kifejlesztésével foglalkozni. A probléma komplexitása miatt ezek a megoldások általában a korlátos modellellenőrzés kiterjesztései, melyek csak a program első k lépéséről képesek érvelni, és így nem tekinthetőek általános megoldásnak.

Jelen dolgozatban bemutatok egy új, memóriamodellt figyelembe vevő, ellenpélda-alapú absztrakciófinomítás (CEGAR) technikát alkalmazó nem-korlátos modellellenőrző algoritmust gyengén rendezett párhuzamos programok kezelésére. Az algoritmus hatékonyságát a terület meghatározó szekvenciális és gyenge rendezésű memóriát feltételező verifikációs eszközeivel összevetve értékelem ki. Bemutatok továbbá egy összefoglaló képet a szekvenciális memóriamodellű, párhuzamos programokat kezelő megoldásokról, melyek a CEGAR technikát használják. Ezeket a megközelítéseket összehasonlítom a komplexitásuk, hatékonyságuk és használhatóságuk szerint is.

Az Innovációs és Technológiai Minisztérium ÚNKP-21-2 kódszámú Új Nemzeti Kiválóság Programjának a Nemzeti Kutatási, Fejlesztési és Innovációs Alapból finanszírozott szakmai támogatásával készült.

A 2019-1.3.1-KK-2019-00004 számú projekt a Nemzeti Kutatási, Fejlesztési és Innovációs Hivatal támogatásával készült, a 2019-1.3.1-KK költségvetési terv alapján finanszírozva.

Abstract

Effectively handling multithreaded programs is an active field of research in the context of formal software verification. As the world moves to multi-core processors in safety-critical settings, a robust solution is necessary to prove safety considering the increased complexity. The main source of the new challenges is the concurrent manipulation of shared data, where new kinds of problems (such as memory- and cache inconsistency) might cause unforeseen faults in software and hardware, leading to potentially catastrophic failures.

Previously published research in software verification has mostly targeted a subset of the problem, narrowing it down to the case when access to shared data is strictly sequential. These approaches are not suitable for the analysis of weakly ordered hardware-software systems, which are widely used and represent a generally more efficient solution to parallelization. In recent years, there has been work on providing broader, memory model-aware approaches. Due to the complexity of the problem, these have generally been extensions to bounded analysis techniques, which can only reason about the first k steps of the program, lacking generality.

In this report, I introduce a novel, memory model-aware Counterexample-Guided Abstraction Refinement (CEGAR) based algorithm for handling the unbounded analysis of weakly-ordered concurrent programs. I evaluate the efficiency of this algorithm by comparing it to the state-of-the-art verification tools for sequentially- and weakly-ordered programs. I also present an overview of existing approaches to handling sequentially-ordered concurrent programs using the Counterexample-Guided Abstraction Refinement (CEGAR) technique. I compare and contrast these approaches by their complexity, efficiency and usability as general-purpose software verification tools.

Supported by the ÚNKP-21-2 New National Excellence Program of the Ministry for Innovation and Technology from the Source of the National Research, Development and Innovation Fund.

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

Chapter 1

Introduction

Embedded software has reached a point in many safety-critical domains, where the performance of a single core is not enough. This is either due to the inherent limitations of the silicon [39], or pricing considerations. For example, in one of the preferred (and safety-certified) microprocessor families of the automotive domain[1], if a clock frequency of 200MHz is not enough, there are only multi-core options available with a higher setting. This has led many suppliers to use a multi-core microprocessor with only a single core running, to take advantage of the performance gain. However, if the same software could run on multiple cores, a less powerful (and therefore cheaper) multi-core solution could be utilized, which could save enough money for the supplier to justify the heightened complexity of the software. Furthermore, many suppliers see that this change is inevitable – sooner or later, the number and quality of features expected from a single electronic control unit (ECU) will overwhelm any single-core processor, at which point either a second unit has to be used, effectively doubling the costs; or an adaptive solution is necessary that utilizes features of a multi-core microprocessor.

However, there are many pitfalls of porting legacy (single-core) code to multi-core architectures, one of which is the hindrance of testing. In a single-core safety-critical system, testing (if done thoroughly) can be considered a somewhat definitive proof of safety – the supplier of a part has to show that any reasonable scenario is taken care of, and no dangerous harm should (with great confidence) befall the future user of the product. However, in the multi-core case, simple testing is inherently non-deterministic, as the overlapping of execution in the software's threads can cause different outcomes, solely based on the timing of the processor cores.

A further source of complexity stems from the complicated memory models of multi-core processors. Almost all multi-core processors handle memory accesses in a relaxed way, i.e. the order of operations in the source code does not necessarily order the actual memory accesses during execution. For example, the microprocessor family mentioned above uses a *Total-Store Ordering* (TSO) memory model [1], which means that only stores and other dependent memory accesses are ordered by their location in the program's source, any other type of access can freely overtake each other.

This phenomenon also means that a software developer has to be aware of these architectural details, otherwise, even a sane implementation of a multi-threaded algorithm might produce unwanted results. Consider the following two-threaded program (x and y are global variables):

[1] https://www.infineon.com/cms/en/product/microcontroller/32-bit-tricore-microcontroller/32-bit-tricore-aurix-tc3xx/

```
x = 0;                int i = y;
y = 0;                int j = x;
x = 1;                if ( i > j )
y = 1;                    ERROR: exit(-1);
```

Even though it seems evident that i will always be smaller than or equal to j because y is always increased *after* x in the writer thread; the ERROR label is sometimes reachable when we run the above program on a TSO architecture. This can be justified by the memory model: even though the writes to global variables order themselves based on the source code (i.e. the actual values in memory will always be dictated by the line-by-line instructions in the writer thread), the read from y might block and give way to the second read before it finishes. This can lead to the outcome designated as erroneous – and similar (but harder to detect) problems can cause silent problems in the product, which are not recognized during testing, but when millions of users execute the same piece of code, *some* will experience a program failure that might lead to physical or monetary harm.

On the other hand, this heightened complexity also means that formal methods such as model checking is even less effective at checking the safety of multi-threaded software. However, as testing is not powerful enough to prove the safety of the product, we have to use some form of formal software verification to make sure that the system we developed is safe enough – and therefore, a practical and performant solution is required to evaluate the multi-threaded algorithms in safety-critical systems.

To further justify the importance of this field, taking a look at the results of the 21st Software Verification Competition (SV-COMP[2]) [11], even though many of the participating tools support a form of concurrency, participants use a wide range of techniques to handle these programs. Out of the tools that performed above the positive average of 600 points in this category, all four use a different approach (such as a mix of partial order reduction, sequentialization, predicate abstraction and bounded model checking) – which indicates that there is no single best technique that is so overwhelmingly performant that others cannot compete.

In this report, I present techniques utilizing the Counterexample-Guided Abstraction-Refinement (CEGAR [21]) approach for handling the problem of concurrency in formal program verification. *As the main contribution, I have developed a technique that handles the abstraction-refinement of weakly-ordered concurrent programs using declarative semantics.* Furthermore, I implement said algorithm along with existing methods of handling sequential programs in a CEGAR loop inside the open-source Theta model checking framework[3] [40, 30] for a just evaluation of these techniques.

This report is structured as follows: in Chapter 2, I introduce the necessary concepts and the definitions the rest of the work uses. In Chapter 3, I present the tools and algorithms whose purpose overlaps with that of this report. In Chapter 4, I introduce the main contribution of my work, the algorithm capable of verifying unbounded, weakly-ordered concurrent programs. In Chapter 5, I elaborate on the implementation details of the algorithms mentioned in the scope of this report. Finally, in Chapter 6, I present the performance comparison of the algorithms above.

[2] https://sv-comp.sosy-lab.org/
[3] https://github.com/ftsrg/theta/

Chapter 2

Background

This report builds upon the theories and findings of many fields of computer science, including embedded programming, formal software verification, memory modeling, and concurrent software design. Some of these fields view the same topics slightly differently, e.g. software verification presumes a formal, mathematical model for the input program, while embedded programmers usually use the much lower abstraction level of source code to reason about properties of the software. This necessitates establishing the basis of the presented work to prevent misunderstanding among experts in these fields. This chapter introduces such concepts and defines their interpretation as used in the context of this work.

2.1 Safety-Critical Systems

If a hardware-software system was designed for a small selection of well-defined tasks, it is generally referred to as an *embedded system*. Such systems are not adept for general-purpose use, as their in- and outputs are often limited, and their software is seldom modifiable with the rare exception of program upgradability. *Embedded systems* can fulfil many kinds of tasks, ranging from operating the electrical windows on a car to performing complicated protocols for mid-air collision avoidance in airplanes[1], or providing a safety shutoff system for a nuclear plant.

Failure of an embedded system might be a minor nuisance or a serious safety problem, depending on the context of the application. If an electrical window fails on a car, the worst that can happen is some discomfort until the faulty unit is repaired or replaced – but failure of the TCAS might result in the collision of two airplanes where hundreds of lives are at risk. Any system that is designed to perform tasks where malfunction could lead to harm (physical or monetary) is classified as a *safety-critical system*.

Depending on the level of tolerable risk, a safety-critical system can be classified e.g. according to *Safety Integrity Levels* (SIL) [32]. Several qualitative and quantitative measures are in place in such standards to mitigate dangerous failures, such as a controlled development workflow, thorough quality assurance and safety evaluation. For software components, the most widely used technique to assess safety is testing, i.e. running the program with defined sets of inputs and analyzing the outputs. This is not a definitive proof, as for untested inputs we cannot evaluate the behavior, but if the testing methodology is thorough enough, we can qualify the software as *probably safe* for the desired

[1] Such as the Traffic Alert and Collision Avoidance System (TCAS)

safety integrity level. To aid testing, formal methods such as model checking [22] (also see Section 2.2) and formal test generation [18] can be used, which are often too complex to be used on their own, but can support conventional testing methods.

With recent years' advancements, even safety-critical systems arrived at the point where scaling up in performance is next to impossible if only a single core is utilized [39]. The next logical step is to introduce multi-processor chips that can use smarter workload management to overcome the need for computing power. However, with multi-processor architectures and multi-threaded programs, the complexity of embedded systems surpasses the verification power of conventional testing, mainly due to the inherent nondeterminism of multi-threaded programs. When a program is strictly run on a single-core processor, it is relatively easy to guarantee that for a single set of inputs, the output of the program will always be the same. Therefore, it is enough to test each input set once, and assess the execution's results. With multi-threaded programs, an otherwise deterministic program can still produce different results based on timing differences among the processors, which cause different sections of the program to overlap in execution. Hence, testing is even less effective at proving safety, and a more formal method is often required.

2.2 Formal Software Verification

Formal software verification is a way to mathematically prove or disprove certain properties of an input program. Such properties might include *memory safety* (detecting use-after-free and other memory allocation problems), *reachability* (detecting if an unsafe state is reachable) or *termination* (detecting if the program will terminate in all its executions). In the context of this work, safety properties are always assumed to be *reachability* related, with a single unsafe state in the program.

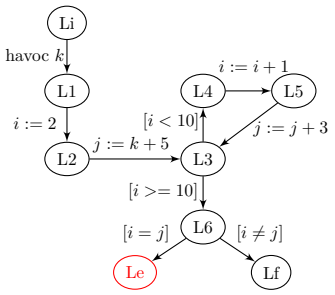
Formal software verification often employs *model checking* [22], a technique that enumerates states of the input program and reasons about the properties of the states. For *reachability*-type queries, it is necessary to know whether the state marked as *unsafe* is reachable from the initial state(s) within a finite number of steps – if such a path exists, the program is deemed *unsafe* and safety cannot be guaranteed. In practice, generating all states of an input model is often infeasible, as even a single 32-bit variable will create 2^{32} different states according to its value. This phenomenon is called *state space explosion* [23], and counteracting it is required for any practically useful model checking algorithm.

A theoretical problem that verification tools have to face is the inherent *undecidability* of the model checking problem. Consider an arbitrary input program and an unsafe state at its exit point. To prove the (un)reachability of said state, the program's termination property has to be decided – which is proven to be undecidable [42]. This means, that any model checking algorithm will either be *incomplete*, i.e. some inputs will result in an *UNKNOWN* classification, or will produce *false* results in the form of *false alarms* and *missed bugs*.

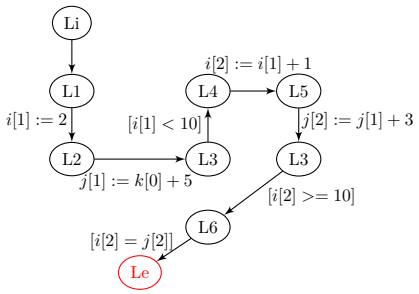
A further problem of such algorithms is bridging the gap between the different abstraction levels in the verification workflow. Embedded programs are usually written in C or a similar high-level language, where concepts such as variable scopes, procedures and pointers make the lives of programmers easier, abstracting away the single instructions that will be generated by the compiler. However, the rich toolset of high-level languages greatly hinder the reasoning power of any formal method, as a formal model of the language semantics would be required. This is hard for some languages, and impossible to create for others: e.g. in the case of C++, the grammar is undecidable, i.e. there cannot exist any program

```
1  int i, j, k;
2  k = ioread32();
3  i = 2;
4  j = k + 5;
5  while (i < 10) {
6      i = i + 1;
7      j = j + 3;
8  }
9  k = k / (i − j);
```

(a) C program with potential division-by-zero



(b) Control Flow Automaton for Figure 2.1a



(c) Single static assignment form of a path in Figure 2.1b

$$i[1] = 2 \wedge j[1] = k[0] + 5 \wedge i[1] < 10 \wedge i[2] = i[1] + 1 \wedge j[2] = j[1] + 3 \wedge i[2] >= 10 \wedge i[2] = j[2]$$

(d) SMT-expression of the path in Figure 2.1c

Figure 2.1: Mapping a program to a CFA

that parses all C++-compliant code correctly[2]. To overcome these kinds of problems, the input programs are first transformed into a formal model, which can be done separately, as a pre-processing step, either by hand or in an automated way. However, it is important to keep in mind that the verification result of the model checking algorithm will only be valid for the formal model that served as its input, and not necessarily the source program – for that, verifying the result against the program's code might be necessary.

One such formalism is called a *Control Flow Automaton* (CFA) [14], which is mainly used to model programs.

Definition 1 (Control Flow Automata). *A control flow automaton is a tuple* $CFA = (V, L, l_0, E)$*, where:*

- V*: A set of variables*
- L*: A set of locations, representing the program counter (PC) in the program*
- $l_0 \in L$*: The initial location*
- $E \subseteq L \times Ops \times L$*: Directed edges in the CFA, describing the set of operations to be executed when the program advances to a new location*
  - $op \in Ops$*: An* assumption *of a predicate over* V *asserting its truth (i.e. an execution is only legal if the predicate is fulfilled), or an* assignment *of a new value to a* $v \in V$*. A special kind of assignment has the form* havoc v*, which assigns a non-deterministic value to* v*.* ▪

An *execution* of a CFA is a path over the directed edges E, starting from l_0, where at least one variable assignment exists that satisfies all assumptions of this path. Note that in the CFA, we use non-constant *variables*, which means multiple values can be assigned to a variable in a single execution. To keep track of variable values, *indexed constants* are used, where the index is increased with each assignment to the same variable. Assumptions and expressions always use the most recent *indexed constant*.

Consider the example in Figure 2.1. The program in Figure 2.1a reads a non-deterministic number k, then does several calculations over the variables i, j, k that includes a potential

[2] https://blog.reverberate.org/2013/08/parsing-c-is-literally-undecidable.html

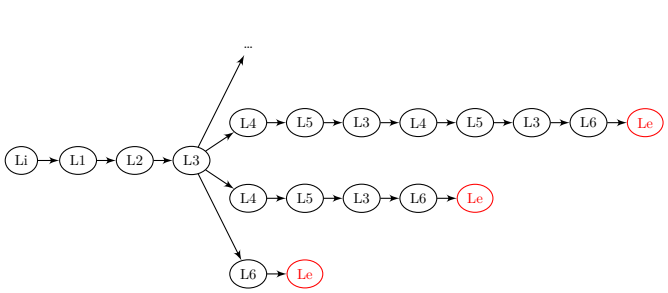
division-by-zero in line 9. This program is transformed into a CFA in Figure 2.1b, which includes an error location Le that represents the division-by-zero case, and a final location Lf which represents the successful termination of the program. An arbitrary path in this CFA leading to Le is presented in Figure 2.1c, which also shows the use of *indexed constants*. This format is called *single static assignment*, as every indexed constant is assigned exactly once, meaning each value is invariant throughout the execution. Figure 2.1d shows the path feasibility query as a satisfiability formula, which can be given to an SMT-solver that can determine whether the path is a legal execution. In the presented case, there is a clear contradiction in the expressions over $i[2]$: if $i[1]$ is 2 then $i[2] := i[1] + 1$ means $i[2]$ must be 3 – which contradicts the $i[2] >= 10$ assertion. This means the path is not a feasible execution, and we have not yet determined the safety of the program.

Note that while not explicitly shown, every *havoc* operation causes the index of the constant to increase, but nothing gets assigned to this constant. This means that the solver is free to choose any assignment, as long as it satisfies all other assertions that refer to this constant. Also, note that the example CFA in Figure 2.1b is not a correct mapping of the program in Figure 2.1a, as the values of the variables are entirely unbounded. This can be problematic if a path is found where the SMT solver reports the query as *satisfiable*, while in practice one of the variables would have wrapped around before reaching a value in the counterexample, making the path infeasible. This is a well-known limitation of SMT-based model checking, as fix-bit-width types cannot easily be mapped to mathematical integers. In the context of this work, best-effort practices are performed to counteract this phenomenon, namely, each *havoc* automatically implies an *assumption* over the variable's bounds (e.g. a C-like integer i will have an assumption that $-(2^{31}) \leq i \leq 2^{31} - 1$); and each *unsigned* integer type will wrap around when an out-of-bounds value is assigned to it, using modular arithmetic (e.g. a simple addition of the unsigned variable $u := u+1$ will be mapped to $u := (u+1)\ mod\ 2^{32}$). As signed overflow is undefined in the C standard (and most other programming languages) [33], that case is unhandled and the values might fall out-of-bounds. A correct solution to this problem is to use bitvector arithmetic instead of integer arithmetic, but the implied performance overhead warrants the presented more performant approximation.

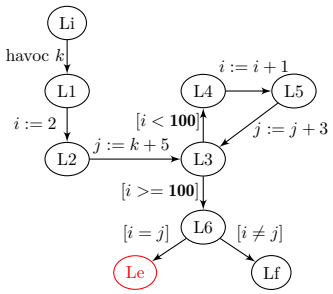
As we saw in Figure 2.1, the arbitrarily chosen path was not a feasible execution. However, that does not mean the program is safe – in fact, all paths would have to be checked first, to see if any produce the undesired division-by-zero problem. It is easy to see that due to the loop in the program, there are an infinite number of paths – to solve this model checking problem in a finite amount of time, many different approaches exist, but most of them fall into the following categories:

- Bounded Model Checking (BMC) – Starting from the initial state(s), check if an unsafe state is reachable within an expanding number of steps [17]
- k-Induction – Starting from the unsafe state(s), check if a safe state can be an expanding number of steps before any of the unsafe states [25]
- Explicit-State Model Checking – Mapping the states and transitions to an abstract state space using explicit-valued abstraction [31]
- Counterexample-Guided Abstraction Refinement (CEGAR) – Mapping the states and transitions to an abstract state, then refining the abstraction in subsequent iterations until a feasible counterexample is found, or safety is proven [21]

These techniques are generally used for different applications, e.g. BMC will usually find bugs the fastest but will not terminate if the state space is too big, while k-Induction is



(a) Some paths to the error state in Figure 2.1b



(b) A hard task for (naive) BMC

Figure 2.2: Bounded Model Checking example and limitations

often capable of proving safety while not finding actually reachable unsafe states. Abstraction based techniques can be more complex and therefore slower, but they can both find bugs and prove safety relatively effectively. In the context of this work, I examined algorithms based on the BMC-approach, while I implemented the introduced algorithms as part of a CEGAR loop. Therefore, I shall introduce these two approaches in detail below.

2.2.1 Bounded Model Checking (BMC)

As stated above, Bounded Model Checking (BMC) starts off with one or more initial states as the root of a tree graph, then repeatedly adds new states to the existing ones along transitions of the state space in a breadth-first-search (BFS) manner, i.e. all N-far states are added to the graph before any of the $(N + 1)$-far states. After reaching the upper bound of the analysis k, the graph is transformed into a satisfiability-modulo-theory (SMT) expression using the following recursive rule: *the expression at a node is the conjunction of its state expression and the disjunction of expressions in subsequent nodes.* This means that any junction will create an *Or* expression, and any paths will create an *And* expression. If this expression and the expression of the unsafe state are satisfiable together, the program is faulty as the unsafe state is reachable within this k-bound. Otherwise, the analysis continues building the graph with a new bound $k' > k$.

Take the example in Figure 2.1. The BMC algorithm will try to enumerate increasing-depth paths that end in the error location. The first few such paths can be seen in Figure 2.2a, but looking at the program in Figure 2.1a, these paths will be infeasible, as the value of i has not yet reached 10, i.e. the exit condition of the loop has not yet been fulfilled. After 8 iterations, and at a depth of 30, a feasible path will be found – if we assign -19 to k, the division will fail to due to the divisor being 0. Therefore, the program can be reported as *unsafe*.

Now consider the program in Figure 2.2b. The only difference to Figure 2.1b is the exit condition of the loop – instead of 8 iterations, the program must take 98 iterations before exiting the loop. For a naive BMC implementation, this means that it has to evaluate 97 infeasible paths before the counterexample is found. While possible, it might take a long time – and in the meantime, no proof of safety is possible as only specific traces are evaluated, rather than the whole program.

This trait of BMC (i.e. evaluating concrete traces rather than an abstract model) is both the source of its limitations, and its main advantage – if there is a bug, BMC is capable of finding it quickly, given it is not buried too deep in the program. This makes it appealing

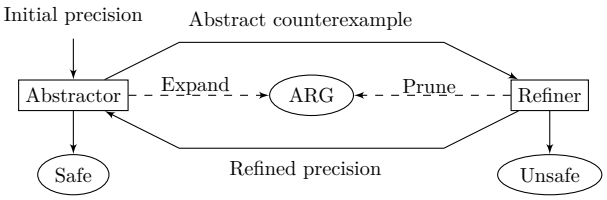


Figure 2.3: The CEGAR loop

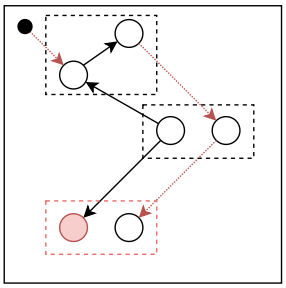


Figure 2.4: An ARG

to use when a guarantee of safety is not required, but there is an incentive for finding bugs.

2.2.2 Counterexample-Guided Abstraction Refinement (CEGAR)

As opposed to BMC, CEGAR is a tool both for proving safety and finding bugs. While implementations of CEGAR are generally slower than BMC-based tools due to the algorithmic overhead; CEGAR can handle a superset of the tasks that BMC could solve (see Section 2.2.2.3). This means that the verification power of CEGAR is at least as big as that of BMC.

2.2.2.1 A Generic CEGAR Loop

CEGAR is a highly configurable verification algorithm [21], where the main product of the workflow is the *Abstract Reachability Graph* (ARG) [13]. The ARG is an over-approximation of the reachable state space, meaning a reachability in the concrete model implies reachability in the ARG, but not necessarily vice versa. This property can be seen in Figure 2.4: even though the filled-in error state is seemingly reachable if only the abstract states (denoted by rectangles) are taken into account, there is no actual path among the concrete states that could lead to the error state. Therefore, this level of abstraction is too high, and a more refined version is necessary.

A notable feature of ARGs is state coverability: if a state is found to be *covered* by another state, i.e. there is another state whose truth value is implied by the truth value of the current state (e.g. $S_1(a = 2, b = 3)$ implies $S_2(a = 2)$, therefore S_2 *covers* S_1). In this case, there is no need to expand the current state any further, as any observed behavior will also be observed by the *covering* state. This is a vital tool for combating state space explosion, as this means that covered states are handled only once. For example, if a program contains a loop that does not influence the reachability of the error state (e.g. waiting for an input), the only state we are interested in is the exit point of the loop – any in-between states are covered-by the loop header state, and therefore not expanded further.

To achieve this continuous abstraction-then-refinement workflow, CEGAR works in a loop, as seen in Figure 2.3. This loop consists of two main algorithms working together: the abstractor and the refiner. The abstractor takes a precision describing the level of abstraction and the input model, and either creates a new ARG or expands the previously created, and pruned back ARG. This method is useful if a path is only found to be infeasible after a few feasible steps, and therefore the beginning of the ARG needs not to

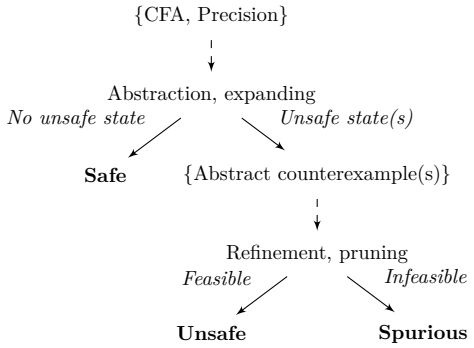


Figure 2.5: The CEGAR workflow

be re-created. When an ARG is ready, the abstractor checks whether an unsafe state is reachable, in which case the program is reported as *safe*. Note that safety is provable over an abstract state space, as it is always an over-approximation of the concrete state space and therefore the lack of an unsafe, abstract, reachable state implies the lack of an unsafe, concrete, reachable state as well.

If the created ARG does contain an unsafe state, the abstractor creates one or more *abstract counterexamples*, which are paths in the ARG leading to an unsafe state. These abstract counterexamples are then passed onto the refiner, which has multiple tasks: firstly, trace feasibility is evaluated (i.e. is at least one abstract counterexample concretizable, in which case the program is reported as *unsafe*). Then, if the counterexamples are infeasible, a new precision is created that is less abstract than the last one. Furthermore, the ARG is pruned back to the point where it became infeasible, then control is given back to the abstractor, where this abstraction-refinement cycle starts again.

The state of the algorithm after a single iteration of the CEGAR loop can have three values: *safe*, *unsafe* and *spurious*, as seen in Figure 2.5. The algorithm also produces proofs by default: for safety, a completely expanded ARG without an unsafe state suffices; and a feasible (concretizable) trace serves as the counterexample that shows the path to the bug in the program.

2.2.2.2 CEGAR Configuration Options

The CEGAR loop, as seen so far, is a declarative specification of the verification algorithm, i.e. only outcomes are specified and not the actual way to achieve said outcomes. This is due to the inherent configurability of CEGAR: as long as the parts are compatible with each other, many aspects of the algorithm can freely be swapped to other techniques fulfilling the same purpose.

As the possibilities are (almost) endless in terms of configurability, I only present the options provided by Theta[3], an open-source, generic and modular model checking framework developed at the Critical Systems Research Group of Budapest University of Technology and Economics [40]. In the context of this report, I developed the proof-of-concept implementations of the presented algorithms in Theta. This choice is in part justified by the maturity of the framework (the implementation has been validated on thousands of input models, e.g. in the SV-COMP 2021 software verification competition[4]), and also

[3] https://github.com/ftsrg/theta

[4] https://sv-comp.sosy-lab.org/2021/results/results-verified/gazer-theta.results.SV-COMP21.All.table.html

based on my previous contributions to the framework, which are prerequisites for the work presented in this report. All implementation-specific details are published in [30] – I will only introduce those relevant to my work.

THETA implements the CEGAR loop with complete modularity in mind. It provides several built-in options for each swappable component, as well as an easy way to define custom ones. The two (arguably) most important ones are the *abstract domain* and the *refinement algorithm*.

2.2.2.2.1 Abstract Domain

The *abstract domain* specifies the basis of the abstraction, and by default, there are two *pure* domains implemented in THETA: the *explicit value* domain and the *predicate* domain. The latter is further divided, based on the way multiple predicates are handled inside a single state – there are *cartesian predicate abstraction*, *boolean predicate abstraction* and *split boolean predicate abstraction* domains. As previous results showed that software verification does not usually benefit from boolean predicate abstraction [30], I only focused on the *explicit* (EXPL) and *cartesian predicate* (PRED_CART) abstraction domains.

Definition 2 (Abstract Domain). *Formally, an* abstract domain *is a tuple* $D = (S, \top, \bot, \sqsubseteq, expr)$ *[30], where:*

- S*: Lattice of abstract states (possibly infinite)*
- $\top \in S$*: Top element*
- $\bot \in S$*: Bottom element*
- $\sqsubseteq \subseteq S \times S$*: Partial order over the lattice* S
- *expr: A mapping from an abstract state to an actual data state (i.e. an expression)*

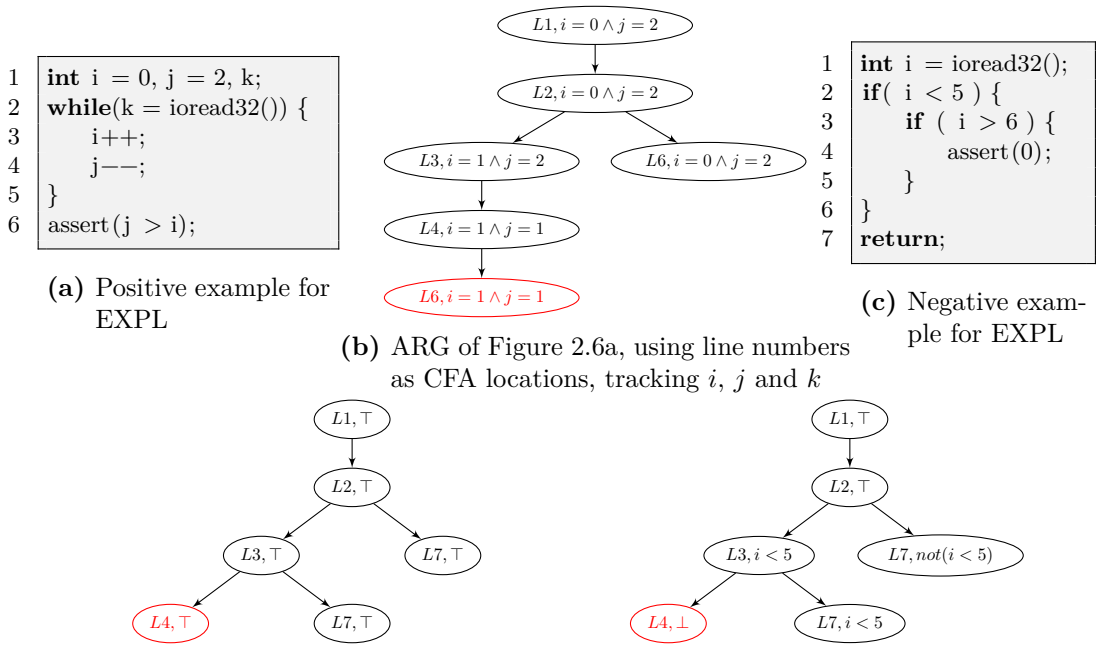
To define an abstract domain, one has to give a mapping for each member of the tuple D**.**

Explicit Domain The *explicit* abstraction domain defines the current abstraction precision as a set of *tracked* variables, i.e. variables whose values are of interest to us. Formally, the explicit domain can be defined as follows:

- S: A variable assignment of each *tracked* variable to a value of its domain, extended with top (arbitrary value) and bottom (no assignment possible) elements.
- $\top \in S$: No specific value is assigned to any of the tracked variables.
- $\bot \in S$: No assignment is possible to the tracked variables.
- $\sqsubseteq \subseteq S \times S$: $(s_1 \in S) \sqsubseteq (s_2 \in S) \iff (s_1 = s_2) \vee (s_1 = \bot) \vee (s_2 = \top)$
- *expr*: The conjunction of the equality expressions for each tracked variable and their value

Note that when applying CEGAR on a CFA, the locations of the CFA are always explicitly tracked, as to always have a $1 : N$ relation between locations and states in the ARG.

The *explicit* abstraction domain also specifies a *maxenum* value, which is an upper bound on the enumeration of values to a variable in a single step – which can be useful if the



(a) Positive example for EXPL

(b) ARG of Figure 2.6a, using line numbers as CFA locations, tracking i, j and k

(c) Negative example for EXPL

(d) ARG of Figure 2.6c, using line numbers as CFA locations, tracking i

(e) ARG of Figure 2.6c, using line numbers as CFA locations, tracking $i < 5$

Figure 2.6: Advantages and disadvantages of the EXPL domain w.r.t. PRED_CART

domain of a variable is infinite or very large. Consider the program in Figure 2.6a: tracking the value of k is next to impossible, as it is always assigned a 32-bit non-deterministic number. Enumerating all possible states leads to the state space explosion we are trying to avoid. Therefore, the algorithm does not try to assign a value to k in any of the abstract states in the ARG in Figure 2.6b, even though the precision would allow it – instead, k is kept at its top element. However, even without the value of k, the algorithm is capable of deciding the safety of the ARG: after just one iteration, the assertion fails. In this example, this abstract counterexample also corresponds to a concrete trace, and therefore the refiner will most likely report the program as *unsafe*.

Predicate Domain The *cartesian predicate* abstraction domain defines the current abstraction precision as a set of tracked (and ponated)[5] predicates. Formally, the cartesian predicate domain can be defined as follows:

- S: A conjunction of predicates
- $\top \in S$: *True*
- $\bot \in S$: *False*
- $\sqsubseteq \subseteq S \times S$: $(s_1 \in S) \sqsubseteq (s_2 \in S) \iff (s_1 \implies s_2)$
- *expr*: The conjunction of the predicates applicable to the current state

Consider the program in Figure 2.6c. If we tried to solve this reachability problem with the explicit domain, we would get the ARG in Figure 2.6d even at the maximal precision of tracking all (one) variables. The unsafe state is reachable in the ARG, and therefore the

[5] A ponated predicate means the outermost expression cannot be a *Not* operator.

abstractor cannot classify the input as *safe* – even though it is evident from the program's source that the assertion would never be reached, due to the contradicting $i < 5, i > 6$ assumptions. However, we cannot assign concrete values to i throughout building the ARG, as i can take up almost 2^{31} different values that would fulfil either criteria, and we can only evaluate one *if* statement at a time if the CFA contains different edges for them. (As a sidenote, this problem could also be solved by using large-block encoding (LBE) [15], but currently, Theta only supports a simple version of that).

In comparison, the cartesian predicate abstraction only needs the predicate $i < 5$ in the precision to deduce the safety of the program, as seen in the ARG in Figure 2.6e. Note the unsafe state in red: the ARG building algorithm correctly assigned the bottom element $\bot$ to its abstract state, as there was a contradiction in the path – meaning the ARG is complete and lacks unsafe states, and therefore the program is *safe*.

An aspect of the abstraction that was previously left out is the *transfer function*. The transfer function T maps sets of abstract states to the tuples consisting of an abstract state, a list of operations and a precision ($T : S \times Ops \times Prec \mapsto 2^S$). In practice, this determines the successor states of an abstract state in the ARG – which is a clear contradiction to the previous description of how an ARG is created (i.e. grouping concrete states together). While that was also a correct way of creating an ARG, it is not practical to create all concrete states just for being able to create abstract states out of it: instead, the transfer function is used to explore the abstract state space, and expand previously discovered abstract states. For example, given the EXPL domain, a precision tracking i, and an edge in the CFA assuming $i > 0 \wedge i < 4$ between locations l_1 and l_2; the state $s_0(l_1, \top)$ would have the following successor states: $\{s_1(l_2, i = 0), s_1(l_2, i = 1), s_1(l_2, i = 2)\}$.

Note that even though the transfer function assigns successor states to abstract states deterministically, the way these successor states are handled deeply influences the verification workflow: it is possible to visit and expand the first successor state in every instance, and therefore explore that state space in a depth-first manner (DFS), and it is also possible to visit all successors first before successors to those are visited and expanded (BFS). Any further combination of these techniques can also exist, such as an error-location-guided search (ERR), which will favor DFS more if the state is closer to the error location, but defaults to BFS otherwise [30].

2.2.2.2.2 Refinement Algorithm

As we have seen in Figure 2.6e, the predicate $i < 5$ in the precision was enough to guide the abstraction algorithm towards discovering that the program is *safe*. However, the discovery of this predicate is not trivial, and it can come from two sources: either from the initial precision (e.g. all assumes in the model), or the refinement algorithm will have to discover it while refuting the abstract counterexamples.

There are many different refinement algorithms implemented in Theta, but the relevant distinction among them in the context of this report is the following:

- Single-counterexample refinement: a single counterexample is generated from the unsafe ARG, and refuting it provides the new precision for the abstractor
- Multi-counterexample refinement: every counterexample is generated from the unsafe ARG, and a combined refutation provides the new precision for the abstractor

In the context of this work, I used 3 different refinement algorithms:

- BW_BIN_ITP: Single-counterexample, backward binary interpolation [30], using the longest feasible *suffix* of a trace.
- SEQ_ITP: Single-counterexample, sequence interpolation, using multiple expressions of increasing order to refute a trace.
- MULTI_SEQ: Multi-counterexample, sequence interpolation.

2.2.2.3 BMC Inside CEGAR

As mentioned above, CEGAR is at least as powerful of a verification tool as BMC, due to BMC being part of CEGAR. In order to justify this claim, consider the following:

- The CFA is extended with a counter c, which is increased after every statement in the model, starting with 0
- domain: EXPL
- initial precision: $\{c\}$
- search algorithm: BFS

This configuration will mimic the BMC algorithm, as it enumerates all paths in the program following the value of c. If BMC can find a counterexample, this method will be able to find it as well – and if BMC runs out of enumerable paths and classifies the program as *safe*, this technique will arrive at the same conclusion as well.

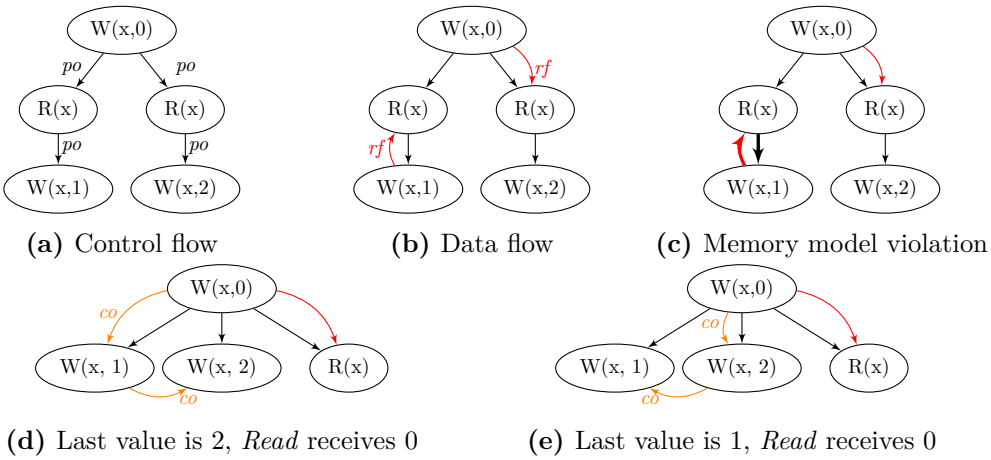
2.3 Multi-Processor Architectures

Modern hardware architectures almost universally offer concurrent memory access in a relaxed way. This means that read and write operations do not have to execute sequentially, the memory controller is free to reorder them (respecting certain constraints) to increase performance. The rules for such relaxed accesses is described by a *memory consistency model* (MCM). The specification of MCMs evolved from textual documentation through small "Litmus-tests" describing forbidden outcomes to well-defined axiomatic formal specifications of the execution semantics [10, 8, 38].

2.3.1 Memory Consistency Models

Generally, a memory model of an architecture can either be *operational* or *axiomatic* [8]. The former uses elements of the hardware platform such as queues and buffers to explain certain behaviors on the target architecture, which makes it easier to implement the architecture directly in hardware, but hinders reasoning on the software side [38]. In contrast, an *axiomatic* memory model uses a declarative approach to forbid certain sets of relations over memory accesses [10]. This approach proved to be better for reasoning about possible executions of concurrent programs, and therefore most software verification tools employ an *axiomatic* model to provide information on the guarantees of the hardware architecture [6, 4, 3, 27, 24] or the programming language [34, 36].

One *axiomatic* memory modeling language is CAT [9], which has been created to specify memory models for HERD [8] but has since seen widespread adoption due to its succinctness



(a) Control flow **(b)** Data flow **(c)** Memory model violation

(d) Last value is 2, *Read* receives 0 **(e)** Last value is 1, *Read* receives 0

Figure 2.7: Candidate executions

and expressivity. However, we further restrict the elements of the language to its core subset as seen in DARTAGNAN [24], as constructs such as *thread scoping for heterogeneous architectures* fall outside of the scope of this work. This leaves us with the primitive relations $\{po, rf, co\}$ for *program order*, *read-from* and *coherence order* respectively; event sets $\{\mathbb{F}, \mathbb{R}, \mathbb{W}\}$ for *fences*, *reads* and *writes*; and relation sets $\{\mathbb{E}, \mathbb{I}, \mathbb{L}\}$ that can be used to refer to *inter-thread*, *intra-thread* and *same-location* relations. Further expressions can be derived through set- and relational operators on the above primitives and other derived expressions. The relations then can be declared *empty*, *irreflexive* or *acyclic* to forbid certain combinations of memory access interactions, effectively expressing the constraints of the memory model.

The CAT language uses the notion of *candidate executions* to express a set of relations that corresponds to a specific execution of memory accesses in a program. Firstly, a *candidate execution* shows the *control flow*, i.e. the path of memory accesses in the program that the candidate execution observes as seen in Figure 2.7a. This creates a directed, acyclic graph, in which each time a node has more than one out-edge the program observes a *fork*, i.e. the creation of one or more new threads. Note that decisions inside a thread do not appear directly in the candidate execution, as each path represents a locally deterministic execution of a thread. Secondly, a *candidate execution* also shows *data flow*, e.g. in the form of *read-from* edges, showing which *Write* event creates the data element a *Read* receives, as seen in Figure 2.7b.

Such *candidate executions* can be checked against the memory model specification to decide if they are *consistent* or *in violation* with the candidate. For example, against the specification that no *Read* on a given thread shall receive a value from a later *Write* on the same thread (any path $(po \mid rf)^+$ is acyclic), the example *candidate execution* in Figure 2.7c is in violation of the memory model. However, if a *candidate execution* is indeed *consistent*, it becomes an *execution graph*, describing an observable outcome on the given architecture.

Note the lack of other primitives on the execution graph. By populating the *po* and *rf* relations the necessary control and data flow is fully defined, and no further relation is necessary – most notably, the *coherence order co* [8, 24] (or *modification order* [34]), i.e. the total ordering of same-location *Write* events, is entirely superfluous. Consider the two *candidate executions* in Figures 2.7d and 2.7e, where this relation is not omitted. In both cases, the *Read* will read 0 from memory, but the order of the *Write* events differs. The outcome and overall execution of the program is not influenced by these differences

other than the final observable value in memory. However, as the two executions are not the same graph, both variants will be produced when enumerating the possible *candidate executions*. Unless the state of the global memory is ever needed to be queried, it is enough to determine for each *Read* event the set of *Write* events it might receive data from. This does not mean that any derived relation that uses *co* as one of its operands will have to be completely re-written, but rather that we can simply leave *co* as unspecified with a few constraints: as long as there is a *co*-order that relates every same-location pair of writes in either in exactly one way (i.e. if elements a and b are same-location writes, either $co(a, b)$ or $co(b, a)$ is true, but not both), the execution is feasible.

2.4 Analysis of Multi-Threaded Programs

Concurrent software verification algorithms also fall into two categories, depending on the execution semantics they employ. The *interleaving* semantics uses overlapping traces of the threads in the concurrent program to explain how it executes. This approach, when used naively, does not scale well due to the large number of possible unique executions. This is partially solved by utilizing e.g. *partial order reduction* (POR) [28], which significantly reduces the number of necessarily explored executions. In contrast, the *declarative* semantics of concurrent program executions uses partial orders to explain a specific execution. The *candidate executions* introduced above are examples to such a semantics, as the *po* and *rf* relations partially order the statements and yield a well-defined single execution of the program [8]. This *declarative* semantics has been shown to perform better on weak memory than the *interleaving* semantics over sequential memory, when implemented in model checking algorithms [7].

2.4.1 Interleaving Semantics

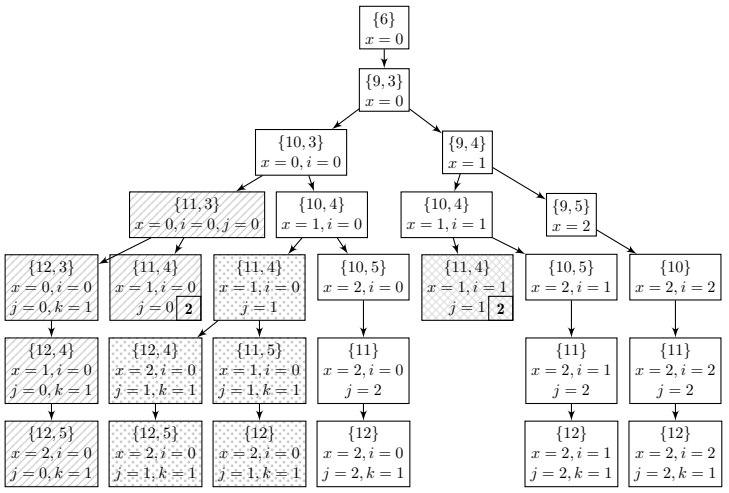
The naive way of dealing with concurrency is to strictly follow the definition of asynchronous systems, i.e. any of the threads may execute at any point in time, meaning every possible total order of the instruction has to be explored. This technique employs *naive interleaving semantics*.

Consider the example in Figure 2.8a (note that based on the C standard, a global variable will be initialized to 0, if no explicit value is assigned [33]). The main thread starts a worker thread, which writes two values to x, while the value of x is read twice. As the value of x increases monotonically, we assert that the latter read's value shall be at least big as the former's. We store this in a boolean k. If we tried to enumerate all executions based on the naive interleaving semantics, we would get the state space in Figure 2.8b – there are 10 different executions that can take place, as the $3 + 2$ operations between the start and end of the worker thread give rise to 10 total orders. However, if we examine the outcomes of the different executions, the set of possible end values is way smaller: x is always 2, j is always bigger than i and k is therefore always 1. Even though i and j can take up any one of the values from the set $\{(0, 0), (0, 1), (0, 2), (1, 1), (1, 2), (2, 2)\}$, this is still only 6 possible outcomes instead of the 10.

To explain this behavior, let us examine the three branches of the state space tree marked with patterns. In these executions, the values to i and j were decided early on, as the main thread progressed more than the worker thread. In theory, this would eliminate the need for further analysis, as any further action on either thread is independent of the other – x will be increased further, but no operation will use its value; and k is never used in any

```
1  int x;
2  void* thr(void* _) {
3    x = 1;
4    x = 2;
5  }
6  int main() {
7    pthread_t t;
8    pthread_create(&t, 0, thr, 0);
9    int i = x;
10   int j = x;
11   _Bool k = (j >= i);
12   pthread_join(handle, 0);
13   assert(k);
14 }
```

(a) Concurrent program testing sequentiality



(b) State space of Figure 2.8a, using tuples of line numbers as locations (States with bold numbers are not fully expanded)

Figure 2.8: State space exploration based on naive interleaving semantics

of the global memory accesses. However, the naive interleaving approach had to explore these subexecutions as well because it had no way of determining which operations would influence the final outcome and which ones would not.

An intuitive step to take is to discover independent pairs of transitions in the model, and forbid the exploration of both total orders. This technique is called *partial order reduction* (POR) [28], and it is widely used in the verification of concurrent systems (Even though there is a specialized version of POR called *dynamic partial order reduction* (DPOR) [26], which is shown to be more optimal, introducing and implementing that algorithm falls outside the scope of this work).

Consider the same input program in Figure 2.8a. If we apply POR based on a global-local partitioning of the transitions, where every transition touching a global memory object is considered dependent on each other, we get the state space in Figure 2.9. In this case, state space exploration was *optimal*, as each explored total order yielded a different outcome, and no possible outcome was left out.

Even though the presented example showed the POR algorithm to be *optimal*, this is not the case in every input program. For example, there could be another, totally independent y global variable, and two threads performing the same operations over y as over x – in this case, all total orders would have to be explored among accesses to the global variables as well, which would yield a suboptimal exploration. There are techniques mitigating this behavior (e.g. in [2], the authors have shown that there is an optimal DPOR, and also gave an example for such an algorithm), but the presented naive POR cannot deal with this problem.

2.4.2 Declarative Semantics

To showcase the differences between the interleaving and declarative semantics, let us look at the same problem in Figure 2.8a. To generate the declarative state space of the program, an *abstract execution graph* is necessary – which is similar to a candidate execution, but Reads are not limited to a single *rf*-edge, and only *po*- and *rf*-edges are present. The semantics of such a construct is the following: all executions are *observable* which stem

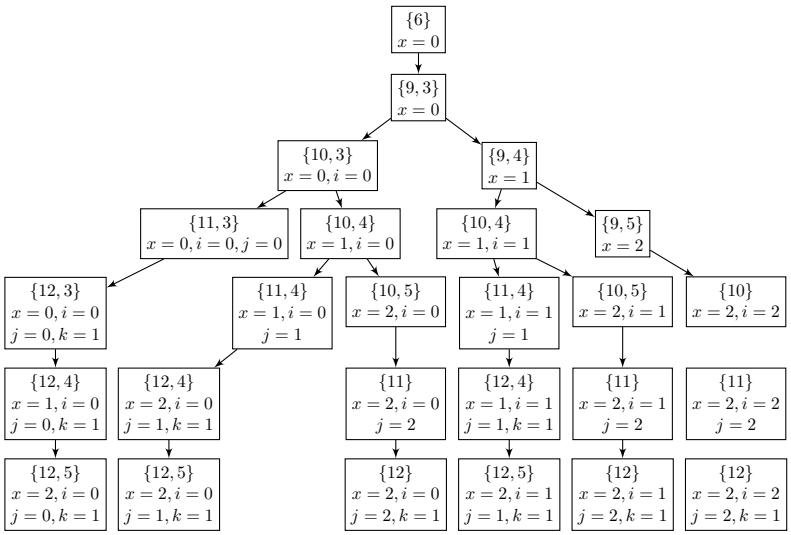
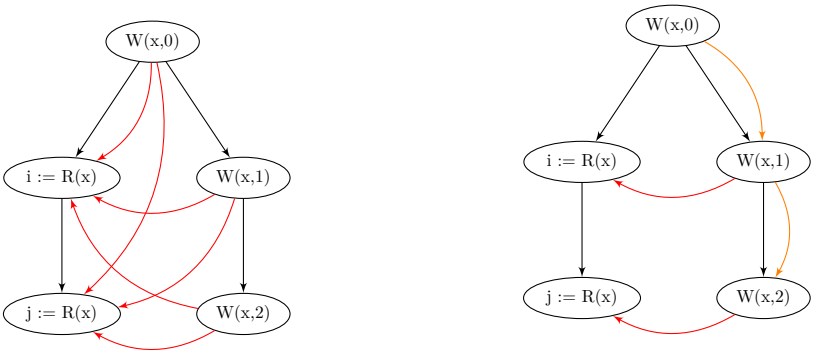


Figure 2.9: POR-based state space of Figure 2.8a, using tuples of line numbers as locations (main thread executes first)



(a) Abstract execution graph of Figure 2.8a

(b) A concrete execution based on Figure 2.10a

Figure 2.10: Program verification based on declarative semantics

from a consistent candidate execution that is a subgraph of the abstract execution graph, and for which a satisfying total *co* order exists over same-location writes. Such an abstract execution graph can be seen in Figure 2.10a, and an example concrete execution is shown in Figure 2.10b.

Note that in this case, state space exploration is optimal by default: after the abstract execution graph is built (which is interleaving-free, and therefore can be built in a single-pass over the operations in the program), only different, and consistent execution graphs are enumerated.

2.4.3 Multi-Threaded CFA

In order to verify multi-threaded programs, a formalism supporting multi-threading is also necessary. As with single-threaded programs, a formalism encoding control-flow in the form of a program counter-like construct is advantageous – therefore, the basis of the chosen formalism is still a control flow automaton (CFA) [14]. However, this formalism has been extended in the following ways, giving rise to the eXtended Control Flow Automata (XCFA):

Definition 3. *eXtended Control Flow Automata (XCFA)*
An XCFA is a tuple $XCFA = (V_g, P)$*, where:*

- V_g*: Global variables*
- P*: Processes, which are tuples* $P = (V_p, F, f_0)$*, where:*
  - V_p*: Thread-local variables*
  - F*: Procedures, which are tuples* $F = (V_l, CFA, P_{in}, P_{out})$*, where:*
    - V_l*: Local variables*
    - CFA*: A conventional CFA (which can use* $V_g \cup V_p \cup V_l$ *as variables), extended with the following operations:*
      - Function call*s*
      - Start thread *and* join thread
      - Atomic begin *and* atomic end
      - Store, Load *and* Fence
    - $P_{in} \subseteq V_l$*: Input parameter variables, which are assigned when the function is called*
    - $P_{out} \subseteq V_l$*: Output parameter variables, which are returned when the function returns*
  - $f_0 \in F$*: The main function of the process (execution starts here)*

Semantically, an XCFA can either be static *or* dynamic*. In the former case, only the starting set of processes can execute. In the latter case, the* start thread *and* join thread *operations manipulate the set of enabled processes. In both cases, the processes fire* asynchronously. ▪

Note that variables can either be assigned via normal assignments (as in a conventional CFA), or through *store* and *load* operations. In the context of this work, I assume total sequentality for assignments, and only apply the memory model for the analysis of the designated memory access instructions.

Chapter 3

Related Work

My contributions presented in this report cover a survey of the applicability of CEGAR on concurrent programs, handling both the sequential and weakly ordered case. In this chapter, I introduce the state-of-the-art tools for handling concurrency in these two cases. For the sequential case, I concentrate on the algorithms employing a form of CEGAR, as that falls the closest to the scope of my work; while for the weakly ordered case I introduce the most advanced *bounded* algorithms, due to the lack of a general solution covering infinite-state programs[1].

3.1 Sequentially Ordered Concurrency

Most model checkers capable of verifying concurrent programs that employ a form of abstraction-refinement techniques use a pre-processing step to determine atomically executable (i.e. thread-local) transitions and global operations. Then, every interleaving is explored among these transitions when calculating abstract successor states. Note that these solutions employ a crude version of POR, with no dynamic element.

VVT [29] uses an LLVM-based front-end to verify C programs, which determines blocks of instructions that can execute atomically without interfering with the allowed set of outcomes. Then, these blocks serve as the individual transitions in a large-block encoded CEGAR loop.

In comparison, CPAchecker [12] uses a pre-processing step on the edges of the CFAs to determine thread-local and global operations, then uses a similar large-block encoded CEGAR loop. In addition, it uses several further optimization steps to be more performant; such as *waitlist ordering* and *partitioning of abstract states*.

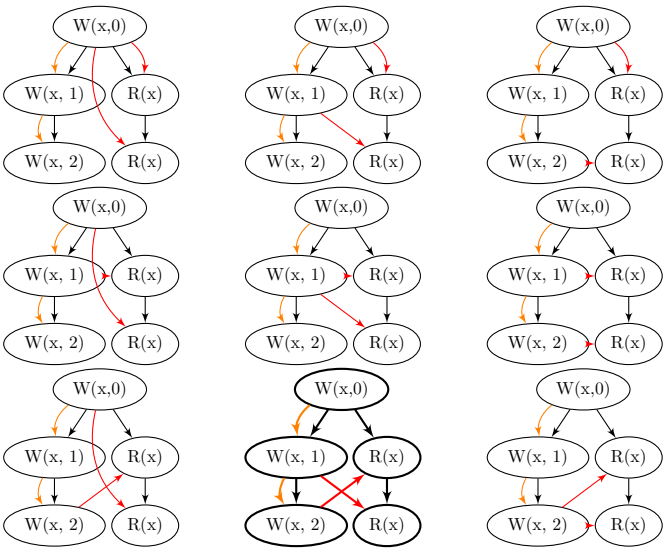
3.2 Weakly Ordered Concurrency

The algorithm presented in this report has been heavily influenced by three existing tools, namely, Herd [8], Dartagnan [24] and rcmc [34]. In this section, I will present the approaches employed by these tools.

[1] At the time of writing this report, I have no knowledge of any approach that utilizes any form of abstract reasoning over declarative semantics.

```
ExampleArch same−loc
{ x=0; }
P0            | P1
MOV [x],$1    | MOV R0,[x]
MOV [x],$2    | MOV R1,[x]
exists  (1:R0=2 /\ 1:R1=1)
```

(a) Litmus test *same-loc*



(b) Candidate executions with fixed *co*- and *po*-orders

Figure 3.1: HERD's input litmus test and the generated candidate executions

3.2.1 Herd

HERD is a memory model simulator [8]. It expects a memory model specification written in the CAT language [9] and a litmus test. Litmus tests are small, assembly-level concurrent programs that include accesses to global memory, as well as constraints on local variables. Litmus tests are widely used to specify guarantees of memory models, e.g. Intel most notably only uses such programs as the specification of the X86 memory model [20]. For example, a memory model rule might forbid the reordering of same-location accesses. The corresponding litmus test in Figure 3.1a has two threads: a producer with two consecutive *Write* events, and a consumer with two consecutive *Read* events, all to the same location. Any execution is forbidden where the consumer observes the two written values in reverse order, i.e. the value of $R1$ is 1 from the earlier *Write*, while the value of $R0$ is 2 from the second *Write*. This outcome is only possible when either the *Reads* or the *Writes* have been reordered.

For a given memory model and litmus test, the question is whether the forbidden behavior is *observable* on the target architecture. To answer this question, HERD will first generate all candidate executions of the litmus test. This is done in an enumerative way: for each primitive relation every semantically correct combination will be explored [8], as seen in Figure 3.1b. After enumeration, the candidate executions are filtered based on whether they are consistent with the specified memory model. If any consistent execution graph of the litmus test produces the forbidden outcome, the specified behavior is *observable* and the litmus test fails. For the example in Figure 3.1a, there is one such candidate execution (given fixed *co*- and *po*-orders), highlighted in bold in Figure 3.1b.

The example in Figure 3.1 also shows that the number of *candidate executions* is generally much higher than the number of *consistent execution graphs*. Given a memory model rule that forbids the reordering of same-location accesses, only 6 *candidate executions* are *consistent* out of the 9 in Figure 3.1b given the fixed *co*-order. However, the total number of *candidate executions* are much higher. The *Write* events can be ordered by any of their permutations, as the algorithm cannot assume that any of those partial orders is inconsistent without taking the memory model into account. However, given the forbidden

```
int x = 0;
void thr1(void* _) {
    x = 1;
    x = 2;
}
void thr2(void* _) {
    int r0 = x;
}
```

Figure 3.2: Example input program

```
int x = 0;
int y = 0;
void thr1(void* _) {
    y = 1;
    x = 1;
}
void thr2(void* _) {
    int r0 = x;
    int r1 = y;
    assert (!( r0 == 1 &&
               r1 == 0 ));
}
```

Figure 3.3: Input causing false positive result over SC

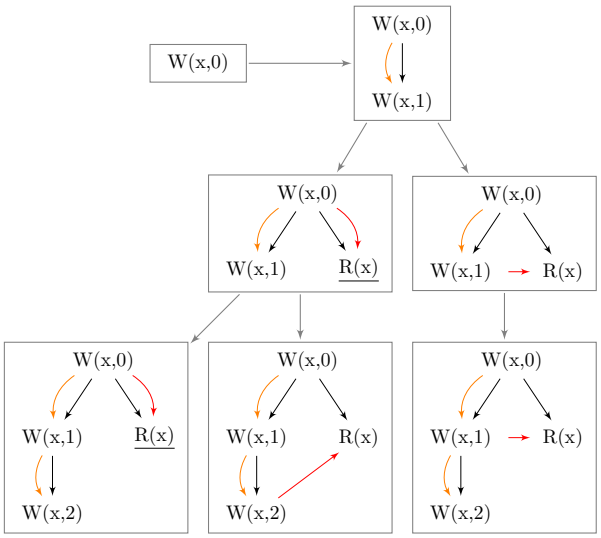
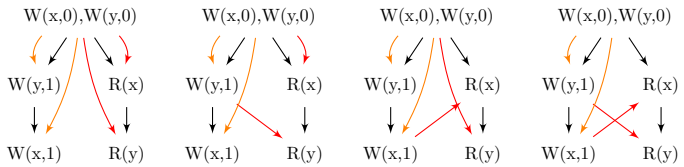


Figure 3.4: Exploring the program in Figure 3.2



(a) $r0 = 0$, $r1 = 0$ **(b)** $r0 = 0$, $r1 = 1$ **(c)** $\underline{r0 = 1}$, $\underline{r1 = 0}$ **(d)** $r0 = 1$, $r1 = 1$

Figure 3.5: Execution graphs generated by RCMC

same-location reordering, only the one in Figure 3.1b is consistent with the memory model. This puts the number of all candidate executions at $3! * 9 = 54$, and the percentage of consistent execution graphs at 11.1%. For larger programs, this ratio is even smaller, as the number of unnecessary partial orders becomes higher. This observation is also established by the practical evaluation of the RCMC tool, which only generates consistent execution graphs [34].

The goal of HERD is not general program verification, but rather architectural verification. Litmus tests are by definition small programs, and therefore it is unnecessary to optimize the algorithm in HERD for input size. For anything larger than an ordinary litmus test, HERD will most likely time out while enumerating the candidate executions. This prompted the development of smarter *candidate execution* generation, such as RCMC [34].

3.2.2 Rcmc

The novelty of RCMC is its smart exploration algorithm. In each step of its algorithm, RCMC will only generate consistent execution graphs, and no execution graph is ever explored twice. The implemented *stateless* model checking algorithm receives a concurrent C/C++ program with optional assertions, and enumerates all consistent executions as its output. If in any of the execution graphs the assertion is violated, or a non-atomic concurrent

access occurs, the tool reports the program as *unsafe* immediately. Note that the memory model is *not* an input, as the C/C++ concurrency model (as formalized in the repaired RC11 memory model [35]) is hard-coded into the algorithm. This significantly reduces the applicability of the tool for custom architectures and potentially yields false positive results.

Consider the input program in Figure 3.2. Two threads are executing concurrently, one writing to memory and another reading from it. Note that atomic accesses have been replaced with regular assignments for the sake of brevity. For the sake of this example, *relaxed* accesses can be assumed.

The execution of the algorithm can be seen in Figure 3.4. RCMC will start by recording the initial values in a node, then one-by-one adding the statements of the program. Any time a *Read* event is added, each subexecution is explored where *Read* receives a value from any existing *Write* event. In exactly one of the subexecutions, *Read* remains *revisitable*, i.e. a later *Write* can provide it a value. *Revisitable* nodes are underlined in the example. Each time a *Write* event is added, each subexecution is explored where the newly added *Write* provides a value for any combination of currently *revisitable Reads*. Furthermore, each consistent *co*-order is also explored, but in Figure 3.4, this is deterministic due to *po*. In the example in Figure 3.4, the order of recorded nodes alternates between the two threads, starting with a *Write* event to x.

The novelty behind the algorithm is to use *revisitable* nodes to mark a single subexecution where a given *Read* event's value is not final. If more than one such subexecution existed, adding a subsequent *Write* event could generate redundant subexplorations [34].

Consider the input program in Figure 3.3 and the generated execution graphs in Figure 3.5. Depending on the received values in the second thread, an assertion failure can occur. The condition of the assertion means that the second *Write* event executed *before* the previous one. This is observable in Figure 3.5c. Considering C/C++ can generally run on any architecture, one cannot assume that the hardware is not e.g. sequentially consistent (SC). SC guarantees that no statements will be reordered, and therefore the assertion is never violated. RCMC, however, reports it as unsafe because C/C++ does not guarantee this assumption, and therefore this can be categorized as a false positive result. This is not a shortcoming of the algorithm itself, but rather of the approach: one cannot assume that the memory model of a programming language is independent of the target architecture [41]. Such a false result might shadow actual problems in the input program, and is therefore inherently unsafe.

Another problem of RCMC is the suboptimal exploration of executions when multiple threads write the same variable. As noted above, exploring artificially generated *co*-orders is detrimental to the number of execution graphs. In the worst case, each new *Write* event will effectively multiply the number of subexecutions by the factor of existing *Write* events to the same variable, even if only one thread observes the value. In this case, enumerating all execution graphs where this *Read* reads from a different *Write* would suffice, yet this is multiplied by the factorial of the number of *Write* events, as seen in Figures 2.7d and 2.7e.

3.2.3 Dartagnan

Most of the concerns above are addressed by DARTAGNAN, a bounded model checker that uses memory models as modules [24, 27]. DARTAGNAN expects a concurrent program and a memory model as inputs, and using the conjunction of SMT-encoded expressions determines whether an unsafe state is reachable within a given bound. To achieve this,

	Software verification	Parametric memory model	Scalable	Optimal execution enumeration	Handle unbounded state spaces
Herd	✗	✓	✗	✗	✗
rcmc	✓	✗	✓	✓*	✗
Dartagnan	✓	✓	✓	N/A	✗

Figure 3.6: Comparison of related verification tools

Dartagnan unrolls and encodes the concurrent program as an SMT-expression; encodes the unsafe state as another SMT-expression; and encodes the input memory model as an SMT-expression. If the conjunction of the expressions above is *satisfiable*, the unsafe state is *reachable* and therefore the concurrent program is *unsafe*.

Dartagnan is a software verification tool, complete with an integration to Smack [37], an LLVM-based program transformation tool that allows Dartagnan to work on formal models rather than source-level programs. The gap between the higher-level LLVM-IR and the ISA of the target architecture is bridged by using *compiler mappings* for translating e.g. memory ordering primitives. This is a conventional procedure [41], but special attention has to be paid to ensure the compiler mappings represent an actual compiler's behavior that might be used to compile the examined program later on.

In comparison with Herd and rcmc, Dartagnan (and its companion tool, Porthos [24]) is not capable of enumerating consistent executions. Even though as a reachability checker, Dartagnan is not expected to provide this feature, it could be useful to provide a way to use the tools embedded into other verification algorithms for handling concurrent parts of an otherwise independent set of threads. In this case, an unsafe state might not only be dependent on the concurrent parts of the program, and therefore Dartagnan could not handle it on its own.

Evaluating the five criteria in Figure 3.6 reveals that none of the tools fulfil every aspect. Herd is not capable of software verification due to scaling issues caused by its suboptimal execution enumeration approach. rcmc is not parametric and therefore only C/C++ guarantees are assumed, and it uses artificially generated *co*-orders which increase the number of explored execution graphs. Dartagnan cannot enumerate consistent execution graphs. Furthermore, neither solution can handle infinite-state programs.

Chapter 4

CEGAR for Declarative Semantics

The main contribution I present in this report is an algorithm that handles the abstraction-refinement of weakly-ordered concurrent programs using declarative semantics. In this chapter, I introduce the challenges that one has to overcome in order to get a *sound* and *performant* algorithm.

4.1 Outline of the Solution

As input, the verification algorithm receives two inputs:

1. A multi-threaded program P in the form of an *XCFA*, and
2. An axiomatic memory model M conforming to the semantics of the CAT language [9].

As an output, it produces one of the following:

- Proof of safety in the form of an ARG lacking unsafe nodes, or
- A counterexample showcasing the necessary control- and dataflow to reach the unsafe state.

To provide a generic CEGAR-based solution, the components of the algorithm fit inside components of the CEGAR loop in a well-defined way (see the patterned boxes in Figure 4.1):

- LTS: The labelled transition system supplying the possible (not necessarily enabled) transitions from a given abstract state
- Init: The initialization function, providing at least one initial state based on the given initial precision
- Trans: The transfer function, providing a mapping from each abstract state-transition pair to a set of new abstract states
- Prec: The precision of the current iteration
- Ord: The partial ordering of the abstract states, given a partial ordering of the data states
- State: The abstract state

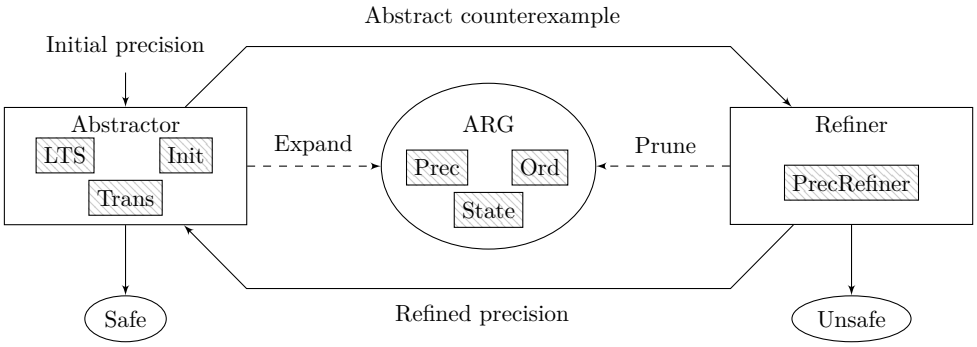


Figure 4.1: The CEGAR loop, extended with the components of the novel algorithm

- PrecRefiner: The precision refining algorithm that creates the new precision based on the last one and the refutation

Given the existing concepts of CEGAR such as the abstract domain, refinement algorithms and the ARG builder; if the components above are defined in a standardized way described by the CEGAR loop, they can re-use these concepts to decrease the complexity of the algorithm. Therefore, I only describe the inputs, outputs and inner behavior of these components, leaving the interaction among them deliberately undefined – these are implementation-specific details, about which Chapter 5 and the tool paper of Theta [30] provides greater insight.

The main idea behind the verification approach is the following: each node in the ARG will correspond to an abstract execution graph. When a node forks into multiple successors, it means that the successors differ in at least one choice: either a different assumption is taken, or a different *rf*-edge is recorded. This is key for a sound approach: every possibility is explored for a definitive proof of safety.

4.2 Motivating Example

Consider the following program:

```
unsigned a = ioread32();      atomic_begin
unsigned b = ioread32();      int i = y;
atomic_begin                  int j = x;
x = a;                        atomic_end
y = a + b;                    if( i < j )
atomic_end                        ERROR: exit(-1);
```

It has many aspects that make it challenging to verify: there are atomic blocks, writes and stores, as well as non-deterministic inputs. However, as I will show, using the novel algorithm it is straightforward to build a proof of safety, assuming sequential accesses and ordinary C semantics [33] (See Figure 4.2). Note that I use sequential accesses to make it easier to follow the example, but in this case, any ordering primitive could have been used – due to the atomic blocks, threads are synchronized in any case.

To start off the verification, an initial state and a precision is necessary. The former is chosen to be the following: the threads have started, all global variables (x, y) are initialized to 0 (which also shows as a Write operation in the execution graph), and a single node is added to the abstract execution graph for every thread, *po*-after the initial writes. The precision shall be an empty predicate precision (i.e. no predicates are tracked).

```
1  SIMPLE_C
2
3  let com = (rf | fr | co)
4  let come = com & ext
5  let cometr = (come | come^-1)
6
7  let hb = po | com
8  acyclic hb as sc
9
10 empty amo & (cometr;hb*;cometr)
```

Figure 4.2: Semantics of sequential accesses in C

From here on, always a single thread is chosen that can execute, but no interleavings are explored beyond that – i.e. an arbitrary sequence of the threads' operations dictate the flow of the verification. Note that this sequence need not be an actually valid overlapping of the operations.

Let us assume that this arbitrary sequence starts off with the reader (right) thread. The first operation is the beginning of an atomic block (i.e. a sequence of instructions which, once started, must fully execute), which will cause the successor state to include a flag for the thread's atomicity.

Next, the value of y is read, and loaded into the local variable i. As the current precision does not include any predicates over y, execution continues and a single successor state is created. This state includes a single read operation in its abstract execution graph, which is *po*-after the initial node. Furthermore, an *rf*-edge is added between the initial write ($W(y, 0)$) and the read. Thereafter, the value of x is loaded into j, causing a similar result. The state now includes two reads, *po*-after each other and the initial node of the thread, as well as *atomically together* – which is also shown in the abstract execution graph as a (reflective) *amo*-relation for the reads. The graph also contains two *rf*-edges, between the respective initial writes and reads of the global variables.

It is important to note that a non-tracked global variable will cause a similar action as a *havoc* statement would – because we cannot be sure if all writes have been added to the graph yet, we do not want to constrain an assumption over the loaded variable – this could lead to potential *missed bugs*.

The next instruction ends the atomic block, which deletes the atomic flag for the thread from the state. Then, the *if*-statement is to be evaluated. As we *havoc*-ed the variables i and j, and there are no tracked predicates contradicting the assumption, we take both paths: the one leading to the error state, and the one leading to the end of the thread. As we have a path to the error location, we stop expanding the ARG, and pass the path to the error location to the refiner.

Counter to the abstraction, the refinement cannot consider loads from non-tracked variables *havoc*s, as that would lead to potential *false alarms*. For example, in this instance, we know that the program is safe, but we have an abstract counterexample that shows otherwise – if we accepted it, the program would be determined unsafe, when in fact it is the opposite.

Instead, the refinement algorithm considers every possible and compatible *rf*-subset that could stem from the abstract execution graph in the error state. This quickly shows that there is a contradiction in the counterexample – those reads could only get their values from the initial writes, which are 0, and therefore the assumption about i being smaller

than j cannot be true. However, if we passed the same counterexample to the refinement algorithm with the *havoc*s still in place, we would get an answer that the counterexample is feasible – and therefore we know that the problem is in the global variables. (Note that without this second check, it would also be possible that an ordinary over-abstraction took place, which can be handled without touching the global variables at the moment.)

To prepare a refutation, one has to decide two things: *where* the counterexample became infeasible (to show the boundary for ARG pruning), and *how* the counterexample became infeasible – in ordinary single-threaded CEGAR, this would be an *interpolant* of some sort that could be used as a predicate in itself, or the variables of which could become tracked variables in the EXPL domain. In this case however, we know that the memory model was the culprit. There are many options to choose from, but in this instance, let us assume that *every non-tracked global variable in the counterexample* needs to become tracked. To fit in the framework of CEGAR, this can be signalled as a self-evident predicate of the form $var = var$ – which will not influence the verification process in unrelated parts of the algorithm (due to it always being true), but can be used to give information on trackable globals for the precision refiner.

To finish up the refinement process, the ARG is pruned back to the first occurrence of a non-tracked global, and a new precision is created with the predicates $x = x$ and $y = y$. Then, control is given back to the abstractor.

The abstractor receives control and a pruned-back ARG that only contains the first non-initial state, i.e. the atomic flag has been raised but no reads have been added. When handling the first read however, the situation is different from last time: the global variable is present in the precision. This triggers a different reaction, in which the state forks into two successors: one where the read received its value from the initial write (and therefore the corresponding events are added to the abstract execution graph); and another where the read simply *blocks* – i.e. the execution on the given thread shall not continue until a suitable value is found. This is called a *revisitable* read (following the nomenclature of RCMC [34]), as it can be revisited later to update its value.

The subsequent read is handled the same way: either reads 0, or blocks the thread. These actions have lead to three active states in the ARG:

1. The first read has blocked
2. The first read has read 0, the second read has blocked
3. Both reads have read 0

As the 3$^{\text{rd}}$ state did not block, execution can continue on the same thread. However, the *if*-assumption is clearly not true, as both i and j are 0. This means that the created leaf in the ARG is safe.

Let us abandon the 2$^{\text{nd}}$ state in favor of the 1$^{\text{st}}$. As the first read has blocked the thread, we have to change threads to the other one. It starts off by assigning two non-deterministic unsigned values to a and b (which has no effect in the state, as a and b have been non-deterministic anyway), then similar to the first thread, starts an atomic block.

As a next step, the thread stores a into x. This results in the usual creation of a new node (a Write in this instance), which is *po*-after the last node. However, this write has nowhere to supply its value, as the read from x has not yet happened on the other thread. This prompts execution to continue as if nothing had happened. However, the next operation stores a value (namely, $a + b$) into a variable (y) that *does* have revisitable reads waiting to take a value: this will cause two subexecutions to be created (as two successor states

in the ARG), one where the revisitable read is not modified and that thread remains blocked (indefinitely, therefore this branch of the ARG is abandoned in this example); and another, where said read takes the value from the new write. This shows up in the abstract execution graph as an *rf*-edge. Of course, the two store operations are also atomically together.

Next, to finish execution on the writer (left) thread, the atomic block is closed and the thread exits. This leaves us with no choice, but to go back to the initially explored (reader) thread. This thread is now unblocked and can continue execution with the read from x. At this point however, the read operation can take a value from two possible locations: either from the initial write, or the write on the other thread. Furthermore, a successor state must also block, as a subsequent store to x should also have a revisitable read waiting for it (even though we know there is no such write in this program). Therefore, three successor states are created once again:

1. The first read has read $a + b$, the second read has read 0
2. The first read has read $a + b$, the second read has read a
3. The first read has read $a + b$, the second read has blocked (dead-end)

The 3$^{\text{rd}}$ state of this second line-up has thus been eliminated from further exploration. The 2$^{\text{nd}}$ state has some potential to be unsafe: i is $a + b$, and j is a – however, as both a and b are non-negative, i cannot be smaller than j.

Furthermore, reaching the 1$^{\text{st}}$ state, the abstraction algorithm evaluates the feasibility of the memory model (it did this in every step, but only now is it actually important). It evaluates every monotonically increasing assertion in the memory model[1] for possible violations, and in this case, it finds that the current state violates atomicity – it should not be possible to read one part of the atomic block in another atomic block, while the other part is left as the initial value. Therefore, this state is designated as $\bot$ and no further expansion is possible.

Note that it is possible to define non-monotonically increasing assertions. In this case, the algorithm will decide to track *all* global variables from the start – even though a more elegant solution is definitely possible, due to the lack of an example for a sane memory model that includes such a construct, I abandoned the further optimization of this case.

Going back to the 2$^{\text{nd}}$ state of the former line-up, the algorithm will deduce either safety or infeasibility for every branch of the ARG – therefore proving the safety of the input program. The entire ARG can be seen in Figure 4.3: two of the branches are consistent but safe; two branches blocked indefinitely; and two branches became inconsistent with the memory model.

4.3 Limitations on Genericity

The algorithm tries to make as few assumptions as possible about the memory model, to truly stay generic. One assumption however has to be to forbid out-of-thin-air (OOTA) executions [19]. Consider the following two-threaded program (x and y are global variables):

```
int r1 = x;          int r2 = y;
y = r1;              x = r2;
```

[1] Not containing set difference

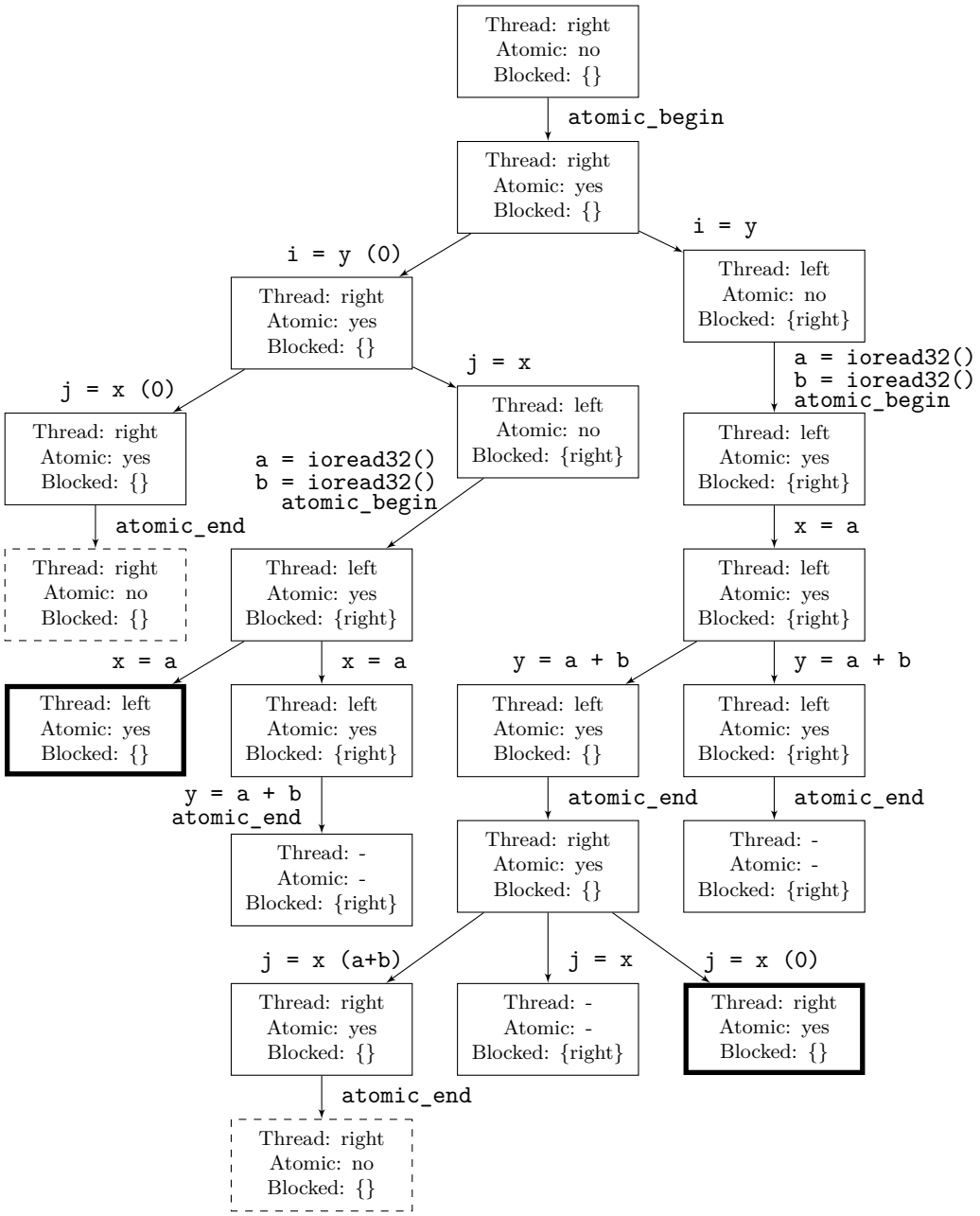


Figure 4.3: Final and safe ARG of the motivational example. Dashed leaves are safe, bold leaves are inconsistent with the memory model.

In this example, the outcome $r1 = r2 = 42$ might be surprising, mainly because the value 42 did not exist anywhere in the program source. However, assuming relaxed accesses and the C programming language semantics [33], this outcome (along with any other value) is allowed by the memory model, as there is a cycle between the suppliers and consumers of the memory accesses – which can lead to a self-inductive execution generation. As this behavior is next to impossible to implement in hardware, and therefore this issue is purely theoretical, forbidding it does not hurt the applicability of the algorithm.

Such interactions are forbidden in the algorithm, as therein a read will block the expansion of its executing thread in at least one instance until all same-location stores are discovered. This means that a *po*-later write will not be able to supply a value to the read; and furthermore, if multiple threads are blocked simultaneously, writes *po*-after either of them will be disregarded in the read's expansion process. In the example above, this latter case is demonstrated: as no previous values are known other than the initial implicit 0, there will be only one full exploration ($r1 = r2 = 0$), and in any other instance, the threads will not unblock, and therefore the writes are not even discovered.

4.4 Formalizing the Approach

To formalize the algorithm introduced above, I will define the 7 custom components in the CEGAR loop (see Figure 4.1), starting with the components related to the ARG. In this section, all descriptions are as technology-agnostic as possible, to provide an adaptable solution rather than an implementation-specific one.

4.4.1 Abstract States for Declarative Analysis

The abstract state is responsible for holding information on the current state of the algorithm, as well as the necessary data structures for the *expr* mapping of the underlying domain. In the case of this algorithm the state vector is a tuple $X = (P, p_c, E, S)$, where:

- P: Set of existing processes (in the form of a tuple $P = (id, loc, lastNode, amo, blocked)$)
  - id: ID of the process
  - loc: CFA location inside the process
  - $lastNode$: Last memory access on the given process
  - amo: A node in E unique to the current atomic block when the process is atomic, or missing otherwise
  - $blocked$: True if the process is blocked due to an unsatisfied read-dependency, false otherwise
- p_c: The currently explored process
- E: The abstract execution graph of existing nodes
- S: The abstract mathematical state (dependent of the domain and the mathematical precision)

The main tool of CEGAR to combat state space explosion is the use of partial orders, which discover covering states in the ARG and stop the expansion of the covered state. For this, a $\sqsubseteq$ function is available for the given domain, which has to be mapped to

the abstract states as well. For example, in the much simpler case of a single-threaded CFA, the partial order would order states whose mathematical states are partially ordered, and whose locations are the same. In the multi-threaded declarative case, this has to be adapted to the following:

Definition 4. *Two abstract states X_1, X_2 are partially ordered by $\sqsubseteq$, if:*

- $S(X_1) \sqsubseteq S(X_2)$, *and*
- $\forall(p_1 \in P(X_1))\exists(p_2 \in P(X_2)).(loc(p_1), blocked(p_1)) = (loc(p_2), blocked(p_2))$, *and*
- $\forall(p_2 \in P(X_2))\exists(p_1 \in P(X_1)).(loc(p_1), blocked(p_1)) = (loc(p_2), blocked(p_2))$, *and*
- $(loc(p_c(X_1)), blocked(p_c(X_1))) = (loc(p_c(X_2)), blocked(p_c(X_2)))$, *and*
- $E(X_1) \cong E(X_2)$. ▪

Finally, a precision is necessary for the ARG building algorithms. In the case of this algorithm, this can be defined as a tuple $Prec = (V, P)$, where:

- V: Set of tracked global variables
- P: Mathematical precision stemming from the underlying domain

Even though the motivating example above suggested that only those global variables are tracked which appear in the mathematical precision in the form of either a tracked variable or part of a predicate; this is not necessarily the case – the algorithm is well-equipped to deal with other forms of precision as well, e.g. tracking all global variables from the beginning. The only constraint is that if a variable appears in the mathematical precision, the algorithm *must* track it.

4.4.2 Building the ARG

To build an ARG from an input model, three components are necessary (as shown in Figure 4.1): the initialization function, the transfer function and the transition supplier (in the form of an LTS).

The initialization function takes a precision and an XCFA as its argument, and returns a set of initial states. As in software verification the initial state is usually modelled to be deterministic, we know that it must be a single-element set, whose sole element is defined as the following state $X_0 = (P, p_c, E, S)$:

- P: $\{p_c\}$
- p_c: $(0, InitLoc(XCFA), Start_0, \emptyset, false)$
- E: All initial writes *po*-before the $Start_0$ node
- S: $\top$

Note that only a single element is added to the set. This is a deliberate choice to only allow *dynamic* XCFAs, which makes it easier to define the algorithm. If support for static XCFAs is also required, it can always be modelled as a dynamic XCFA that starts off by forking into the separate processes.

After the initial state has been added as the root of the ARG, the ARG building algorithm queries the possible transitions from the state using the provided LTS component. This takes a state as argument, and provides a set of transitions, which are totally ordered sequences of actions. Such actions might be:

- `ContextChange(pid: int)`: change the current process to the one with *pid*
- `Advance(loc: XCFA Location)`: change the location of the current process to *loc*
- `AtomicBegin()`: enter an atomic block (no-op when the context is already atomic)
- `AtomicEnd()`: exit an atomic block (no-op when the context is not atomic)
- `ThreadStart(key: Var, f: XCFA Function)`: start a new process with entry point f, and associate the new *pid* with *key*
- `ThreadJoin(key: Var)`: await the exit of the process associated with *key* (no-op when no such process exists)
- `Store(w: Store, R: Set<Load>)`: start a *store* operation from *write* to $\forall r \in R$
- `Load(r: Load, w: Opt<Store>)`: start a *load* operation from w to r when w exists, or block the current process otherwise
- `Fence()`: add a *fence* node to the execution graph of the current process
- `Statement(stmts: List<Stmt>)`: map the mathematical state to a new state according to the statements in *stmts* and the current precision

For example, a transition might be a sequence of a `ContextChange`, `Statement` and `Advance` actions to change the current process, and execute an edge of the XCFA.

For a given state, the LTS will return one of the following (preferential choice, i.e. a later choice is only evaluated if no previous steps returned a valid answer):

1. A non-empty set of transitions originating from the current location of the current process, or
2. A non-empty set of transitions originating from the current location of a non-blocked process, or
3. An empty set

For the first two options, the algorithm is similar:

1. Collect the outgoing edges from the current location of the given process: E_{out}
2. For every $e \in E_{out}$:
   (a) Collect the pure statements (CFA-operations) from e: St
   (b) Create a list of actions, containing two elements: [`Statement(`St`)`, `Advance(Target(`e`))`]
   (c) Add any `AtomicBegin`, `AtomicEnd`, `ThreadStart`, `ThreadJoin` and `Fence` operations as actions to the list
   (d) If a *Read* operation is found on the edge, create as many copies of the current list of actions as many *Write* operations the previous state had (over the same variable), and add a corresponding `Load` action to each list. One of the lists will have a `Load` action with no supplying store. Propagate all lists to the following step of the algorithm.
   (e) If a *Write* operation is found on the edge, create as many copes of the current list of actions as many subsets the corresponding blocking *Read* operations had in the previous state. Add a corresponding `Store` action to each list, with the subset receiving the value of the store. Propagate all lists to the following step of the algorithm.

3. If the process is different from the current process of the previous state, add a `ContextChange` action to the beginning of each list.

4. If at least one list exists, return all lists.

So far, an initial state was successfully created, and we know which transitions the program might take from an abstract state. The only remaining part is the mapping of abstract states to state-transition pairs, i.e. the transfer function.

The transfer function is *modular*, in the sense that each action from the list above corresponds to an independent transfer subfunction. These modify the abstract state the following ways (note the `mathTransFunc` function, which is provided by the underlying domain)[2]

- `ContextChange(pid)`:
  $P' := P \setminus \{P(pid(p_c))\} \cup \{p_c\}$,
  $p'_c := P(pid)$
- `Advance(loc)`:
  $p'_c := (id(p_c), loc, lastNode(p_c), amo(p_c), blocked(p_c))$
- `AtomicBegin()`:
  $amo' := amo(p_c)?amo(p_c) : newAmo$,
  $p'_c := (id(p_c), loc(p_c), lastNode(p_c), amo', blocked(p_c))$
- `AtomicEnd()`:
  $p'_c := (id(p_c), loc(p_c), lastNode(p_c), \emptyset, blocked(p_c))$
- `ThreadStart(key, f)`:
  $P' := P \cup \{(|P|, InitLoc(f), Start_{|P|}, \emptyset, false)\}$,
  $E' := E \cup \{\text{po}(lastNode(p_c), Start_{|P|})\}$
- `ThreadJoin(key: Var)`:
  $p'_c := (id(p_c), loc(p_c), newLastNode, amo(p_c), blocked(p_c))$,
  $E' := E \cup \{\text{po}(lastNode(p_c), lastNode(p'_c)), \text{po}(End_{|P|}, lastNode(p'_c))\}$
- `Store(w: Store, R: Set<Load>)`:
  $p'_c := (id(p_c), loc(p_c), w, amo(p_c), blocked(p_c))$,
  $E' := E \cup \{\text{po}(lastNode(p_c), w), \forall(r \in R).\text{rf}(w, r), [amo(p_c)?\text{amo}(w, amo(p_c))]\}$
- `Load(r: Load, w: Store)`:
  $p'_c := (id(p_c), loc(p_c), r, amo(p_c), blocked(p_c))$,
  $E' := E \cup \{\text{po}(lastNode(p_c), r), \text{rf}(w, r), [amo(p_c)?\text{amo}(r, amo(p_c))]\}$
- `Load(r: Load)`:
  $p'_c := (id(p_c), loc(p_c), r, amo(p_c), true)$,
  $E' := E \cup \{\text{po}(lastNode(p_c), r), [amo(p_c)?\text{amo}(r, amo(p_c))]\}$
- `Fence()`:
  $p'_c := (id(p_c), loc(p_c), F, amo(p_c), blocked(p_c))$,
  $E' := E \cup \{\text{po}(lastNode(p_c), F), [amo(p_c)?\text{amo}(F, amo(p_c))]\}$
- `Statement(stmts: List<Stmt>)`:
  $S' :=$`mathTransFunc(`S`, expr(`$stmts$`))`

[2]The <op_1> ? <op_2> : <op_3> is the ternary *if-then-else* operator. The binary [<op_1> ? <op_2>] operator means the value is only present if <op_1> is present, and in that case, it is equal to <op_2>.

Furthermore, in the *eager* version of the algorithm any action that modifies the execution graph triggers a consistency check. In the *lazy* version, this is delayed until an error location is reached.

The final remaining component of the CEGAR loop is the precision refinement. This is straightforward: starting from an existing precision (with a set of tracked variables and a mathematical precision) and a new mathematical precision, the tracked variables are updated with each variable in the new precision; and the mathematical precision is entirely changed to new one.

With this, all seven components are well-defined to be used in a CEGAR loop. In Chapter 5 I elaborate on the implementation-specific details of the solution, and then in Chapter 6 I present the characteristics of the implementation.

Chapter 5

Implementation

To aid the evaluation of the algorithm introduced in previous chapters, I developed a proof-of-concept implementation in the CEGAR-centric model checking framework Theta[1] [40, 30]. Furthermore, I implemented the following algorithms as well:

- A naive, interleaving based algorithm
- An interleaving based algorithm utilizing partial-order reduction

These will serve as a baseline in the evaluation, so that the underlying technologies do not interfere with the results (as every algorithm will use the same framework and algorithms). Furthermore, these serve as a general overview on the state of concurrent program verification using abstraction-refinement techniques – the presented algorithms roughly cover the lineup of technologies on the software verification competition SV-COMP 21 [11].

The implementation of these algorithms can be found on the xcfa-pr branch of the base Theta repository[2]. Note however, that the implementation is not stable (hence the feature branch), and a complete rework is likely in the near future. Hence, the state of the branch is not considered part of my submission to the Conference.

5.1 Exploring Interleavings

First, I present the implementation of the interleaving based algorithms. In this case the complexity is not in the CEGAR-specific parts of the verification workflow, as the following structures suffice:

- State: a set of enabled processes (each having an ID, a current location and a flag for atomicity), the current PID, and a mathematical state
- Actions: an edge of the XCFA (having a target location and a list of statements), or a thread change action
- Partial order: if all locations in the processes are the same, and the mathematical states are partially ordered
- Precision: a pure mathematical precision

[1] https://github.com/ftsrg/theta/
[2] https://github.com/ftsrg/theta/tree/xcfa-pr

- LTS: each outgoing edge of each process's location is offered (those on another process are extended with a thread change action), unless one of the processes is atomic, in which case only that process is offered
- Initialization function: The main process is added to the state with its initial location, along with a mathematical state of $\top$
- Transfer function: A thread change action changes the current PID, an XCFA-edge action modifies the mathematical state according to the mathematical transfer function; and the location of the current process is changed to the target location

For the naive algorithm, implementing this list of components is enough. However, it will explore every interleaving of the operations, which (as seen in Section 2.4.1) will produce superfluous nodes in the ARG.

For the partial order reduction algorithm, the implementation can go two ways:

- Make the LTS smarter in the sense that it shall only offer a single interleaving from an equivalence class (in a DPOR-like fashion [26])
- Create a pre-processing pass that works on the XCFA before the verification, which identifies atomically executable blocks of code, and places them on a single edge

I opted to implement the latter (based on the experiences of CPAChecker [12], this is a *good enough* solution). Therefore, the algorithm had to differentiate between thread-local and global operations – which in this case meant that any statement that includes a global variable in the XCFA will become a global statement, and all other operations will be classified *thread-local*. Then, any subsequent thread-local block will be given on a single edge, but each global operation will have its own edge. This resembles a course-grained large-block encoding [15]. Thus, the inner behavior of the algorithm need not be changed.

5.2 Declarative Semantics

As the complexity of the declarative algorithm (as seen in Chapter 4) is more complicated than the interleaving-based techniques, the implementation grew more complex as well. Most notably, the evaluation of the abstract execution graph is a part of the algorithm in almost every step of expanding the ARG, and therefore its performance is one of the most important parts of the implementation. The main source of its complexity comes from its speculative nature: it is not enough to keep track of ground- and derived relations; the *co*- and *rf*-relations have to conform to well-formedness criteria. For example, there must exist a total order over same-location writes in *co* in the execution graphs – even though we add no such relation, and therefore a simple check is not enough. I used four different techniques to handle this problem:

1. A pure Datalog [5] implementation (using the built-in Datalog engine in Theta that I previously developed), iteratively trying to find satisfying *rf*- and *co*-relations
2. An SMT-solver based solution, using functions as relations
3. A SAT-solver based solution (still using an SMT-solver, but only with the theory of boolean logic)

4. A SAT-solver based solution (still using an SMT-solver, but only with the theory of boolean logic), using Datalog as a pre-processing step for monotone increasing relations

Further details of the implementation closely follow the steps of the algorithm described in Chapter 4.

5.3 Common Parts

To verify C programs, Theta uses an ANTLR-based front-end, which maps (possibly multi-threaded) programs to the XCFA formalism. This has the advantage (in comparison with e.g. LLVM-based front-ends) that the number of variables does not grow significantly higher than the number of variables in the source program[3]. However, many aspects of the C programming language are not yet mapped to the elements of XCFAs, such as pointer support other than simple alias-analyses. Even though there is not a definitive list of features this front-end supports, we took the SV-COMP benchmarks[4] as the baseline to aim for – and in its current state, the front-end is capable of parsing more than half of the ReachSafety and ConcurrencySafety categories.

Furthermore, the solvers are also shared among the algorithms. Theta recently received support for SMT-LIB, a cross-tool communication language of SMT-solvers, which enables a wide range of solvers to be used. However, during the early stages of development, I implemented the algorithm to mainly rely on the legacy Z3 solver (as it has native bindings in Theta, and therefore no inter-process communication is necessary).

[3]This is a problem of the SSA-format LLVM uses to store the intermediate representation, which inherently erases information on the variables due to a register-like view of the program.

[4]https://github.com/sosy-lab/sv-benchmarks/tree/master/c

Chapter 6

Evaluation

This chapter compares and contrasts the performance implemented proof-of-concept algorithms introduced in Chapters 4 and 5. Furthermore, it elaborates on the possible configuration options' implications on performance in the declarative algorithm.

6.1 The Benchmark Set

I used a subset of the ConcurrencySafety benchmarks of the SV-COMP benchmark repository[1]. These include atomic blocks, dynamic thread creation, mutual exclusion algorithms, tasks concentrating on data races, and so on. To save on evaluation time (due to the high number of configurations I planned to test), I picked 50 tasks from the set of programs the front-end could parse. I tried to make it as diverse as possible (from the sub-categories of the category), also taking safety into account: overall 25 *true* and 25 *false* tasks got chosen.

Even though this benchmark set might seem too small, it is good enough to demonstrate high-level properties of the algorithms. It was more important to be able to test a high number of configurations, as the main reason of this evaluation is the characterization of the algorithms – a deeper-dive into performance evaluation is a plan for the future.

Furthermore, it might seem unfair to the declarative algorithm that it is pitted against purpose-built sequential algorithms, when it is a more general solution. While this is true, and it will show a bias in the evaluation of the performance metrics, it was important to see if there are tasks where the declarative approach is better than the interleaving-based one. However, due to this bias, the declarative approach is also compared with itself over different memory models: namely, a "*noassert*" memory model featuring no assertions on the possible execution graphs; a "*coherence*" memory model only guaranteeing total coherence (i.e. lack of loops among *rf*- and *po*-edges); and the sequential C memory model (both with and without atomicity). This will hopefully provide a more clear view on the characteristics of the algorithm, even though in these cases the results cannot be evaluated (the SV-Benchmarks repository contains results for the tasks, but only using sequential C semantics) and therefore there might even be false results – even though a manual check was performed on most tasks, it is not a definitive proof the implementation is sound.

[1] https://github.com/sosy-lab/sv-benchmarks/tree/master/c

Exec. Graph Solver \ Domain + Refinement	EXPL MULTI_SEQ	EXPL SEQ_ITP	EXPL BW_BIN_ITP	PRED_CART MULTI_SEQ	PRED_CART SEQ_ITP	PRED_CART BW_BIN_ITP	All
Pure Datalog	30	15	11	20	14	22	**112**
SMT-Functions	5	4	7	8	2	6	**32**
SAT	27	15	9	17	10	21	**99**
SAT + Datalog	29	16	10	21	17	24	**117**

(a) Four different abstract execution graph solvers in 6 CEGAR configurations. Higher score is better.

Memory model \ Domain + Refinement	EXPL MULTI_SEQ	EXPL SEQ_ITP	EXPL BW_BIN_ITP	PRED_CART MULTI_SEQ	PRED_CART SEQ_ITP	PRED_CART BW_BIN_ITP	All
Noassert	34	30	19	43	26	35	**187**
Coherence	35	23	17	39	20	25	**159**
Sequential C	33	19	13	34	18	22	**139**
Sequantial C + atomicity	30	15	11	20	14	22	**112**

(b) Four different memory models in 6 CEGAR configurations. Higher score is better.

For the execution of the benchmarks, I used the benchexec[2] [16] framework. I ran the tests using the BME-NIIF cloud[3], on virtual machines equipped with 8 CPU cores and 16GB of RAM. I allocated 300 seconds (5 minutes) for each test execution.

As the execution timing results are not important on a per-second level, I used the number of successfully verified tasks as the basis of the comparison among the execution runs.

In the benchmark configurations, I typically only use the *domain* and *refinement* options. The rest of the configuration are at sane default values, as follows (see Theta [30] for more information on these options):

- Search: DFS
- PredSplit: Whole
- MaxEnum: 1
- InitPrec: Empty
- PruneStrategy: Lazy
- AbstractionSolver: Z3
- RefinementSolver: Z3

Throughout the benchmark sets, I used every combination of the *EXPL* and *PRED_CART* domains and the *MULTI_SEQ*, *SEQ_ITP* and *BW_BIN_ITP* refinement algorithms.

6.2 Benchmark Results

In this section, I introduce the benchmark sets and their raw results, leaving their interpretation to later sections.

[2]https://github.com/sosy-lab/benchexec
[3]https://niif.cloud.bme.hu/

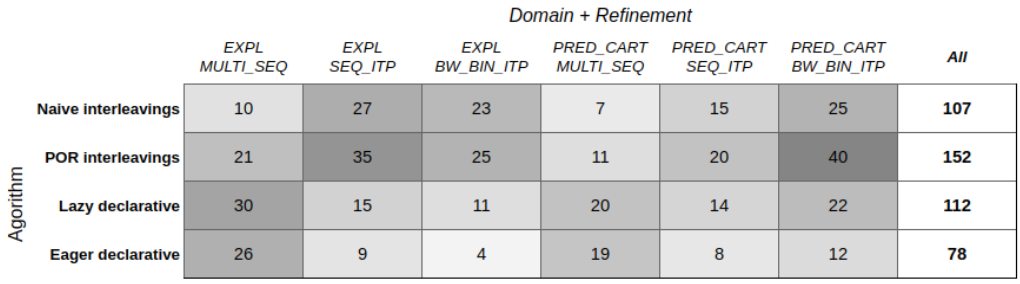


Figure 6.2: Four different algorithms in 6 CEGAR configurations. Higher score is better.

6.2.1 Declarative Verification

To assess the strengths and weaknesses of the different configuration options, I ran two sets of benchmarks:

1. Four different execution graph evaluation solvers (see Section 5.2), using sequential C semantics with atomicity, and lazy evaluation
2. Four different memory models, using pure Datalog as a solver, and lazy evaluation

The results of these benchmark sets can be seen in Figures 6.1a and 6.1b as heatmaps of the number of successfully verified tasks within the timeframe.

6.2.2 Verification of Sequential Programs

To compare the different algorithms, I ran the two interleaving-based algorithms side-by-side a lazy and an eager declarative algorithm. The results can be seen in Figure 6.2.

6.3 Result Evaluation

The following conclusions can be drawn from the results above:

Interleaving-based techniques perform better on sequential C programs. As it can be seen from Figure 6.2, the interleaving-based POR algorithm solved more tasks in almost every configuration than any of the declarative approaches – and even the naive interleaving-based algorithm solved almost the same number of tasks as the better declarative configuration. However, this result is not surprising – as previously mentioned, comparing the two types of algorithm is inherently unfair, as the declarative algorithm can solve more types of problems, and comparing it to a purpose-built algorithm will not highlight its advantages.

Interleaving-based techniques perform akin to single-threaded verification in terms of CEGAR configurations. As shown in [30], single-threaded verification tasks usually prefer EXPL with SEQ_ITP, or PRED_CART with BW_BIN_ITP – the same can be said about the interleaving-based algorithms in this test. Neither of the cross-configurations (EXPL with BW_BIN_ITP or PRED_CART with SEQ_ITP) performed well, and MULTI_SEQ was not a performant choice either.

Declarative techniques prefer MULTI_SEQ rather than single-trace refinement. Contrary to the interleaving-based algorithms, the declarative configurations preferred MULT_SEQ over single-trace refinement. My theory is that the abstraction is much more expensive to start again from the beginning (due to the solver), so a one-time full exploration is more advantageous.

Lazy evaluation performs better than its eager counterpart. Figure 6.2 also shows a clear winner among the declarative algorithms: the lazy configuration solved more than 40% more than the eager algorithm. This clearly contradicts the findings of Dartagnan [24], which uses a check at every step of the algorithm. Hence, further examination is necessary to determine the source of this contradiction.

More relaxed memory models tend to perform better. This is not a surprising conclusion, as a more relaxed memory model has fewer assertions and therefore the solver generally returns a result quicker.

Declarative algorithm performs better on relaxed memory models than interleaving-based techniques on sequential programs. Contrary to the previous conclusion, this is surprising: given the same set of programs, a declarative approach could verify more in a given amount of time than a purpose-built algorithm with a more strict memory model, even though the number of traces grows with the relaxation of the memory model (i.e. more outcomes are explored with a less strict memory model). However, there has been similar published results before: it has been shown, that even in a general case, this finding holds [7].

SMT-function based solver is overwhelmingly slow. Even though at first glance it could be evident that it is better to leave the SAT-projection of the problem to the SMT-solver, these experiments showed that it is much more beneficial to create the SAT-problem by hand. This is however well justified by the fact that SMT functions assume a possibly infinite domain, while the problems in this algorithm are always finite – therefore we do not need the rich expressivity of SMT-functions for our purposes.

Pure Datalog and SAT+Datalog options dominate the other solvers. As discussed under the previous conclusion, it is beneficial to prepare the SAT-problem equivalent to the execution graph by hand. Furthermore, it is even more beneficial to help out the SAT-solver by supplying easily calculable values (e.g. using a bottom-up Datalog engine) instead of raw facts and relations.

6.4 Summary

To summarize the report, in Chapter 2, I introduced some concepts the rest of the report relied upon. In Chapter 3, I presented the related tools and algorithms. In Chapter 4, I introduced the main contribution of my work, the algorithm capable of verifying unbounded, weakly-ordered concurrent programs. In Chapter 5, I presented the proof-of-concept implementation of the algorithm. Finally, in Chapter 6, I presented the performance comparison of the algorithms above as well as the performance implications of the configuration options.

Even though the performance metrics showed that the declarative approach is inferior to the interleaving-based techniques when it comes to sequential C programs, I still consider the presented work a success. It was never a realistic goal to provide a solution that is more generic and also performs better under the same conditions. However, having showed that the presented approach works and can verify actual programs under the semantics of almost any memory model is a success in itself – and the fact that it outperformed the best tested sequential algorithm with a relaxed memory model shows that it has potential to be used in places where sequentality is not provided.

6.5 Future Work

Even though the presented theory works as a proof-of-concept implementation, there is still many aspects of the solution I would like to dedicate more time to. These are, among others and in no particular order, the following:

- A formal proof that the algorithm is *sound* and *optimal*, i.e. no particular execution graph is explored twice
- A deeper evaluation of the declarative algorithm in terms of program characteristics – which features are advantageous for the algorithm and which hinder verification
- A more robust implementation of the algorithms that could be used to verify actual embedded software – currently, the main bottleneck is the front-end, but other parts of the algorithms would have to be modified as well
- A front-end for the CAT language, as currently, the memory models can only be specified in Java
- A test suite of litmus tests that would help verify the implemented algorithms
- A test suite of memory models

The motivation for this work is the rapid progression of safety-critical systems towards the use of multi-core hardware. I strongly believe that a technique similar to mine could be used to bridge the gap between the field of safety-critical development and formal software verification, and in the future, I hope to continue working on this approach to guide it towards this goal.

Bibliography

[1] TriCore TC1.6.2 core architecture manual - Instruction set (Volume 2 of 2). URL https://www.infineon.com/dgdl/Infineon-AURIX_TC3xx_Architecture_vol1-UserManual-v01_00-EN.pdf?fileId=5546d46276fb756a01771bc4c2e33bdd.

[2] Parosh Abdulla, Stavros Aronis, Bengt Jonsson, and Konstantinos Sagonas. Optimal dynamic partial order reduction. *SIGPLAN Not.*, 49(1):373–384, January 2014. ISSN 0362-1340. DOI: 10.1145/2578855.2535845. URL https://doi.org/10.1145/2578855.2535845.

[3] Parosh Aziz Abdulla, Stavros Aronis, Mohamed Faouzi Atig, Bengt Jonsson, Carl Leonardsson, and Konstantinos Sagonas. Stateless model checking for TSO and PSO. In Christel Baier and Cesare Tinelli, editors, *Tools and Algorithms for the Construction and Analysis of Systems - 21st International Conference, TACAS 2015, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2015, London, UK, April 11-18, 2015. Proceedings*, volume 9035 of *Lecture Notes in Computer Science*, pages 353–367. Springer, 2015. DOI: 10.1007/978-3-662-46681-0_28. URL https://doi.org/10.1007/978-3-662-46681-0_28.

[4] Parosh Aziz Abdulla, Mohamed Faouzi Atig, Bengt Jonsson, and Carl Leonardsson. Stateless model checking for POWER. In Swarat Chaudhuri and Azadeh Farzan, editors, *Computer Aided Verification - 28th International Conference, CAV 2016, Toronto, ON, Canada, July 17-23, 2016, Proceedings, Part II*, volume 9780 of *Lecture Notes in Computer Science*, pages 134–156. Springer, 2016. DOI: 10.1007/978-3-319-41540-6_8. URL https://doi.org/10.1007/978-3-319-41540-6_8.

[5] Serge Abiteboul, Richard Hull, and Victor Vianu. *Foundations of Databases*. Addison-Wesley, 1995. ISBN 0-201-53771-0. URL http://webdam.inria.fr/Alice/.

[6] Jade Alglave, Daniel Kroening, and Michael Tautschnig. Partial orders for efficient bounded model checking of concurrent software. In Natasha Sharygina and Helmut Veith, editors, *Computer Aided Verification - 25th International Conference, CAV 2013, Saint Petersburg, Russia, July 13-19, 2013. Proceedings*, volume 8044 of *Lecture Notes in Computer Science*, pages 141–157. Springer, 2013. DOI: 10.1007/978-3-642-39799-8_9. URL https://doi.org/10.1007/978-3-642-39799-8_9.

[7] Jade Alglave, Daniel Kroening, and Michael Tautschnig. Partial orders for efficient bounded model checking of concurrent software. In Natasha Sharygina and Helmut Veith, editors, *Computer Aided Verification*, pages 141–157, Berlin, Heidelberg, 2013. Springer Berlin Heidelberg. ISBN 978-3-642-39799-8.

[8] Jade Alglave, Luc Maranget, and Michael Tautschnig. Herding cats: Modelling, simulation, testing, and data mining for weak memory. *ACM Trans. Program. Lang. Syst.*, 36(2):7:1–7:74, 2014. DOI: 10.1145/2627752. URL https://doi.org/10.1145/2627752.

[9] Jade Alglave, Patrick Cousot, and Luc Maranget. Syntax and semantics of the weak consistency model specification language cat. *CoRR*, abs/1608.07531, 2016. URL http://arxiv.org/abs/1608.07531.

[10] Mark Batty, Scott Owens, Susmit Sarkar, Peter Sewell, and Tjark Weber. Mathematizing C++ concurrency. In Thomas Ball and Mooly Sagiv, editors, *Proceedings of the 38th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2011, Austin, TX, USA, January 26-28, 2011*, pages 55–66. ACM, 2011. DOI: 10.1145/1926385.1926394. URL https://doi.org/10.1145/1926385.1926394.

[11] Dirk Beyer. Software Verification: 10th Comparative Evaluation (SV-COMP 2021). In Jan Friso Groote and Kim Guldstrand Larsen, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, pages 401–422, Cham, 2021. Springer International Publishing. ISBN 978-3-030-72013-1.

[12] Dirk Beyer and Karlheinz Friedberger. A light-weight approach for verifying multi-threaded programs with CPAchecker. *Electronic Proceedings in Theoretical Computer Science*, 233:61–71, 2016. DOI: 10.4204/eptcs.233.6.

[13] Dirk Beyer, Thomas A. Henzinger, Ranjit Jhala, and Rupak Majumdar. The software model checker Blast. *International Journal on Software Tools for Technology Transfer*, 9(5-6):505–525, September 2007. DOI: 10.1007/s10009-007-0044-z. URL https://doi.org/10.1007/s10009-007-0044-z.

[14] Dirk Beyer, Thomas A. Henzinger, and Grégory Théoduloz. Configurable software verification: Concretizing the convergence of model checking and program analysis. *Computer Aided Verification*, page 504–518, 2007. DOI: 10.1007/978-3-540-73368-3_51.

[15] Dirk Beyer, Alessandro Cimatti, Alberto Griggio, M. Erkan Keremoglu, Simon Fraser Univers, and Roberto Sebastiani. Software model checking via large-block encoding. *2009 Formal Methods in Computer-Aided Design*, 2009. DOI: 10.1109/fmcad.2009.5351147.

[16] Dirk Beyer, Stefan Löwe, and Philipp Wendler. Reliable benchmarking: requirements and solutions. *Springer Science and Business Media LLC*, 21(1):1–29, November 2017. DOI: 10.1007/s10009-017-0469-y. URL https://doi.org/10.1007/s10009-017-0469-y.

[17] Armin Biere, Alessandro Cimatti, Edmund M. Clarke, Ofer Strichman, and Yunshan Zhu. Bounded model checking. *Advances in Computers*, page 117–148, 2003. DOI: 10.1016/s0065-2458(03)58003-2.

[18] Paul E. Black, Paul Ammann, and Wei Ding. *Model checkers in software testing.* U.S. Dept. of Commerce, Technology Administration, National Institute of Standards and Technology, 2002.

[19] Hans-J. Boehm and Brian Demsky. Outlawing ghosts: avoiding out-of-thin-air results. *Proceedings of the workshop on Memory Systems Performance and Correctness*, 2014. DOI: 10.1145/2618128.2618134.

[20] James Bornholt and Emina Torlak. Synthesizing memory models from framework sketches and litmus tests. In Albert Cohen and Martin T. Vechev, editors, *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2017, Barcelona, Spain, June 18-23, 2017*, pages 467–481. ACM, 2017. DOI: 10.1145/3062341.3062353. URL https://doi.org/10.1145/3062341.3062353.

[21] Edmund Clarke, Orna Grumberg, Somesh Jha, Yuan Lu, and Helmut Veith. Counterexample-guided abstraction refinement for symbolic model checking. *Journal of the ACM*, 50(5):752–794, September 2003. DOI: 10.1145/876638.876643. URL https://doi.org/10.1145/876638.876643.

[22] Edmund M. Clarke, Orna Grumberg, and Doron Peled. *Model checking*. MIT Press, 1999.

[23] Edmund M. Clarke, William Klieber, Miloš Nováček, and Paolo Zuliani. Model checking and the state explosion problem. *Lecture Notes in Computer Science*, page 1–30, 2012. DOI: 10.1007/978-3-642-35746-6_1.

[24] Hernán Ponce de León, Florian Furbach, Keijo Heljanko, and Roland Meyer. BMC with memory models as modules. In Nikolaj Bjørner and Arie Gurfinkel, editors, *2018 Formal Methods in Computer Aided Design, FMCAD 2018, Austin, TX, USA, October 30 - November 2, 2018*, pages 1–9. IEEE, 2018. DOI: 10.23919/FMCAD.2018.8603021. URL https://doi.org/10.23919/FMCAD.2018.8603021.

[25] Alastair F. Donaldson, Leopold Haller, Daniel Kroening, and Philipp Rümmer. Software verification using k-induction. *Static Analysis*, page 351–368, 2011. DOI: 10.1007/978-3-642-23702-7_26.

[26] Cormac Flanagan and Patrice Godefroid. Dynamic partial-order reduction for model checking software. In *Proceedings of the 32nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '05, page 110–121, New York, NY, USA, 2005. Association for Computing Machinery. ISBN 158113830X. DOI: 10.1145/1040305.1040315. URL https://doi.org/10.1145/1040305.1040315.

[27] Natalia Gavrilenko, Hernán Ponce de León, Florian Furbach, Keijo Heljanko, and Roland Meyer. BMC for weak memory models: Relation analysis for compact SMT encodings. In Isil Dillig and Serdar Tasiran, editors, *Computer Aided Verification - 31st International Conference, CAV 2019, New York City, NY, USA, July 15-18, 2019, Proceedings, Part I*, volume 11561 of *Lecture Notes in Computer Science*, pages 355–365. Springer, 2019. DOI: 10.1007/978-3-030-25540-4_19. URL https://doi.org/10.1007/978-3-030-25540-4_19.

[28] Patrice Godefroid, Jan van Leeuwen, Juris Hartmanis, Gerhard Goos, and Pierre Wolper. *Partial-order methods for the verification of concurrent systems: an approach to the state-explosion problem*, volume 1032. Citeseer, 1996.

[29] Henning Günther, Alfons Laarman, and Georg Weissenbacher. Vienna verification tool: IC3 for parallel software. *Tools and Algorithms for the Construction and Analysis of Systems*, page 954–957, 2016. DOI: 10.1007/978-3-662-49674-9_69.

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

[31] Gerard J. Holzmann. Explicit-state model checking. *Handbook of Model Checking*, page 153–171, 2018. DOI: 10.1007/978-3-319-10575-8_5.

[32] IEC 61508:2010. Functional safety of electrical/electronic/programmable electronic safety-related systems. International standard, International Electrotechnical Commission, April 2010.

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

[34] Michalis Kokologiannakis, Ori Lahav, Konstantinos Sagonas, and Viktor Vafeiadis. Effective stateless model checking for C/C++ concurrency. *Proc. ACM Program. Lang.*, 2(POPL):17:1–17:32, 2018. DOI: 10.1145/3158105. URL https://doi.org/10.1145/3158105.

[35] Ori Lahav, Viktor Vafeiadis, Jeehoon Kang, Chung-Kil Hur, and Derek Dreyer. Repairing sequential consistency in c/c++11. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2017, page 618–632, New York, NY, USA, 2017. Association for Computing Machinery. ISBN 9781450349888. DOI: 10.1145/3062341.3062352. URL https://doi.org/10.1145/3062341.3062352.

[36] Brian Norris and Brian Demsky. A Practical Approach for Model Checking C/C++11 Code. *ACM Trans. Program. Lang. Syst.*, 38(3), May 2016. ISSN 0164-0925. DOI: 10.1145/2806886. URL https://doi.org/10.1145/2806886.

[37] Zvonimir Rakamaric and Michael Emmi. SMACK: decoupling source language details from verifier implementations. In Armin Biere and Roderick Bloem, editors, *Computer Aided Verification - 26th International Conference, CAV 2014, Held as Part of the Vienna Summer of Logic, VSL 2014, Vienna, Austria, July 18-22, 2014. Proceedings*, volume 8559 of *Lecture Notes in Computer Science*, pages 106–113. Springer, 2014. DOI: 10.1007/978-3-319-08867-9_7. URL https://doi.org/10.1007/978-3-319-08867-9_7.

[38] Susmit Sarkar, Peter Sewell, Jade Alglave, Luc Maranget, and Derek Williams. Understanding POWER multiprocessors. In Mary W. Hall and David A. Padua, editors, *Proceedings of the 32nd ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2011, San Jose, CA, USA, June 4-8, 2011*, pages 175–186. ACM, 2011. DOI: 10.1145/1993498.1993520. URL https://doi.org/10.1145/1993498.1993520.

[39] Thomas N. Theis and H.-S. Philip Wong. The End of Moore's Law: A New Beginning for Information Technology. *Computing in Science; Engineering*, 19(2):41–50, 2017. DOI: 10.1109/mcse.2017.29.

[40] Tamás Tóth, Ákos Hajdu, András Vörös, Zoltán Micskei, and István Majzik. Theta: a framework for abstraction refinement-based model checking. In Daryl Stewart and Georg Weissenbacher, editors, *Proceedings of the 17th Conference on Formal Methods in Computer-Aided Design*, pages 176–179, 2017. ISBN 978-0-9835678-7-5. DOI: 10.23919/FMCAD.2017.8102257.

[41] Caroline Trippel, Yatin A. Manerkar, Daniel Lustig, Michael Pellauer, and Margaret Martonosi. Tricheck: Memory model verification at the trisection of software, hardware, and ISA. In Yunji Chen, Olivier Temam, and John Carter, editors, *Proceedings*

of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS 2017, Xi'an, China, April 8-12, 2017, pages 119–133. ACM, 2017. DOI: 10.1145/3037697.3037719. URL https://doi.org/10.1145/3037697.3037719.

[42] Alan Mathison Turing. On computable numbers, with an application to the Entscheidungsproblem. *Journal of Math*, 58:345–363, 1936.