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

Stateless software model checking parameterized with memory consistency models

Scientific Students' Association Report

Author:

Levente Bajczi

Advisor:

Vince Molnár

2020

Contents

Kivonat

A többmagos hardveren futó többszálú programok formális verifikációja sokáig a terület legnagyobb kihívásai közé tartozott. A probléma nehézségét elsősorban a szálak végrehajtásainak tetszőleges átfedése jelenti. Ennek ellenére a többmagos beágyazott processzorok iránti igény egyre hangsúlyosabban megjelenik biztonságkritikus környezetben is, elkerülhetetlenné téve, hogy a problémakörrel foglalkozzunk. A többszálú programok nemdeterminisztikus viselkedése ugyanis nagyban megnehezíti a tesztelésüket, így még fontosabbá válik a formális módszerek használata.

Ezen felül a többmagos processzorok a teljesítmény növelése érdekében sok optimalizációs technikát tartalmaznak – egy ilyen módszer a memóriakezelő utasítások átrendezésének lehetővé tétele. Ennek motivációja az, hogy a processzor az általánosan sokkal lassabb memóriautasítások befejezésére való várakozás közben is hasznos munkát végezhessen. Az átrendezés azonban bizonyos esetekben váratlan viselkedésekhez vezethet a tisztán szekvenciális futáshoz képest. Kevés ellenőrzési módszert adaptáltak ezen viselkedés kezelésére, és ezek többsége is előre meghatározott memóriamodelleket feltételez, melyek testreszabása nem lehetséges. Ez csökkenti a módszerek alkalmazhatóságát, mivel a legtöbb hardver nem teljesen feleltethető meg egy-egy elméleti modellnek (akár szándékosan, akár tervezési hibák miatt).

Ezen dolgozatban egy olyan algoritmust mutatok be, ami bemenetként egy futásidejű hibadetektálásokkal (assert) annotált programot és egy memóriamodellt fogad, kimenetként pedig megadja, hogy az adott memóriamodellt betartó processzoron futtatva elérhető-e hibaállapot a programon belül. Az algoritmus az *állapotmentes modellellenőrzés* megközelítésére épít, és okos állapottér-bejárási stratégiájával lényegesen kisebb memóriahasználatot eredményezhet, mint a hagyományos modellellenőrző algoritmusok. A dolgozatban belátom, hogy – bizonyos feltételeknek megfelelő programok esetén – az algoritmus helyes, és optimális a megvizsgált lefutások tekintetében. Ezen felül néhány ismert architektúrára és programra alkalmazva a teljesítményét is kiértékelem, korszerű szoftvermodellellenőrző eszközökkel összehasonlításban. Munkám eredménye várhatóan hozzájárul majd a többmagos architektúrákon futó többszálú szoftverek kritikus beágyazott rendszerekben való elterjedéséhez, ezzel végső soron jobb teljesítményt és alacsonyabb költségeket hozva az érintett iparágakban.

A dolgozatban ismertetett eredmények a Budapesti Műszaki és Gazdaságtudományi Egyetem Villamosmérnöki és Informatikai Kar Balatonfüredi Hallgatói Kutatócsoport szakmai közössége keretében jöttek létre a régió gazdasági fejlődésének elősegítése érdekében. Az eredmények létrehozása során figyelembe vettük a balatonfüredi központú Rendszertudományi Innovációs Klaszter által megfogalmazott célkitűzéseket, valamint a párhuzamosan megvalósuló EFOP 4.2.1-16-2017-00021 pályázat támogatásával elnyert „BME Balatonfüredi Tudáscentrum” térségfejlesztési terveit.

Az Innovációs és Technológiai Minisztérium ÚNKP-20-1 kódszámú Új Nemzeti Kiválóság Programjának a Nemzeti Kutatási, Fejlesztési és Innovációs Alapból finanszírozott szakmai támogatásával készült.

Abstract

Formal verification of multithreaded software running on multi-core hardware has for long been challenging for anything other than the simplest of programs. The complexity of dealing with the arbitrary interleavings of such a program makes it one of the hardest problems of software verification. However, the industry trend of introducing embedded multi-core processors into the world of safety critical systems makes it unavoidable having to deal with this problem, because formal methods are even more important for concurrent programs due to their inherent nondeterminism, which makes testing unreliable.

Furthermore, multi-core processors offer many types of optimizations to decrease execution time – one such technique is to allow the reordering of memory accesses whenever possible, to avoid having to wait for the typically much slower memory instructions to finish. This may introduce unexpected behavior compared to a purely sequential execution. Few analyses have been adapted to deal with such behavior, and most of them follow predefined memory models hard-coded into their algorithms without providing any facilities to customize it. This hurts the applicability of such techniques, as most types of hardware do not fully conform to theoretical models (either by design, or due to design flaws).

In this work I propose an algorithm that accepts a concurrent program including assertions and a memory model as inputs, and reports whether the program can reach an erroneous state when run on a processor that abides by the given memory model. This algorithm builds on the *stateless model checking* approach, which yields a significantly lower memory usage than other techniques by using a smart exploration strategy to manage the large state space. I show that the algorithm is optimal in terms of explored executions and sound when the program meets certain criteria. Furthermore, I apply the algorithm to several well-known architectures and programs, and evaluate its performance compared to state-of-the-art software model checking tools. The expected impact of this work is to facilitate the correct implementation of concurrent software on multi-core architectures, ultimately leading to better performance and lower costs in embedded systems.

The results presented in this work were established in the framework of the professional community of Balatonfüred Student Research Group of BME-VIK to promote the economic development of the region. During the development of the achievements, we took into consideration the goals set by the Balatonfüred System Science Innovation Cluster and the plans of the "BME Balatonfüred Knowledge Center", supported by EFOP 4.2.1-16-2017-00021.

Supported by the ÚNKP-20-1 New National Excellence Program of the Ministry for Innovation and Technology from the Source of the National Research, Development and Innovation Fund.

Chapter 1

Introduction

Safety critical embedded systems surround us in most aspects of our lives. The software in cars, elevators, airplanes and power plants are expected to virtually *never*[1] fail – a metric not easily achievable through conventional testing methods. Thus, a more rigorous approach is necessary for reasoning about the correctness of software systems: formal software verification.

Formal verification aims to prove correctness (e.g. all safety requirements are respected) in a mathematically precise way [18]. This process is generally very resource intensive, and therefore only the most critical components have been traditionally verified using this approach. However, even such components have been getting more and more complex, necessitating the rapid development of verification techniques.

One phenomenon behind this trend of increasing complexity is the use of multi-core processors in embedded systems to run concurrent programs, which is partly caused by the industry approaching the upper limits of single-core performance yet needing more [30], and partly due to cost saving measures – in modern processor manufacturing processes, a lower clocked but multi-core CPU is often cheaper than a single-core but more powerful one.

However, concurrent semantics allow arbitrary interleaving among the threads of a program, often making its state space of possible behaviors undepictably large. This represents a barrier for verification tools that employ any kind of state enumeration technique, and therefore new methodologies are required to deal with this problem.

To combat the obstacle of state enumeration, several specialized approaches have been developed that aim to decrease the number of executions to be explored – one of the first such algorithms for concurrent programs is *Partial Order Reduction* (POR) [33], which determines whether some executions are equivalent on some abstraction level in their outcomes, and then explores only a representative trace. However, this method also records all explored executions (to determine whether a new candidate execution is equivalent to an already handled one), meaning the memory usage tends to scale poorly.

In conventional software verification, to avoid having to keep a record of each state (and therefore to decrease the memory usage), one might use a *stateless* model checking algorithm. This family of algorithms explore all reachable states of the program without recording them. However, special techniques must be employed to distinguish the reach-

[1] For example, a *SIL 4*-classified component is expected *not* to fail until more than 10 000 years of continuous use

able states based on their equivalence – this has led to algorithms such as *Dynamic Partial Order Reduction* (DPOR) [19].

Furthermore, even if an algorithm can handle concurrent program semantics, there are many aspects to dealing with *multi-core* processors and real concurrency, as opposed to single-cores and apparent concurrency (parallelism). One of these aspects is the *weakened* memory model of such CPUs, which allows the reordering of memory accesses whenever possible, in order to avoid having to wait for the typically much slower memory instructions to finish [20]. However, this introduces unexpected behavior when only *sequential* semantics is taken into account, either by a programmer developing software for multi-core hardware, or by *verification tools themselves*.

However, even if a specific algorithm can handle the *weak* semantics of a given hardware-software co-verification problem, it might still occur that an assumption of the tool does not hold. One such example is the verification of flawed memory architectures, where the design does not follow a predefined template for weak memory – such as the ARM Cortex A9 processor that includes a fault where same-variable reads can be reordered, which goes against the specifications of the ISA [1]. In this case, any tool using standard memory models might fail to report safety violations, thus a *parameterizable* solution is necessary, which can handle any memory model.

At the time of submitting this work, no real solution exists to this problem. I presented this problem in my earlier TDK[2] submission in 2018 [9], where I provided a hardware-centric approach as the solution, which used litmus tests as test cases and conventional model checking algorithms as a reasoning tool, which was however limited by the utilized tools' capabilities. In the paper presented at the *EMSOFT19* conference [10], we have defined the scope of the problem and formalized the problem statement, as well as provided a theoretical solution to the problem. In this work, I propose a *stateless model checking algorithm parameterized with memory consistency models*, which satisfies the criteria imposed by the problem statement. Furthermore, I implement the novel algorithm in the Theta verification framework [31] to show its applicability and scalability compared to similar, but not entirely adaptable model checking tools.

This work is structured as follows: After presenting the theoretical background of the ideas behind the solution (Chapter 2), I introduce the model checking algorithms and tools that are closest in the targeted use-case of the problem above (Chapter 3). Afterwards, I present the proposed algorithm through examples and formal specification (Chapter 4), as well as elaborate on its implementation aspects (Chapter 5). Finally, I evaluate the functional, usability and performance properties of the created tool (Chapter 6).

[2]Students' Scientific Conference/Tudományos Diákköri Konferencia (hu)

Chapter 2

Background

In this chapter I introduce the core concepts of the concurrent software model checking field. Firstly, I establish the basis of the *concurrent software* paradigm I aim to analyze, after which I show the properties of the *multi-core hardware* they can be run on. Finally, I present the foundations of *software verification*, the approach to reason about the correctness of a software product.

2.1 Concurrent Software

At the beginning of the history of digital computing – due to the single processing units of the era's CPUs – software was running sequentially, executing the instructions in the order it was specified in its source. The current state of the program could be given with the *current stack s and the valuation of the variables in the global memory v*. When the first multitasking operating systems appeared, these programs moved into *processes* and *threads*[1] (see Definition 2.1.1), but were still only executing on a single processing unit and therefore were still sequential, even though multiple processes could overlap during their execution, providing a false sense of parallelity. This *apparent concurrency* paired with sequentially ordered processes and threads had been the *programmer's view* for quite a few decades before the first devices capable of *real concurrency* [13] arrived to the general consumer. Up until then, actual parallelity was mostly a niche subject of distributed systems [14], and had not reached the mainstream software developer's world – and even if it did, the sequentiality property was never violated.

Definition 2.1.1: Concurrent programs

A concurrent program is a program containing more than one thread executing in parallel. In this work, it is assumed that a concurrent program $p(T, Q)$ is characterized with the following properties:

1. T: The set of threads in p
2. Q: The globally accessible variables of p

Note that a program has a predetermined amount of threads.

[1] For the sake of simplicity, I use the terms *process* and *thread* interchangeably

However, even in the case of *apparent concurrency*, the number of ways a program with more than a single thread can be executed explodes compared to the only possible execution of a single-threaded program. This is due to the arbitrary interleaving of threads in a system – there are no built-in guarantees concerning the order of execution among them. This also means that the state vector of the program grows, as it is no longer enough to specify the (s, v) tuple – that is valid only for a given thread. The global state can be described with the state vector shown in Definition 2.1.2.

Definition 2.1.2: State vector of a concurrent program

The current execution state of a concurrent program $p(T, Q)$ can be described with the following tuple: (S, V, E), where

1. $S \in (s, v)^{|T|}$: Stacks and valuations of local variables of the individual threads in T
2. V: Valuation of global variables Q
3. E: Enabled threads in T (i.e. *not blocked*)

A *stack* entry contains the following types of information:

- The current location of the execution in the program (like the Program Counter in the hardware)
- The locations to jump back to when a function finishes execution

With the introduction of multi-core processors, the performance limit of a single thread could be overcome by utilizing more processing units. The same multitasking operating systems could be used as before, but with an actual performance benefit to using more threads. However, this also introduced unexpected behavior to programs accessing globally shared memory due to the more advanced memory models of such hardware – the reason for this elevated complexity is explained in detail in Section 4.1.5.

2.2 Multi-Core Hardware

As we are approaching the theoretical limits of the maximum frequency in the current technology's processors [30], it has become evident that we need to concentrate on aspects of the microprocessors other than its single-core performance – such as the number of cores it contains, and their effective cooperation. Theoretically, we can almost double the available performance if we double the number of cores, as they can all work in parallel.

The term *multi-core* can mean that a processor either

- has more cores of the same kind (homogenous multi-processing), or
- has more cores of different kinds (heterogeneous multi-processing) [23].

Furthermore, *multi-cores* might differ in terms of access to caches and main memory, how they are interconnected and what isolation is in place among them, if any. A common property of such systems is that they are on a single chip, i.e. the cores reside on the same silicon wafer. Furthermore, legal restrictions have been placed to ensure that the consumer understands the use of the term *multi-core*: the processor manufacturer company AMD

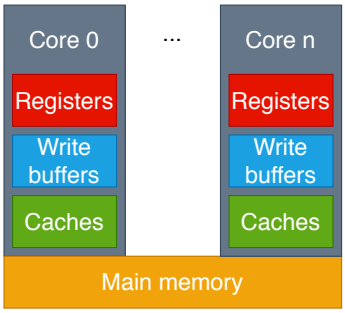


Figure 2.1: Multi-core architecture structure

had erroneously called one of its products an 8-core multicore, when in fact it only had 4 fully-fledged cores, the other 4 being only integer execution cores[2].

To summarize, the term *multi-core processor* can be used in a lot of ways. In the scope of this work, it is assumed to conform to Definition 2.2.1 and a visualization can be seen in Figure 2.1. However, adhering to its criteria is not easily achievable – for example, the cores cannot communicate among each other, but there still needs to be a total ordering of the memory instructions, as only a single core can assume access to the memory at any time. This arbitration is the task of the *memory model* [20].

Definition 2.2.1: Multi-Core Processor

In a *multi-core processor*

1. there is a single integrated circuit containing more than one homogenous and equally capable execution core,
2. each core has access to its own cache hierarchy and local buffers,
3. each core has direct access to the main memory of the system,
4. the cores can only synchronize with each other through the main memory,
5. only one core can assume access to the main memory at a given time.

2.2.1 Memory Model

To solve the problem of allowing only a single core to access the memory at a time, most architectures use a separate memory controller that governs access to the main memory. However, it is far from trivial how a total ordering of memory accesses should be produced – it is the task of the *memory consistency model* (MCM) to establish these rules [20].

Consider the program containing two threads in Figure 2.2, where each writes to a separate global variable[3] and then prints the other's variable. If we try to sequence these instructions assuming *sequential consistency* (SC, see Definition 2.2.2), we get the 6 possible total orders in Figure 2.4. Note that in no case can *both* read instructions execute before *any* of the write instructions finished – therefore, the outcome $a = 0 \wedge b = 0$ is not allowed [20].

[2] https://www.anandtech.com/show/14804/amd-settlement

[3] In all examples the variables $a, b, c, \ldots$ are *local* and $x, y, z, \ldots$ are *global*

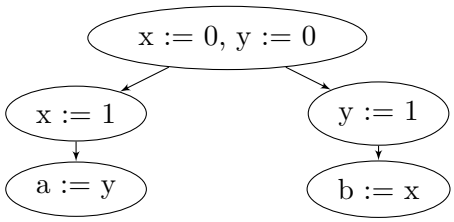


Figure 2.2: Two threads performing a global write and read access

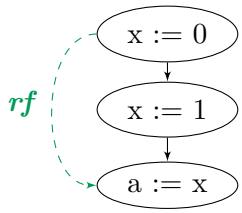
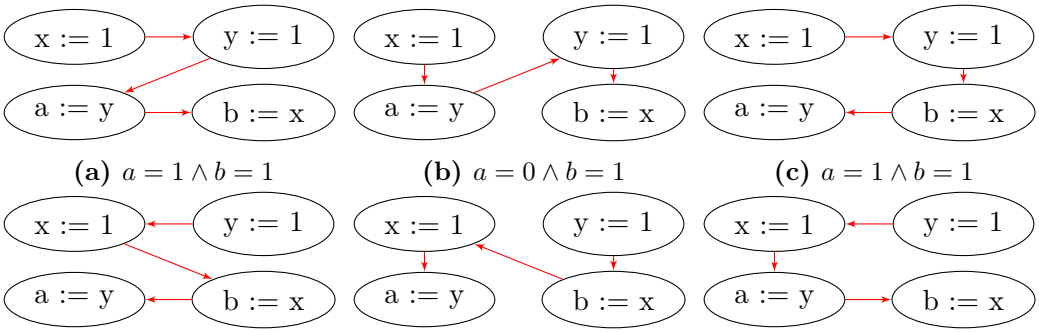


Figure 2.3: a reads 0 instead of 1



(a) $a = 1 \wedge b = 1$ **(b)** $a = 0 \wedge b = 1$ **(c)** $a = 1 \wedge b = 1$

(d) $a = 1 \wedge b = 1$ **(e)** $a = 1 \wedge b = 0$ **(f)** $a = 1 \wedge b = 1$

Figure 2.4: Executions of the program in Figure 2.2 assuming SC

During the execution of the program there are four memory accesses: two writes and two reads. These are very expensive in terms of execution time, because generally the main memory is much slower than the processor that accesses it. To lessen the impact of this phenomenon, most modern multi-core architectures employ a *weakened* memory model, meaning that memory instructions do not need to finish[4] before execution can resume [20].

Definition 2.2.2: Sequential Consistency (SC)

A sequentially consistent memory model ensures the preservation of all happens-before relations in the *program order*, i.e. the sequence of instructions in the source. All threads observe the memory in the same way during execution [20].

However, the performance benefit of this *weakened* memory model does not come without a price – it introduces behavior not expected when assuming sequential consistency. If we allow *any* reordering of instructions, the program in Figure 2.2 can have **24** executions in total (any permutation of the four instructions). This could also mean that if a single thread wrote a variable and then read it back, it could be the old value (before the write), as seen in Figure 2.3. The responsibility of any MCM is to constrain the reordering of instructions, to find a balance between performance and unexpected behavior.

An example of a widely used memory model is the *total store ordering* (TSO, see Definition 2.2.3). This is used (among others) in the x86 and x86_64 architectures, and introduces only a few types of unexpected executions – for example, in the case of the example in Figure 2.2, the previously forbidden outcome $a = 0 \wedge b = 0$ is observable, as it is possible that neither of the write operations actually finished before both of the

[4] In this context, *to finish* means that the written value has propagated to the global memory and visible by all threads, and a read value is fixed in its target local variable [20]

read accesses returned their values – on any of the threads, they were accessing different variables in memory, and therefore could be reordered.

Definition 2.2.3: Total store ordering (TSO)

The *total store ordering* memory model provides the following guarantees:

1. All accesses to the same memory location are totally ordered
2. All write events are totally ordered.

This means that read instructions can be reordered with other read or write instructions, as long they are accessing different parts of the memory [20].

What most architectures aim to achieve is *coherence* [20]. *Coherence means that if there is a certain order to write events appearing in memory, then all observing threads will see these changes in the same order.* Furthermore, it is desired to only allow causal write-to-read events, as reading from a future write is certainly not the intended outcome a programmer could want. With these constraints, the most permissive memory model is called the *weak memory* model (see Definition 2.2.4). This is most famously used in ARM and RISC-V chips.

Definition 2.2.4: Weak memory

A weak memory model only provides the following guarantees:

1. All accesses to the same memory location are totally ordered
2. All reading threads observe the memory changes at the same time

This means that any two memory instructions accessing different parts of the memory can be reordered on a given thread, but their result will be observable by other threads in a single total order [20].

To demonstrate the capabilities of this weakest memory model, take the program in Figure 2.5. There are two threads, one performing two write operations to two distinct memory locations, and the other reads them back in a reverse order. Through intuition (which represents the SC-centric programmer's view), the outcome $a = 1 \wedge b = 0$ should be forbidden, because that would mean that by the time the first read returns, the second write has already been finished, but *after that* the first write still has not appeared in memory. However, in the case of weak memory, this is *allowed* in the execution seen in Figure 2.6 – on any of the two threads, the memory operations were accessing different memory locations and therefore could be reordered.

Note, that in the scope of this work, we are talking about the memory models of *hardware*. Concurrent languages such as C/C++ and Java also have memory models, but they serve more as an upper bound on supported relaxation rather than an actually enforceable ruleset. They aim to ensure that one can reason about the possible behavior of portable code as well, where the target platform (and therefore its memory model) is not fixed.

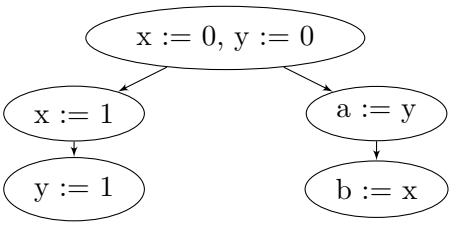


Figure 2.5: Two threads performing two write and two read operations

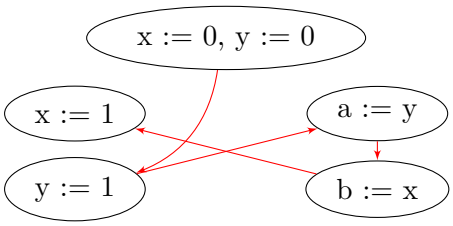


Figure 2.6: One execution yielding $a = 1 \wedge b = 0$

2.2.1.1 Memory Model Specifications

The formal specification of a memory consistency model describes the actual guarantees a specific architecture provides to the programs running on it. The above mentioned memory models (SC, TSO, weak) are *families* of memory models, and their broadly specified guarantees have to be concretized. In this section, I introduce three ways to provide this specification.

Litmus tests Litmus tests are an *example-based* way of providing the specification of the memory model. Litmus tests are small, concurrent programs containing memory accesses and classify outcomes as *forbidden* and *allowed* [20]. This is an *informal* specification, because the set of provided litmus tests can be both incomplete and self-contradicting. However, they provide an easy-to-understand way of communicating the platform specific behaviors to the programmers, and therefore are widely used – Intel for example *only* provides this specification to its memory model [12].

Microarchitectural specification The microarchitectural specification describes the exact way a processor solves the memory ordering problem – and therefore it can be classified as the *imperative* specification of a memory model [32]. It contains the pipeline stages of the microprocessor, and the way instructions are treated in each of them. This is harder to grasp for a programmer due to the complexity of the specification, but it is a *formal* specification, and tools can use this to provide insight into the predictable behavior of a program – however, mostly only by *simulating* it, as reasoning about anything bigger than a few lines of code is infeasible through this approach. What it can be used for, however, is to provide a second view into the memory model – for example, verifying if the specified litmus tests of the ISA are honored by the designed architecture.

Declarative specification A declarative specification provides certain forbidden patterns over candidate executions [6]. The natural language specification of the memory models above (SC, TSO, weak) are closest to this approach, but the specification is a *formal* rephrasing of the mentioned rulesets. It can be used to validate if a candidate execution is allowed by the architecture, and this approach (in comparison to the microarchitectural specification) is scalable: these patterns are validated on execution graphs, rather than after every instruction. In this work, I am using declarative specifications to verify certain properties of concurrent programs.

2.2.1.2 Memory Model Faults

Memory consistency models are complex parts of the architecture, requiring both theoretical planning and corresponding hardware design. This makes it unsurprising that sometimes there are discrepancies between the specification and the actually observable behaviors due to design flaws in the architecture. One such example is the *read-read hazard* found in the ARM Cortex A9 microprocessor [1], which makes it possible that two read instructions accessing the same variable on the same thread are reordered, and therefore observe values in a different sequence than the globally coherent order. Such faults lead to a unique memory model, where not even the most basic assumptions hold [32] – and therefore programs running on them require special procedures to verify their correctness.

2.3 Software Verification

Software verification aims to prove that a software product satisfies its requirements [18]. This is done either by testing these properties through running the program on the target platform (called *dynamic verification*), or by analyzing the program with a purpose-built verification tool without having to run it (called *static verification*). The former is useful for catching bugs in the application, but in most cases cannot provide a proof of correctness – there will always be a corner case the testing workflow leaves out. Meanwhile, static verification techniques can perform the following checks (among others) on a software product:

1. Anti-pattern detection
2. Code conventions verification
3. Software metrics calculation
4. Formal verification

The first three options are built into most development environments to some extent, and developers use it extensively to receive feedback on the quality of the code before ever running it. The fourth option is more complicated: it aims to verify that formally specified criteria are never violated by a program. This is slowly making its way into the above mentioned IDEs as well, and there are tools that use formal verification techniques to prove common requirements for a broad selection of programs (e.g. all programs written in one programming language)[5], but the ultimate goal of formal verification is to check project-specific criteria, specified in a formal language. This way, it can be used to possibly provide a *proof of correctness* or a *violation witness* that formally proves or disproves a specified criterion.

2.3.1 Model Checking

Model checking is an area of formal verification, which uses a finite-state model of a system (in the case of software verification, a program) for checking whether it meets a given set of criteria, to provide feedback on the product's correctness [17]. It can be used to verify both *liveness* (e.g. the lack of a livelock in the system) and *safety* requirements.

[5]For example, *clang-tidy*: https://clang.llvm.org/extra/clang-tidy/

An easy-to-implement approach of model checking is the generation of the state space through enumeration. The state space is the set of all possible states in the model – for example, in the case of a concurrent program, a single state can be described with the state vector in Definition 2.1.2, so the entire state space is the universe over this vector. If we can enumerate all possible states, and can filter out those that are actually reachable, checking the criteria is a simple matter of iterating over this set and proving or disproving the requirements one-by-one. However, it is easy to see that this approach is not feasible for anything other than the simplest of models: if we include a single 32-bit integer in our program, the states required to differentiate its value are 2^{32}. Even if a single state can be stored as simply the value of this integer (so in 4 bytes), the state space would take up more than 17 GB of data. If we included a second integer of the same size, this storage requirement would jump to 147 EB, or $1.47 * 10^{20}$ bytes, not to mention the time required to iterate over all of these states.

When dealing with concurrent behavior, this already fast growing state space further explodes due to the arbitrary interleavings of the different threads. This leads to the need for a smarter approach for dealing with these kinds of problems. Some of these techniques are introduced in Chapter 3.

Chapter 3

Related Work

To date, a number of approaches have been developed for dealing with the problem of concurrent software model checking. These techniques are mostly inadequate for verifying criteria on concurrent software models where the system employs a unique memory model. However, these provide the basis for the proposed algorithm for dealing with this problem and thus their introduction is necessary to fully understand the presented approach.

In this chapter I elaborate on some of these techniques, contrasting their targeted use-cases with that of this work.

3.1 Assuming Sequential Consistency

When the modeled system provides a sequentially consistent execution environment, the problem simplifies greatly: on any of its threads, the instructions are guaranteed to be executing in the order specified in the source, and their effects are visible to all observing threads at the same time. This means, that the only problem to be tackled is the one introduced in Section 4.1.5 – the large number of the possible states. The standard method of dealing with this problem is to employ a *bounded model checking* algorithm [15, 16] which tries to enumerate all violating states withing a given *bound* of the starting state. To avoid visiting the same state multiple times, this method keeps track of all the states and compares any newly visited state with the elements of this set. This is impractical due to the large memory footprint of this approach, and has lead to another technique called *stateless model checking* [21]. The main premise of this method is that it does not store the set of visited states, but rather uses smart state space exploration techniques to avoid redundant exploration therein.

However, even this smarter technique, when employed naively, is impractical due to the large number of interleavings a program might have among its threads. The original solution to this problem is the *partial order reduction* (POR) algorithm [33], which uses

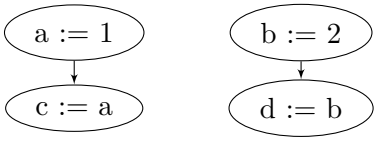


Figure 3.1: All total orders yield $c = 1 \wedge d = 2$ because a, b, c, d are thread-local variables, so differentiating among the states is unnecessary

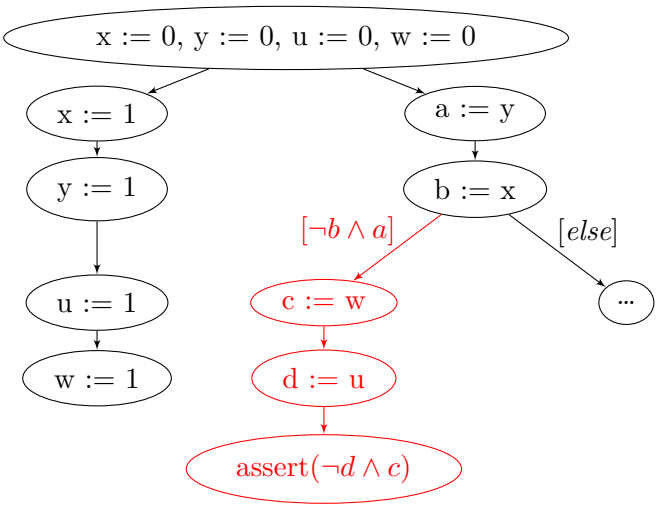


Figure 3.2: The assertion is only reachable when instructions are reordered, and is only violated if other instructions are also reordered

static analyses to discover *thread-local* variables and uses that information to produce a smaller state space, because when a thread only works with local variables, it does not matter whether execution overlaps with another thread doing the same (as seen in Figure 3.1). This approach radically reduced the number of "interesting" states, but was not suitable for large-scale verification due to its reliance on static analyses [19]. To combat this, POR was combined with the idea of *stateless model checking*, resulting in the *dynamic partial order reduction* (DPOR) algorithm [19]. This generated the "interesting" states greedily, when it was time to dissolve a nondeterministic part of the program.

These approaches all assume that the modelled system employs a sequentially consistent memory model, i.e. instructions are never observable out-of-order. As discussed in Section 4.1.5, this is rarely the case due to the aggressive optimization on the processor's part. This means, that while providing stable basis for the following algorithms, they themselves are incapable of handling the proposed problem involving weakly ordered memory.

It is important to mention that by itself, this not only means that the SC-assumption is disadvantageous when verifying the requirements, but rather the whole approach is flawed. Take the program in Figure 3.2 as an example: the problem is not only that the body of the assertion never evaluates to *false* over SC ($w \implies u$ due to program order, and therefore $\neg d \wedge c$ is always *true*), but also that the location containing the assertion is *not even reachable* with that assumption. This means that even if we replaced the checks with smarter ones that could report such violations, the violating state would never be generated by the underlying algorithm, thus a simple adaptation of these algorithms to weak memory is not enough, new approaches are necessary.

3.2 Getting Weaker

When talking about weak memory, it is important to distinguish between memory models of *languages* and *processors*. While there has been substantial work on the modelling of weak memory in processors, most tools that can handle weak memory do so on the abstraction level of the programming language and therefore can assume a rather fixed memory model.

3.2.1 Weakly Ordered Languages

Some algorithms handling weakly ordered languages extend SC algorithms to include previously hidden behaviors in the reachable state space, and some use novel approaches to solve the problem of almost arbitrary reorderings.

An example of the latter is the *CDSChecker* tool [28], which aims to use "*several novel techniques for modeling the relaxed behaviors allowed by the memory model and for minimizing the number of execution behaviors*" to verify concurrent programs using C/C++ atomics. Even though at the time it performed quite well, compared to other tools of the field, it has since been surpassed in both performance and features by other tools.

An algorithm which extends an SC algorithm to correctly and performantly handle weak memory has only recently been published [24]. This approach (implemented in the tool *RCMC*) uses a stateless model checking approach, but extends the rules of dissolving nondeterministic read- and write events to correctly handle the weak memory model of the C/C++ language – or rather, the one created by *Lahav et al.* [26] called *RC11*, which is the fixed version of the official C11 memory model (in the sense that it is prefix-closed). This algorithm has been the basis for the approach I propose in Chapter 4, but in itself it is not suitable for the verification of systems employing any memory consistency model other than that of RC11 – this means that while a C program might be correct, even taking the weakness of the language into account, the executing hardware might still introduce bugs or constrain the set of observable behaviors yielding false positive and negative proofs.

3.2.2 Weakly Ordered Hardware

There had been substantial work in formalizing memory consistency models before [5, 8], but the *cat* language [7] in the tool *herd* [6] revolutionized their specification. Instead of the interleaving semantics, it uses a declarative approach to describe forbidden patterns in the candidate executions of programs and thus constraining the set of executions to that of *observable* executions.

The main idea behind this declarative approach is to keep track of specific relations in the candidate executions, such as:

- ***po***: Relation between immediate successors in the program code
- *ppo*: Subset of ***po*** edges that are respected by the architecture
- ***rf***: Read-From relation between a Write and a Read that gets its value from the Write
- ***fr***: From-Read relation between a Read and a Write that gives its value to the Read

Take the program in Figure 3.2 for example. When annotated with some of these labels (***rf***, ***po***) in Figure 3.3, the nondeterminism disappears and yet the necessary interleavings are never explored, which means that this method discovers equivalence classes among executions *without having to generate their elements* – this is the advantage of the declarative approach.

As an example for the checks performed by this method, let us see the program in Figure 2.5 again. If we annotate it with the labels {***po***, ***rf***, *ppo*} according to *weak* consistency, we get Figure 3.4 – the *ppo* relationship only appears between accesses to the same locations[1].

[1] In the case of TSO, $ppo := \boldsymbol{po} \setminus W * R_{diff} \setminus R * R_{diff}$ and in the case of weak memory, $ppo := \boldsymbol{po} \setminus W * W_{diff} \setminus W * R_{diff} \setminus R * R_{diff}$, where $A * B_{diff}$ means A and B access different variables

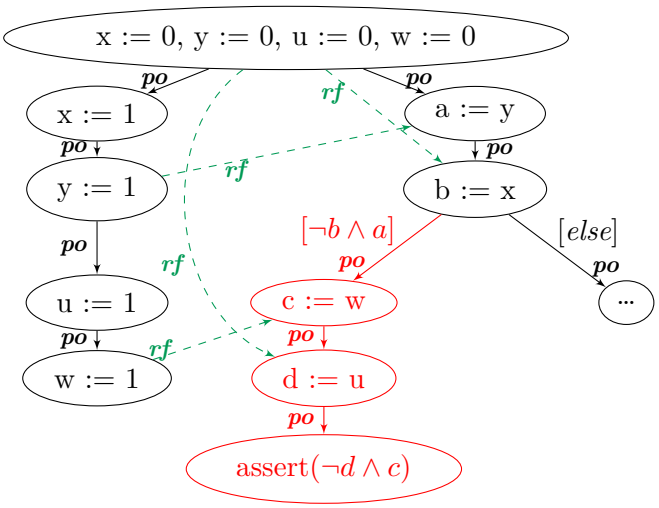


Figure 3.3: With this annotation, it is evident that the assertion is violated.

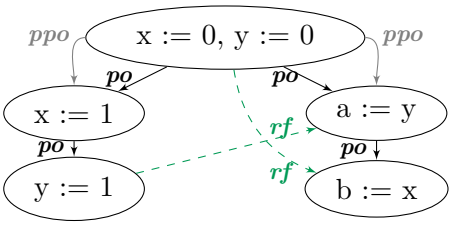


Figure 3.4: Figure 2.5, annotated over a weak memory model

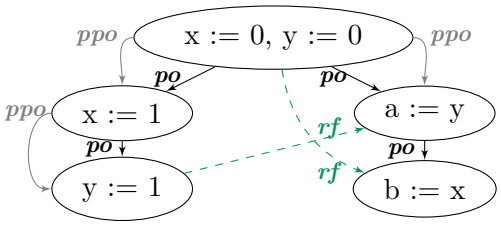


Figure 3.5: Figure 2.5, annotated over a TSO memory model

Take *coherence* as an example, which in this context can be defined the following way: *if a thread has read from another thread's write event, it cannot later read from a* ppo*-previous write event.* Because the $y := 1$ node is not preceded by any other write events in *ppo*, the execution is legal – but if we annotate it as if it was over TSO, we get Figure 3.5, where this property is *violated* and thus we need to discard this execution due to violating coherency.

The main problem with this approach implemented in herd is that it still does not scale well – tests on the scalability of tools handling weak memory done by *Kokologiannakis et al.* [24] indicate that among the other algorithms, *herd* performs the worst because it aims to generate *all* candidate execution graphs and only filters this set to get the *legal* set of executions thereafter.

A promising implementation of a stateless model checking algorithm can be seen in the tool *Nidhugg* [2] – it includes purpose-built algorithms for several well-known architectures (SC, PSO, TSO, POWER), based on a DPOR algorithm for the former three [4], and a combined operational and declarative methodology for POWER [3]. However, as shown by *Kokologiannakis et al.* [24], Nidhugg performs significantly worse than the presented RCMC tool and it cannot handle unique memory models either.

3.3 Handling Faults

A novel idea in the field of concurrent model checking is the consideration of using faulty hardware and still being able to verify whether safety criteria are violated when running

a program on top of it [10]. This can be useful in a number of ways: for example, having integrated a large amount of microprocessors into a line of products only to later realize that a newly found fault breaks the memory ordering assumptions, it would require a recall without the option to see if a simple software update could solve the issue. This saves not only costs, but the reputation of the company as well, because recalls generally hurt the image of the manufacturer.

To date, there has not been published any solution to this problem other than the theoretical algorithm in the paper I authored [10]. However, there has been work on the *prevention* side of the hardware-software co-verification workflow: the *Check* family of tools [27, 32] aim to ensure that the ISA specification corresponds to the architectural implementation by verifying whether litmus tests with *forbidden* or *allowed* outcomes are indeed *observable* on the architecture – and based on the result, the implementation can be classified as *too strict, too weak* or *correct*. When ran against the RISC-V specification, more than a hundred violations were uncovered that aided the correction of the architecture [32]. However, the utilized approach is not scalable, as these tools use an architectural specification based on the behavior of the hardware to check the programs against – this means that after each instruction, the entire pipeline of the processor has to be simulated. Furthermore, the relations are defined between the different pipeline stages rather than the instructions. This makes it unsuitable for anything other than the small litmus tests they aim to verify.

To summarize the introduced approaches, I placed the tools and algorithms claiming to handle concurrent programs on the Venn-diagram in Figure 3.6. Note, that there are no contenders that provide all three aspects: handling weak memory **and** unique memory models, while staying scalable. The proposed algorithm of this work aims to fill this gap and bring concurrent software verification closer to being applicable to any execution environment.

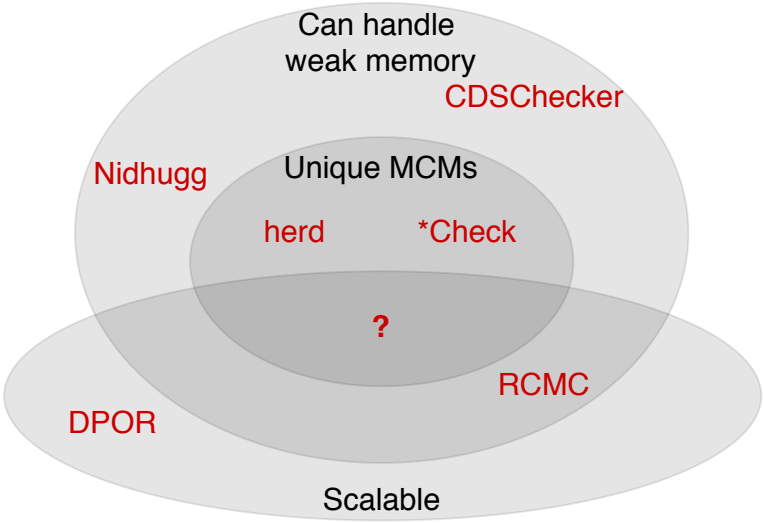


Figure 3.6: Venn-diagram of the related tools and techniques

Chapter 4

Algorithm

In this chapter I describe the stateless model checking algorithm that can be parameterized with memory consistency models to provide a tool for verifying weakly ordered architectures. I base the method on the algorithm implemented in *RCMC* [24], but modify it in its roots to be capable of handling the customizable memory models. For the memory models, I use a declarative specification based on a simplified version of the *cat* language [7]. This chapter is structured as follows: in Section 4.1, I show how the algorithm manipulates execution graphs (introduced in Section 3.2.2) to generate all possible execution graphs through informal descriptions and examples. After that, I formalize the algorithm for generating these execution graphs in Section 4.2. Next, I elaborate on the *filtration* possibilities to get the legal executions in Section 4.3 and finally I provide a proof of soundness and optimality in Section 4.4.

The implementation specific parts of this algorithm can be found in Chapter 5.

4.1 Execution Graphs

Execution graphs are directed, edge- and vertex-labelled graphs. The vertices' labels contain information on the type of memory instruction they represent:

- W(g, v): Write to g with literal value v.
- R(g): Read from g
- F: fence

The edges are labelled with relations akin to the ones defined in Section 3.2.2. Namely:

- ***po***: Relation between immediate successors in the program code
- ***rf***: Read-From relation between a Write and a Read that gets its value from the Write
- ***mo***: Modification order (total order of Write events to a specific location)

Any other relation (such as *ppo*) can easily be constructed during filtration, as seen in Section 4.3.

Furthermore, we maintain a set of read nodes called *Revisitables* ($\mathbb{R}$), the importance of which will be explained in further sections.

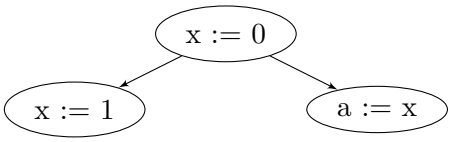


Figure 4.1: Two threads performing a write and a read operation

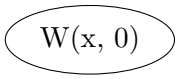
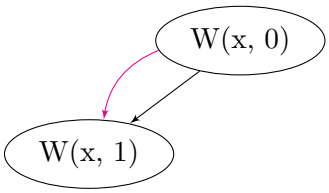
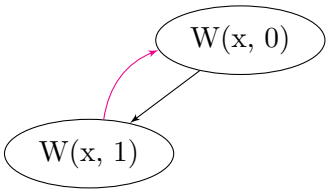


Figure 4.2: Execution graph containing the initial write



(a) New write after initial



(b) New write before initial

Figure 4.3: Execution graphs after inserting the first write

It is important to note that an execution graph only contains nodes related to globally accessible variables. Local and thread-local variables are not handled through this algorithm, as they generally behave predictably and deterministically.

Through the following sections, I show how the execution graphs are generated from the simple program in Figure 4.1.

4.1.1 Initial Values

At first, the algorithm starts off with an empty execution graph. As most programming languages (such as C/C++) demand an initial value for all global (or rather, *static*) variables, every such variable will have a corresponding Write node. These are not yet connected to any other vertex, and represent the root(s) of the directed graph later on.

In the example, there is one global variable: x. After creating the corresponding node, the execution graph can be seen in Figure 4.2.

4.1.2 Handling a Write

Let us take the Write instruction ($W(x, 1)$) and see how we need to maintain the three types of relations on the graph. The instruction is sequenced after the initial write, so there must be a ***po*** edge between them. Furthermore, all possible ***mo*** sequences have to be explored, so we duplicate the graph and on one we insert the new write *after* the initial write to x, and on the other we insert it *before* that. It seems obvious that only of these executions is sensible – but without knowing the memory model, we have to throw away all presumptions about it, such as that same-address stores are totally ordered in ***po***.

At this time there were no Read nodes to accept the written value, so no ***rf*** edges were added. The resulting execution graphs can be seen in Figure 4.3.

4.1.3 Handling a Read

If we take the read instruction now, there are two possible sources it could read from – the initial write $W(x, 0)$, or the other write $W(x, 1)$. We have to explore all possibilities, so

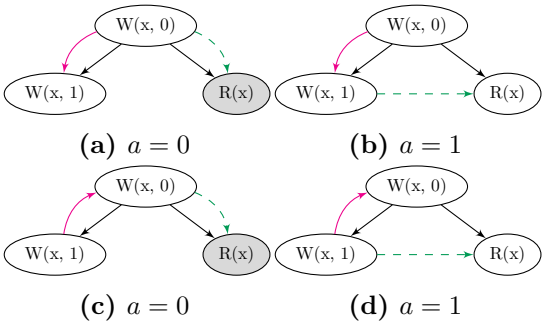


Figure 4.4: Execution graphs after inserting the read

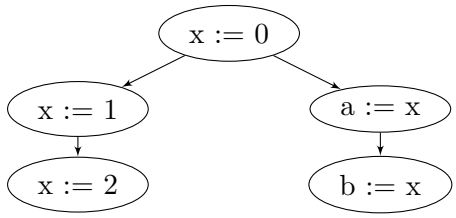


Figure 4.5: Two threads performing two write and two read operations

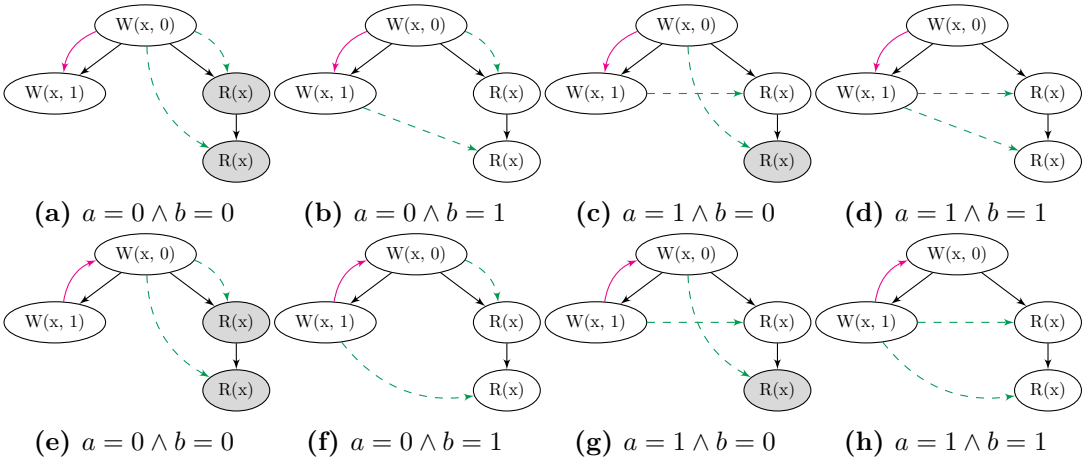


Figure 4.6: Execution graphs after inserting the read

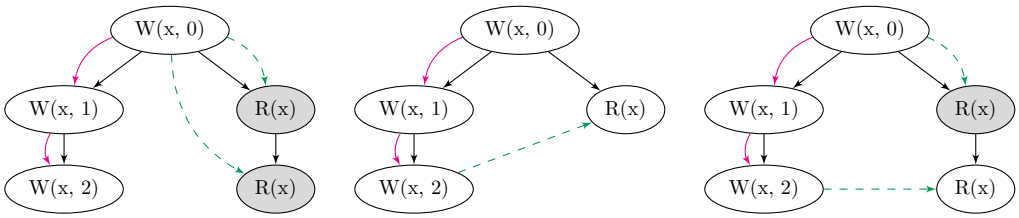
yet again we duplicate the graphs and draw the corresponding ***rf*** edges and the ***po*** relations between the nodes, in *all execution graphs*. Furthermore, we append the new Read instruction to the set of revisitable reads $\mathbb{R}$ in every pre-existing execution graph (so in 2 of the resulting 4). The result is visible in Figure 4.4.

With this addition, the generation of the execution graphs has finished and yielded 4 execution graphs. Now let us extends this example with a new read and new write node, as seen in Figure 4.5!

If we start with the new Read instruction, the process is very similar to the last read – we need to duplicate all existing execution graphs, and in half of them the new read will get its value from the initial value while the other times it reads from $W(x, 1)$. In all pre-existing graphs we also mark the new read as revisitable, and place it in the corresponding $\mathbb{R}$. However, in every other case we also mark all reads in the new node's $(\boldsymbol{po} \cup \boldsymbol{rf})$-prefix non-revisitable to avoid redundant exploration. The resulting 8 execution graphs can be seen in Figure 4.6.

4.1.4 Handling Revisit Sets

If we try to insert the last remaining node $W(x, 2)$, we run into the problem of deciding which (if any) existing reads should get its value instead of the one represented on the



(a) $a = 0 \wedge b = 0$ **(b)** $a = 2$ **(c)** $a = 0 \wedge b = 2$

Figure 4.7: One ***mo***-order of the subexecutions of Figure 4.6a.

existing graphs. This is where the sets of revisitable reads ($\mathbb{R}$) come into play: they contain the reads that are not necessarily bound to their current incoming ***rf*** edge.

As for ***po*** and ***mo*** edges, everything goes as discussed earlier: we bind the first write to the second via a ***po*** edge, and then multiplicate the graph as many times as necessary to being able to place the new node in ***mo*** in any order – in this case *before* the ***mo***-first write, *after* the ***mo***-last write or in-between. This makes the set of execution graphs grow threefold.

After dealing with the ***po*** and ***mo*** edges, new ***rf*** edges have to be added. Firstly, we calculate all *independent* (i.e. the reads cannot reach each other through $\textbf{\textit{po}} \cup \textbf{\textit{rf}}$) subsets of $\mathbb{R}$ to get the so-called *revisit sets* $\mathcal{R}$. Afterwards, we yet again multiplicate the graph to get the number of revisit sets, and assign each their own $\mathcal{R}$. In each of these execution graphs, the elements of $\mathcal{R}$ lose their incoming ***rf*** edges, and instead new ***rf*** edges are added between the new write and these reads. We remove all $(\textbf{\textit{po}} \cup \textbf{\textit{rf}})$-*future* these reads might have had (as this new value potentially disturbs the program flow later on), and mark these nodes and all of their $(\textbf{\textit{po}} \cup \textbf{\textit{rf}})$-*predecessors* non-revisitable in the resulting $\mathbb{R}$.

Following this procedure, we are left with 42 execution graphs[1]. This is more than I can present here, so I included only a subset of the subexecutions of Figure 4.6a in Figure 4.7. Note the lack of an $a = 2 \wedge b = 2$ outcome – because we revisited the first read, its entire future had to be erased as to not breach program flow. However, that outcome *will* be generated – the missing read can be re-added with all the rules that apply when handling a read, and there will be a corresponding outcome.

4.1.5 Effect of the Memory Model

As we have previously seen, even a seemingly simple program might have numerous execution graphs due to the different combinations of ***rf***-edges and ***mo***-orders. However, most of these combinations are most likely illegal according to the underlying memory model – the execution graphs greatly outnumber *consistent* execution graphs in most cases. However, striving to be memory consistency-independent, the algorithm cannot discard executions without proof that a specific execution will cause a memory model violation.

In this work, I evaluate two separate approaches to this problem. One method is to generate all execution graphs without paying attention to legality, until such an execution is found that violates the safety criteria. If that execution is allowed by the memory model, the violation is observable and should be reported that the program does not

[1] Revisiting 4.6a and 4.6e yield 3-3 new execution graphs, 4.6c and 4.6g yield 2-2 and the others remain untouched. Growth from the ***mo*** order is threefold, so $3 * (3 + 1 + 2 + 1) * 2 = 42$ new graphs are created.

Algorithm 1: ExecutionGraphs(p) returns the set of execution graphs in p

Input: A program $p(T, Q)$
Output: The set of execution graphs in p

```
1  𝕊 ← {( InitialStack(T), EmptyGraph(), T)}
2  𝔼𝔾 ← ∅
3  while |𝕊| > 0 do
4  |   (S, EG, E) ← s ∈ 𝕊
5  |   𝕊 ← 𝕊 \ {(S, EG, E)}
6  |   while ¬IsBlocked(p, S, E) do      // IsBlocked returns true if p is blocked
7  |   |   q ← GetNextStmt(p, S, E) // GetNextStmt returns an enabled statement
8  |   |   𝕊 ← 𝕊 ∪ Execute(q, s)          // Execute builds EG, returns new states
9  |   end
10 |   𝔼𝔾 ← 𝔼𝔾 ∪ {EG}
11 end
12 return 𝔼𝔾
```

satisfy its requirements. Otherwise, the search continues until such a violation is found, or all executions are explored.

Another method is to keep track of violations as they happen. A disadvantage of this approach is the overhead of checking for violations every time a new element is added to the execution graph, but this can potentially generate significantly fewer executions in comparison with the former approach. The key idea is to classify violations as *transient* or *permanent* based on the possibility of being dissolved by the rules of execution graph construction. If a *permanent* violation is found, no subexecution of the execution graph can be legal (as the violation can never be dissolved), and therefore there is no need in exploring them. If however, a *transient* violation is found, it does not permit this broad discarding procedure but rather potentially filters the threads that can supply new instructions. The details to this approach can be found in Section 4.3.

4.2 Generating the Executions

In this section I formalize the algorithm introduced in Section 4.1. To summarize its goals:

- The algorithm should generate all possible execution graphs of its input program
- No execution graph should be generated multiple times
- All subexecutions are *self-contained*, i.e. they are entirely independent of each other (to achieve statelessness)

The entry point to this algorithm is the ExecutionGraphs procedure, visible as Algorithm 1. At first, the set of current states $\mathbb{S}$ is initialized with the starting state of the program, expressed as a tuple (S, EG, E). This tuple is slightly different from the one defined in Definition 2.1.2, as the global valuation has been replaced with an execution graph EG. However, EG contains a superset of the information contained in the global valuation (as this is readily available in the form of ***rf*** edges), so we do not lose any information.

The main loop of the algorithm (starting at line **3**) runs until there are states to be explored in $\mathbb{S}$ and fills up the previously initialized set $\mathbb{EG}$ with execution graphs, by choosing one

in each iteration (s) the elements of which are S, EG, E, respectively. Firstly, this element is subtracted from $\mathbb{EG}$ (to avoid being used multiple times), then all of its instructions are processed in the loop starting in line **6**). This loop runs until all threads become blocked (i.e. they have finished execution, or are blocked by some other construct – indicated by the predicate IsBlocked), as after that no instructions remain to be processed. If execution has not yet finished, let q be an *enabled* statement (i.e. in a non-blocked thread, fulfilling all possible prerequisites in S) received from GetNextStmt. After processing q in Execute (see Section 4.2.1), we add the newly created states to $\mathbb{S}$. If a *blocked* state is reached, we add the execution graph EG to the $\mathbb{EG}$ set. When the set of states $\mathbb{S}$ is empty, we return $\mathbb{EG}$ and the algorithm is finished.

4.2.1 Executing a Statement

For the purposes of this algorithm, there are four distinct types of statements in a program:

1. Read instructions (assigning a local variable the value of a global variable)
2. Write instructions (assigning a literal to a global variable)
3. Fence instructions (semantics depends on the memory model)
4. Other instructions (local variable assignments, program flow modifiers, etc.)

The 4^{th} type of statement is only used by the algorithm to determine which statements are enabled (in the form of an *if* or *assume* statement, for example) and to keep track of the local variables which appear on the local valuation. These statements are *not* present on the execution graphs, unlike the first three, which appear directly.

The first three types are handled by the procedures NewRead, NewWrite and NewFence, respectively, which are called by the Execute procedure based on their type. The role of these procedures is to maintain the set of revisitable reads $\mathbb{R}$, and insert or delete nodes and edges in the execution graph EG. The NewFence procedure only places the fence at the end of the corresponding thread with a ***po*** edge, and therefore it will not be further discussed. Its importance will come up in Section 4.3.

Note, that for the sake of simplicity the $\mathbb{R}$ set is assumed to contain reads from a single memory location. Were this not the case, a simple filtration can yield the variable-specific set of revisitable reads, after which it behaves exactly as described.

4.2.1.1 NewRead

When processing a read statement q_r, the goal is to find suitable write statements to read from, and to make sure any future write statement will be able to supply a value as well. To make this happen, we must explore all such subexecutions of EG where q_r reads from an existing write, and in exactly one of those subexecutions we also add q_r to the set of revisitable reads $\mathbb{R}$. This means that in exactly one subexecution the write q_r reads from is not fixed, and therefore any future write might *revisit* it, i.e. supply its own value to q_r.

The algorithm that satisfies these requirements can be seen as Algorithm 2. It iterates over all the writes to the global variable where q_r reads from, and it creates a new state with an added $w \in W \rightarrow q_r$ ***rf*** edge[2] for every such w. In all but the last case (where the

[2] And the ***po*** edges as well (from the last instruction of the given thread when that exists, or all the initial writes if it does not)

Algorithm 2: NewRead(q_r, s) returns the set of new states after adding q_r

Input: A read $q_r(l, g)$ and a state $s(S, EG, E)$ (l: local-, g: global variable)
Output: The new states after applying q_r to s

```
1  𝕊 ← ∅
2  W ← W(EG, g)                                   // All writes in EG writing to g
3  while |W| > 0 do
4  |   w ∈ W
5  |   W ← W \ {w}
6  |   if |W| = 0 then
7  |   |   s ←AddRRead(s, w, q_r)    // AddRRead creates a revisitable read w → q_r
8  |   else
9  |   |   s' ←AddRead(s, w, q_r)          // AddRead creates a simple read w → q_r
10 |   |   𝕊 ← 𝕊 ∪ {MarkFinal(s', {w, q_r})}
   |   |                  // MarkFinal marks the (po ∪ rf)-predecessors non-revisitable
11 |   end
12 end
13 return 𝕊
```

remaining W set is empty), this new state gets put into $\mathbb{S}$, where it corresponds to a newly created and to-be-explored subexecution. When the last w is processed, the resulting state replaces the old state of the execution under construction.

An important difference between the new states and the current state's replacement is *revisitability*. When the current state is replaced with a new one, q_r is marked to be *revisitable*, and is put into $\mathbb{R}$. In all other cases, the read is not only not marked to be *revisitable*, but all ($\boldsymbol{po} \cup \boldsymbol{rf}$)-*predecessors* of both w and q_r are marked as non-revisitable as well. This is done to avoid redundant exploration – the set of revisitable reads that we discarded this way are included as a subset of the $\mathbb{R}$ set in the former case, and are therefore properly handled.

4.2.1.2 NewWrite

A write statement q_w is the source of ***rf*** edges and supply the values to read statements. When a new one is added, on the one hand all possible ***rf*** edges need to be drawn, while on the other all possible ***mo***-orders have to be generated, while remaining non-redundant in all subexecutions (and therefore optimal in the terms of generated execution graphs). Special care has to be taken when generating the ***mo***-orders, because those are not reflected in the final execution graphs as value assignments, and therefore it is easy to create non-unique subexplorations.

The resulting algorithm can be seen as Algorithm 3. It starts off by creating two sets:

- R: The set of all possible revisit sets $\mathcal{R}$. This includes all subsets of $\mathbb{R}$ filtered by the Independent function, which only lets sets containing ($\boldsymbol{po} \cup \boldsymbol{rf}$)-independent reads remain.
- W: The set of (ordered) list of writes to the same memory location as q_w, with the new write q_w included at every possible position.

Algorithm 3: NEWWRITE(q_w, s) returns the set of new states after adding q_w

Input: A write $q_w(g, v)$ and a state $s(S, EG, E)$ (g: global variable, v: value)
Output: The new states after applying q_w to s

```
1  𝕊 ← ∅
2  R ←INDEPENDENT(2^ℝ(EG))                    // Set of all possible independent 𝓡s
3  W ←SEQUENCE(W(EG, g))                      // All sequences among writes in EG writing to g
4  while |R| > 0 do
5      𝓡 ∈ R, R ← R \ {𝓡}
6      while |W| > 0 do
7          w ∈ W, W ← W \ {w}
8          s' ←UPDATEMO(s, w)                 // UPDATEMO updates the mo-order to reflect w
9          while |𝓡| > 0 do
10             r ∈ 𝓡, 𝓡 ← 𝓡 \ {r}
11             s' ←INVALIDATE(s', r)          // INVALIDATE invalidates r's (po ∪ rf)-future
12             s' ←ADDREAD(s', q_w, r)
13             s' ←MARKFINAL(s', {r})
14         end
15         if |W| > 0 ∧ ¬is_last_w(q_w) then s' ←MARKFINAL(s', {succ_w(q_w)})
16         if |W| = 0 ∧ |R| = 0 then
17             s ← s'
18         else
19             𝕊 ← 𝕊 ∪ {MARKFINAL(s', {q_w})}
20         end
21     end
22 end
23 return 𝕊
```

The procedure then iterates over the elements of the product of these two sets, and in each step it updates the ***mo***-order to that of $w \in W$, and then performs the following actions on $\mathcal{R} \in R$:

1. Invalidates the effects of $\forall r \in \mathcal{R}$ on the execution graph (i.e. removes all nodes from their (***po*** $\cup$ ***rf***)-future)
2. Adds a (non-revisitable) ***rf*** edge between q_w and every $r \in \mathcal{R}$
3. Removes $\forall r \in \mathcal{R}$ and all of their (***po*** $\cup$ ***rf***)-predecessors from $\mathbb{R}$

If this iteration is not yet final over W, and q_w is not the last element of the ***mo***-order, all reads (***po*** $\cup$ ***rf***)-preceding the ***mo***-next write of q_w in w are removed from the revisit set $\mathbb{R}$.

Afterwards, the procedure adds this newly created state (while marking all (***po*** $\cup$ ***rf***)-predecessors of q_w non-revisitable) to the set of new states ($\mathbb{S}$) – unless it is the last iteration over $W \times R$, in which case the old state is replaced by this new state in the subexecution that called NEWWRITE in the first place. With this step, the procedure concludes and returns $\mathbb{S}$.

So far, the algorithm generates all possible execution graphs, without paying attention to legality. As discussed earlier, this is one of the approaches to this problem – generate all executions first, and when a safety violation is found, check if the execution leading up

to it was legal. However, there is also the approach when we want to eliminate illegal instructions early on. In this case, the procedures adding to and removing from the execution graphs (nodes and/or edges) must test the result for violations, and either stop the subexecution's exploration (in the case of a permanent violation), or constrain the enabled threads (when a transient violation is found).

4.3 Is an Execution Legal?

Inspired by the *cat* language [7] employed by *herd* [6], memory inconsistency patterns are described over an execution graph using:

1. its edges and nodes,
2. constructs of these edges and nodes, and
3. their primitive properties such as *acyclicity*, *emptyness* and *irreflexivity*.

The edges that the execution graph contains by default (as seen in previous sections) are ***po***, ***mo*** and ***rf*** edges. Nodes are **W**rites, **R**eads and **F**ences. Expressions (constructs) among these edges can be of the following types (expressed as n-ary functions):

1. EMPTY() → new empty execution graph
2. NEXT(EG, EG_1, EG_2) → new execution graph only containing existing edges in EG, whose origin nodes are in EG_1 and target nodes are in EG_2
3. SUCC(EG, EG_1, EG_2) → new execution graph only containing edges between nodes that reach each other in EG, whose origin nodes are in EG_1 and target nodes are in EG_2
4. SOURCE(EG) → new execution graph only containing edges' origin nodes in EG
5. TARGET(EG) → new execution graph only containing edges' target nodes in EG
6. MULTIPLY(EG_1, EG_2) → new execution graph containing edges between all nodes of EG_1 and EG_2
7. SUBTRACT(EG_1, EG_2) → new execution graph containing the differences between EG_1 and EG_2
8. INTERSECT(EG_1, EG_2) → new execution graph containing the intersection of EG_1 and EG_2
9. UNION(EG_1, EG_2) → new execution graph containing the union of EG_1 and EG_2

Furthermore, execution graphs can be *filtered* by type (edge- and node types as well), and we also define expressions for the following constructs:

1. FORACHTHREAD($expr$) → expr is evaluated for $\forall t \in T$
2. FORACHVARIABLE($expr$) → expr is evaluated for $\forall v \in Q$
3. FORACHNODE($EG, expr$) → expr is evaluated for $\forall node \in EG$

```
Empty(
    ForEachThread(
        next(
            rf( EG ),
            CurrentThread( W( EG ) ),
            CurrentThread( R( EG ) )
        )
    )
)
```

Listing 1: Constraint on the lack of intra-thread ***rf*** edges

For each of these complex constructs, nodes can be filtered by the current iteration of threads, variables or nodes – for example, in Listing 1, the CurrentThread filter will only let those nodes in EG pass through to the next function, which are in the thread of the current iteration step in ForEachThread.[3]

Furthermore, properties (*acyclicity*, *irreflexivity* and *emptyness*) can be combined through the Boolean operations *and* and *or*, but cannot be negated (see Section 4.3.1 for justification).

Beside following rules constructed from the expressions above, memory models must also abide by the following rules:

- Reads can only read from existing writes[4]
- Both Writes and Reads behave *correctly*, i.e. they atomically interact with exactly one memory location (and that location is fixed from the source)

However, if these constraints are satisfied, any memory model can be specified. To justify this, consider the constructs the *cat* language describes [7]. It is easy to see that all such constructs either exist in this model as well, or can be created by combining some functions above – see Appendix A.

4.3.1 Permanence

As mentioned in previous sections (4.1.5, 4.2), we classify inconsistencies as *transient* or *permanent*. The better an algorithm is at identifying *permanent* violations, the fewer pointless executions we have to explore – the goal is to keep this number at 0.

The *permanent* violations only consist of nodes that are:

- not ($\boldsymbol{po} \cup \boldsymbol{rf}$)-preceded by any revisitable reads

and edges that are:

- not removable later (e.g. ***rf*** edge going to a *non-revisitable* read), or
- not path-removable[5] later (only in the case of succ, e.g. ***mo***)

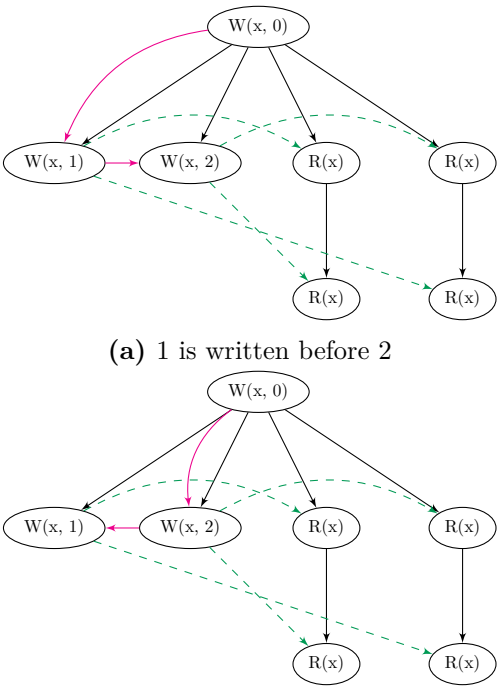
[3] Note that this example expresses inconsistencies in the form of same-thread ***rf*** edges, which is rarely a violation in any memory model.

[4] If a write gets removed from the execution graph, no read can read from it anymore

[5] If there is an edge $a \to b$, then in all subexecutions there will be a path $a \rightsquigarrow b$

```
1  Empty(
2      ForEachNode(
3          R( EG ),
4          next(
5              rf( EG ),
6              source(
7                  succ(
8                      mo( EG ),
9                      W( EG ),
10                     source(
11                         next(
12                             rf( EG ),
13                             W( EG ),
14                             CurrentNode()
15                         )
16                     )
17                 )
18             ),
19             target(
20                 succ(
21                     po( EG ),
22                     CurrentNode(),
23                     R( EG )
24                 )
25             )
26         )
27     )
28 )
```

Listing 2: Coherence relation for weak memory



(a) 1 is written before 2

(b) 1 is written after 2

Figure 4.8: Four threads demonstrating incoherence

Any other violation is classified as a *transient* violation.

The reason why negation of the properties is not allowed in the memory model is this *permanence* detection – it is easy to decide whether a result might become non-empty, acyclic or irreflexive but deciding the same about their negations is much harder (e.g. will an empty set become non-empty at some point?). Lacking information on *permanence* will lead to suboptimal exploration, which we are trying to avoid – so this constraint on the properties exists to aid that. Furthermore, there is no reason why a violation might come from being acyclic, empty or irreflexive – an empty execution graph would be violating if that were the case, and then there is no point in running the algorithm.

As an example, see Figure 4.8 and the accompanying rule *coherence* on Listing 2. The rule states that *there shall be no such construct for any read instruction (lines 1-3) where an* **rf** *edge exists (lines 4-5) between a write node such that it is* **mo***-succeeded by the write (lines 6-9) the chosen read reads from (lines 10-16); and a read node that is* **po***-preceded by the chosen read (lines 19-25).* Consider the two executions in Figure 4.8 in light of this rule – in the first case (4.8a) $W(x, 2)$ is sequenced after $W(x, 1)$ and therefore the 4^{th} thread is incoherent (there *is* a ***mo***-previous write of the write a read reads from, which supplies its value to a ***po***-successor of the same read), while in 4.8b the same can be said about thread 3 – furthermore, there are no executions where the two threads could read the values in a different order (this is what *coherence* is about). As there are no revisitable reads in either of the executions, and the only removable edge type (***mo***) is not path-removable and is only present in SUCC functions, all violations are *permanent*, i.e. no further exploration is necessary to know that every subexecution will also be in violation of the MCM.

Now consider the execution in Figure 4.9 where on the violating reader thread the first read is revisitable. Even though this is the same violation as in Figure 4.8b, the nodes

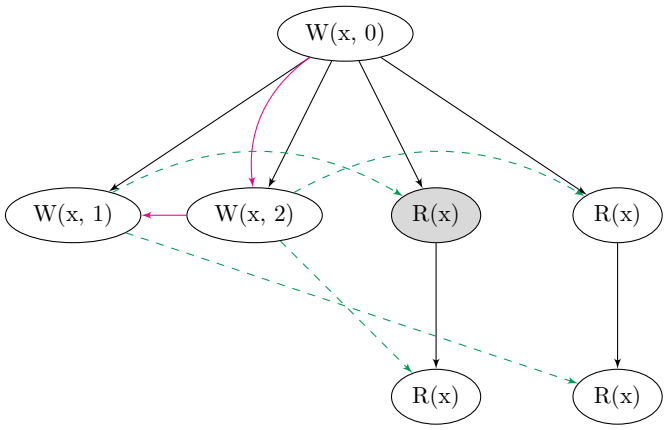


Figure 4.9: The first read is revisitable on thread 3

and edges that make up the violation are not all non-removable. The ***rf*** edge going to the revisitable read is easily removable by a new write to x, and by extension to that the next read disappears (as the $(\boldsymbol{po} \cup \boldsymbol{rf})$-future of the first read is discarded), making it removable as well. This means that the exploration must not end here, there still might be a future subexexution where this violation is dissolved – but it does not mean that nothing can be done. We can constrain the set of enabled threads to aid dissolving this issue: all the threads that contain removable nodes that also constitute a violation are disabled until this violation is sorted out. This does not remove any possible subexecutions from the set of possible ones – on the one hand, the order of handling instructions does not matter, and on the other hand that future would always be removed if the removable nodes get removed (because any ***po***-successor to a removable is also a removable).

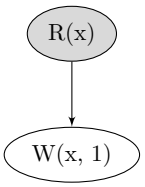
4.4 Soundness and Optimality

As the algorithm is based on the one described by *Kokologiannakis et al.* [24], their proofs on optimality and soundness (with minor modifications) hold for this method as well. Therefore, only an outline will be presented here that is necessary to understand *how* this algorithm works, and not a full formal proof.

4.4.1 Soundness

To prove that the algorithm is *sound*, it needs to be shown that there will never be any false positive and false negative results.

False negatives in this context mean that we failed to report a program that breaks a safety requirement. This can be caused by three things: either the assertion (or other violation) was not reached in program flow, the assertion was evaluated to be *true* when it should have been *false*, or a false MCM violation was reported and therefore the assertion violation was dismissed. The first case is impossible following the subexecution generation algorithm: all reads will eventually try to read from all writes (see [24] for further details), and therefore every possible branch of the program will be explored. The second is also impossible when the algorithm is implemented correctly: the state vector holds a valuation to all variables, and therefore any expression can be easily and correctly evaluated. The final possibility for a false negative outcome is in the hands of the specification designer: if the MCM was correctly specified, there will never be an incorrectly reported violation.



(a) $W(x, 1)$ is removable because $R(x) \in \mathbb{R}$

(b) $R(x) \rightarrow W(x, 1)$ is removable because $R(x) \in \mathbb{R}$

Figure 4.10: Types of removability

False positives on the other hand are cases where we reported a safety criterion violation by mistake. Through the same reasoning as with false negatives, this can never happen – the algorithm will always correctly evaluate both an assertion- and a MCM violation.

4.4.2 Optimality

To show that the algorithm is optimal, consider the method where we discard any *permanent* violations. I show that in each step, all explored subexecutions have the potential to be legal over the given MCM.

As seen in [24], their algorithm (that hardcoded the check for RC11 memory model violations) is *optimal* in the sense that it never generates two identical subexecutions. This needs to be extended with the fact that no subexecution will be explored that can only yield violating ones.

To do this, I show that no *transient* violations exist that yield only violating subexecutions. Consider the types of removables – either a node that is $(\boldsymbol{po} \cup \boldsymbol{rf})$-preceded by a revisitable read, or an edge that is not yet final, i.e. that can be re-routed when more information is added to the execution (see Figures 4.10a and 4.10b).

In both cases, the violation might come from the following sources:

1. $W(x, 1)$ or the ***rf*** edge is in an execution graph that is supposed to be *empty*
2. $W(x, 1)$ or the ***rf*** edge is part of a cycle in an execution graph that is supposed to be *acyclic*
3. $W(x, 1)$ or the ***rf*** edge is part of a reflexive relation in an execution graph that is supposed to be *irreflexive*

In all cases, the lack of the removables consitute a dissolution of the violation – in the case of the emptyness property this is trivial, as it leaves the set that is supposed to be empty; in the case of the cycle it is broken if any edge or node is taken out; and in the last case a reflexive relation ceases to exist when either of the two edges (that constitute the relation) or the node disappears. Thus, there will be no such subexecution where a violation is marked as *transient* but no legal execution can come of it as a subexecution[6].

[6] note that in a specific execution this might still be the case – but from the information at hand during this decision that could not be deduced

Chapter 5

Implementation

In the previous chapter I introduced the algorithm for generating consistent execution graphs given a specific memory model. However, that description was purely theoretical for the sake of formality – in this chapter, I present a practical implementation as part of the Theta framework[1] [31], which is a generic and configurable verification framework supporting various formalisms and algorithms.

5.1 The Formalisms

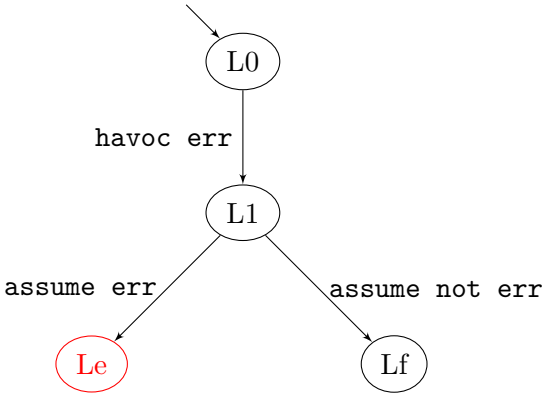
In order to implement the algorithm, I needed two formalisms: one for the representation of the programs defined in Definition 2.1.1 and another for the representation of the memory model as seen in Section 4.3. For the latter, I also could have used the feature-wise similar *cat* language [7], but to aid the simplicity of the implementation I opted for a DSL consisting of a few constructs (as opposed to the syntactically rich *cat* language). Its grammar (in BNF-like format) is:

```
<SPECIFICATION> ::= <DEFINITION>*  <CONSTRAINT>*

<DEFINITION>    ::= <ID> "=" <EXPR>
<EXPR>          ::= "(" <EXPR> ")"                                  # enclosedExpr
                  | "{}"                                            # emptysetExpr
                  | <ID>                                            # namedExpr
                  | <ID> ("[" <TAG> "]")*                           # taggedExpr
                  | <ID> "(" <EXPR> "->" <EXPR> ")"                 # nextExpr
                  | <ID> "(" <EXPR> "-->" <EXPR> ")"                # succExpr
                  | "for_each_var begin" <EXPR> "end"               # forEachVarExpr
                  | "for_each_thrd begin" <EXPR> "end"              # forEachThreadExpr
                  | "for_each_node" <EXPR> "begin" <EXPR> "end"     # forEachNodeExpr
                  | <EXPR> "union" <EXPR>                           # unionExpr
                  | <EXPR> "intersect" <EXPR>                       # sectionExpr
                  | <EXPR> "\" <EXPR>                               # differenceExpr
                  | <EXPR> "*" <EXPR>                               # multiplyExpr
                  | "source(" <EXPR> ")"                            # sourceExpr
                  | "target(" <EXPR> ")"                            # targetExpr

<CONSTRAINT>    ::= <ID> ("acyclic" | "irreflexive" | "empty")      # simpleConstraint
                  | "(" <CONSTRAINT> ")"                            # enclosedConstraint
                  | <CONSTRAINT> "&" <CONSTRAINT>                   # andConstraint
                  | <CONSTRAINT> "|" <CONSTRAINT>                   # orConstraint

<ID>            ::=  "[a-zA-Z]"("[a-zA-Z0-9_]")*
```

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

```
main process simple {
    var err : bool

    init loc L0
    loc L1
    error loc Le
    final loc Lf

    L0 -> L1 {
        havoc err
    }

    L1 -> Le {
        assume err
    }

    L1 -> Lf {
        assume not err
    }
}
```

(a) Source code



(b) Graphical representation

Figure 5.1: A fairly minimal CFA program

Semantically, *expressions* are functions of the type $e(p_i^*) \to (EG \bigvee \{nodes\})$, i.e. they take some arguments and produce either an execution graph or a set of nodes[2]. The *definitions* consist of a named expression, whose name can also be used as part of a *constraint* (or a *namedExpr*) to specify properties of the expression. These properties include the three checks (*acyclicity*, *irreflexivity*, *emptyness*) and boolean *and* and *or* operations among them.

As for the representation of a concurrent program, I opted to use the *eXtended Control Flow Automaton* (XCFA) formalism, which I had previously integrated into the Theta framework and it exists as a beta future of the tool[3]. This formalism relies on the CFA formalism at its core, which has the following elements:

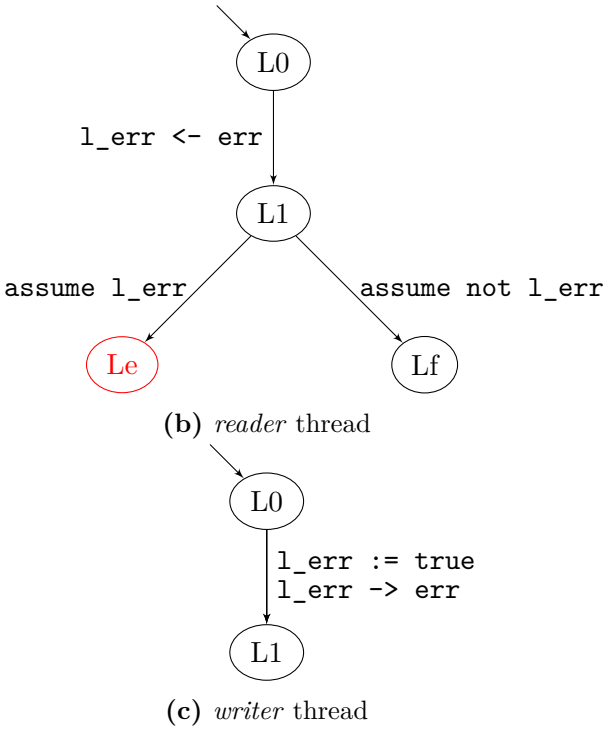
- Variables
- Locations
  - Initial location
  - Final location (not mandatory)
  - Error location (not mandatory)
- Transitions between Locations
- Statements on transitions
  - *Assume* statements as guards
  - *Arithmetical* and *logical* statements of variables
  - *Havoc* statements for as nondeterministic inputs
  - *Skip* statements for no-op transitions

See Figure 5.1 as an example to a pure CFA model.

[2]For a more detailed specification see Appendix A
[3]https://github.com/ftsrg/theta/blob/xcfa/

```
var err : bool (false)
main process reader {
    main procedure mproc() {
        var l_err : bool

        init loc L0
        loc L1
        error loc Le
        final loc Lf

        L0 -> L1 {
            l_err <- err
        }

        L1 -> Le {
            assume l_err
        }

        L1 -> Lf {
            assume not l_err
        }
    }
}
process writer {
    main procedure mproc() {
        var l_err : bool
        init loc L0
        final loc L1

        L0 -> L1 {
            l_err := true
            l_err -> err
        }
    }
}
```

(a) Source code



(b) *reader* thread

(c) *writer* thread

Figure 5.2: A fairly minimal XCFA program

XCFA on the other hand extends CFA the following ways:

- CFAs represent *procedures*, which are self-contained pieces of behavior embedded into *processes*
- *Processes* represent the threads of a concurrent program
- *Global* variables are accessible from any scope, *thread-local* variables are accessible from the containing process, and *local* variables are the variables of the CFA-like *procedures*
- New statements aid concurrent program representation
  - *AtomicBegin*, *AtomicEnd* statements for atomic blocks
  - *Wait*, *Notify*, *NotifyAll* statements for conditional synchronization
  - *MutexLock*, *MutexUnlock* statements for mutual exclusion
  - *Load* and *Store* statements for global variable accesses
  - *Call* statements for procedure calls
- Variables can have *initial values*

A similar program to Figure 5.1 can be seen as an XCFA program in Figure 5.2.

Semantically, an XCFA program is handled the following way in this implementation:

1. All processes are started at the beginning of the execution and run asynchronously
2. A process starts at its *main* procedure's initial location and finishes in its final
3. An *error-state* can either be defined as an error location (when it is local), or as a predicate on the state vector
4. *Atomic* blocks run exclusively when active (i.e. no other thread can perform anything during its execution)
5. *Atomic* accesses are denoted by the *Load* and *Store* statements, non-atomic accesses by the *Assignment* statement
6. Accesses to global memory are not ordered

As the algorithm is working with the accesses to the global memory area, other types of synchronization are forbidden (conditional and mutex-based synchronizations).

5.2 The Algorithm

As mentioned above, I developed a PoC implementation to the proposed algorithm as part of the Theta framework [31]. As this framework is predominantly developed in Java, I opted to use this language as well – because I am only trying to show the applicability of the approach and not the absolute best performance it could achieve, this is not a problem and also allows for a more concise implementation.

The core of the algorithm for generating executions is about the closest it can get to the pseudocode in Section 4.2. The main difference is the use of a *maximal depth* parameter to avoid being stuck in a loop forever.

Furthermore, the use of an *ThreadPoolExecutor* to explore the subexecutions means that the algorithm can work concurrently on the set of execution graphs – as a subexecution barely accesses globally available structures, this can mean a significant speedup compared to the single-threaded approach. This affects neither soundness nor optimality, because due to the *statelessness* of the algorithm all subexecutions are independent of each other (except for direct descendants).

As part of the CLI tool for the algorithm, a number of attributes are customizable before running the algorithm. These are the following:

```
    --insitu-filtering
      Enables in-situ filtering for memory model violations (Default: false)
    --all-states
      Print all resulting states as .dot files (Default: false)
  * --mcm
      Path of the input MCM model
  * --model
      Path of the input XCFA model
    --poolsize
      Size of the thread pool (Default: 1)
    --print-cex
      Print counterexample as cex.dot (Default: false)
    --max-depth
      Maximal depth of exploration in any thread (0 for unlimited depth) (Default: 0)
```

As seen from these switches, there are two main options for the algorithm: if the `--all-states` switch is specified, then the algorithm will not stop at the first error

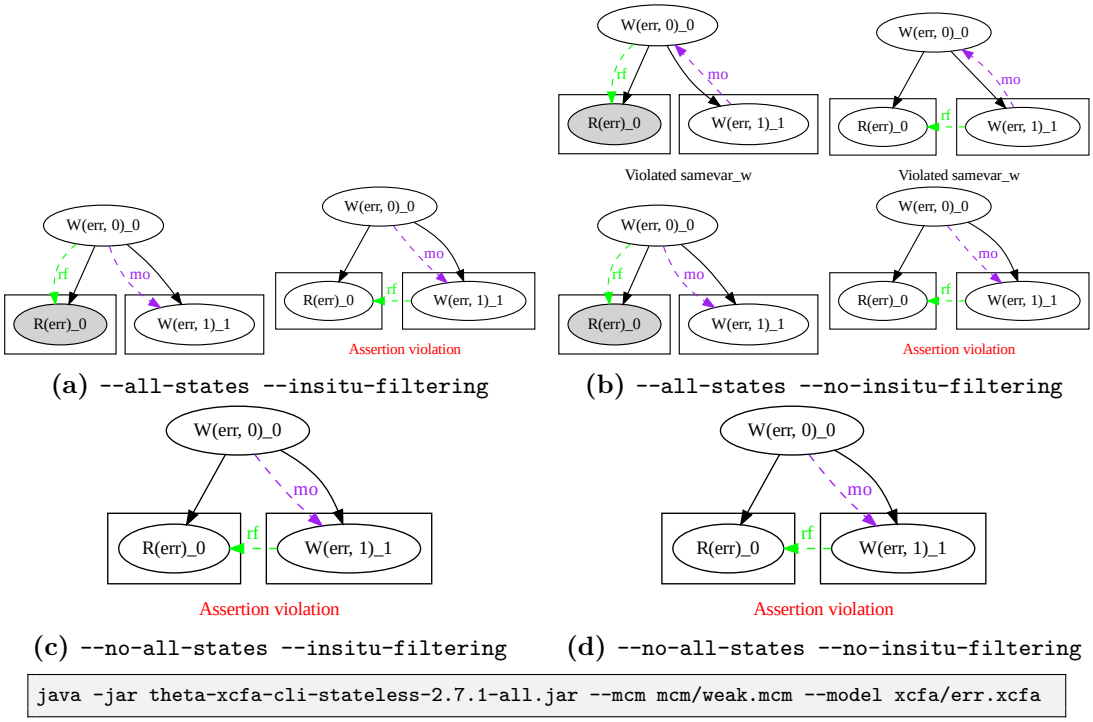


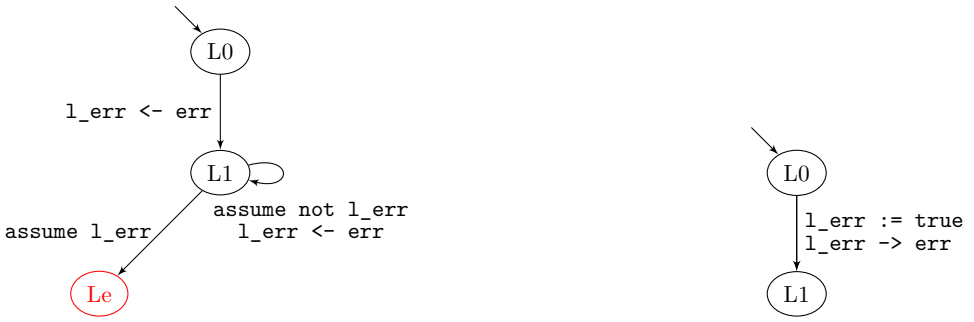
Figure 5.3: Returned graphs of Figure 5.2 over a weak memory model

state but will rather explore all executions and print the results as .dot files, and if the `--insitu-filtering` switch is specified not all subexecutions will be explored, only those containing no *permanent* violations. This gives four possible methods for verifying a program, as seen in Figure 5.3 – they all produce the counterexample when the read reads from the second write, but the set of returned states differ.

The maximal depth attribute comes in handy when a thread is updating its state based on a global variable in a continuous loop. Consider the program in Figure 5.4 – it is almost the same as Figure 5.2, but this time the *reader* process waits in a loop until the value of *err* becomes true. When verifying this with a `--max-depth` value of 1, we get the same results as in Figure 5.3 – except for the assertion violations, because after adding the first node on the execution graph, we cease to explore instructions on that thread. However, if we increment this number, we get the possible executions that can happen in that amount of iterations[4] as seen in Figure 5.5. At a bound of N, the output (with the `--insitu-filtering` switch) contains all executions where at the $n^{th}(n < N)$ iteration the *reader* thread finally reads *true* and the *error* location is reached (e.g. an assertion is violated), and two further executions, where one represents the possibility that the *reader* thread never reads *true* under this bound, and the other represents the case that the thread reads *true* at exactly the bound value – this causes execution to stop, so the XCFA cannot step into its *error* location.

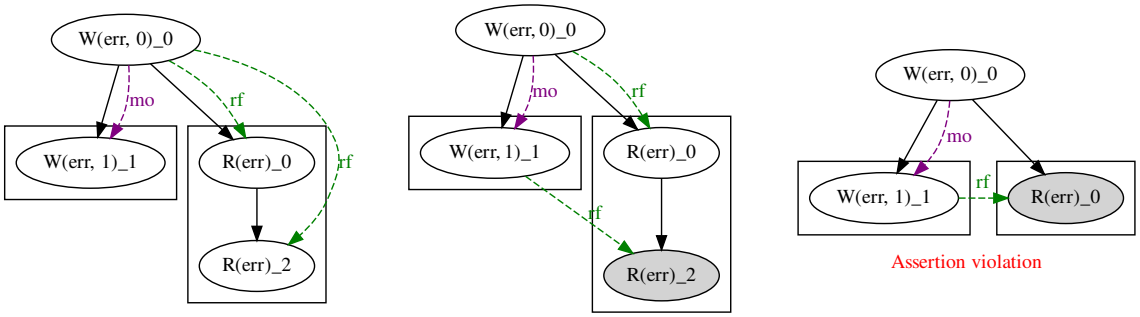
Without this upper bound on the number of nodes in the execution graph, exploration of the state space will never terminate – there will always be a case where the loop can still read $false$ in its next iteration. Of course, this also means that we have constrained the set of verifiable programs to those with a *finite* state space, and to any other program we can only say whether it violates safety criteria in the first N instructions. This is however

[4]In general, the number of added nodes is what counts – any number of unrelated instructions can happen, the execution graph can only contain that many nodes in a given thread

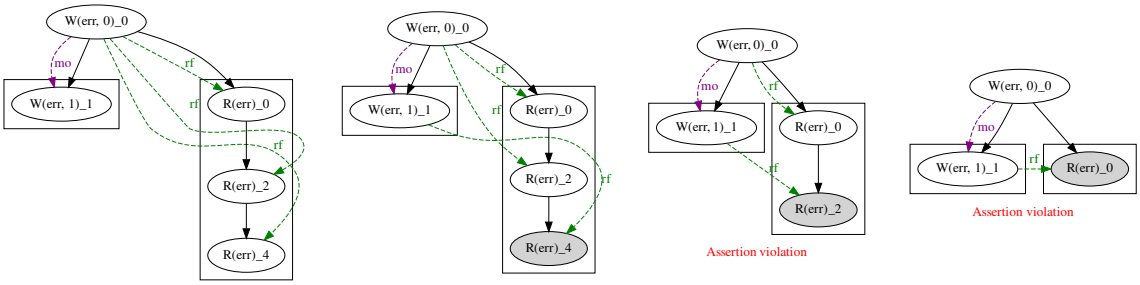


(a) *reader* thread **(b)** *writer* thread

Figure 5.4: Figure 5.2 with a loop



(a) `--max-depth 2`



(b) `--max-depth 3`

Figure 5.5: Output of the verification with a given bound

a practice employed by many tools that operate with a Bounded Model Checking (BMC) algorithm – and therefore I classify this as a minor limitation as opposed to an unsuitable approach.

Chapter 6

Evaluation

In order to show the capabilities of the proposed approach, in this chapter I show how simple problems can be mapped to the inputs of the implementation, and how it performs compared to tools targeting similar fields (*Nidhugg* [2] and *RCMC* [24]). Finally, I discuss future improvements to the approach.

6.1 Carrying Out Hardware-Software Co-Verification Tasks

As seen throughout the chapters of this work, the proposed algorithm takes two formal models as inputs: a memory consistency model and a concurrent program, the formalisms for which have been introduced in 5. In this section I show how the three basic memory consistency models (as seen in Section 4.1.5) can be mapped to their declarative specification and elaborate on the transformation of concurrent programs written in a high-level language to the formal XCFA language.

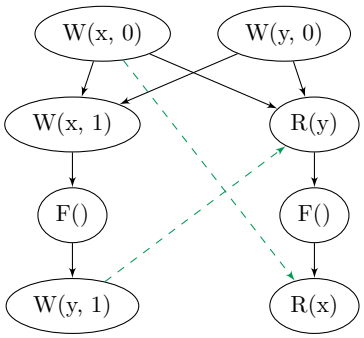
6.1.1 Memory Modeling

When it comes to memory models, there are two basic aspects to handling them:

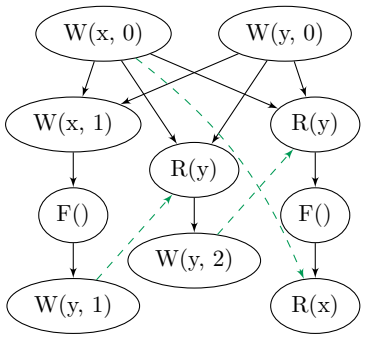
1. How fence-less $W \to R$, $R \to W$, $W \to W$ and $R \to R$ relationships behave (both intra- and inter-thread)
2. How fences modify the above relationships

The toolset for handling these questions is the functions and properties in the memory specification language. Firstly, let us see the weakest memory model (Definition 2.2.4) – there are no constraints on different-location accesses at all, but every same-location access in the source defines a partial order in the program. Furthermore, the architecture is *causal* and *coherent* – meaning a read must read from a non-***po***-successor write, and any two threads must read the same location in a non-contradicting order. These rules yield the following definitions:

1. `samevar_w`: same variable *write* events are ordered in ***po***, i.e. the ***mo*** and ***po*** edges cannot form cycles.
2. `causality`: no ***rf*** edge can span backwards in the ***po*** tree.



(a) Two threaded illegal execution



(b) Three threaded illegal execution

Figure 6.1: Two examples of synchronization using fences: direct and transitive constraints on executions

3. `coherence`: no *read* event shall read from a *write* such that a ***po*** successor *read* reads from a ***mo*** previous *write*.

The *fence* instruction can have multiple interpretations based on the specific architecture[1]. For the sake of this example, consider an architecture, where a fence means that any *Write, Read* event before the fence must finish before any *Write, Read* event finishes after the fence. In terms of a weak memory model, this means that if a *fence* is between two *read* instructions on any (***po*** ∪ ***rf***)-path, then they cannot read from two writes that have a *fence* instruction between them on any (***po*** ∪ ***rf***)-path and are in a reverse order to the reads (See Figure 6.1).

With these rules, we get following specification:

```
porf      = po union rf

samevar_w = po(W --> W) union mo(W --> W)

causality = rf(W -> R) union po(R --> W)

fence     = for_each_node F begin
              for_each_node F begin
                  porf(
                      source( porf(
                          source(rf(W --> target(porf(F[node1] --> R))))
                          -->
                          F[node0])
                      )
                      -->
                      target( porf(
                          F[node0]
                          -->
                          source(rf(W --> source(porf(R --> F[node1]))))
                      )
                  )
              end
            end

coherence = for_each_node R begin
              rf(source(mo(W --> source(rf(W -> R[node])))) -> target(po(R[node] --> R)))
            end

fence empty
coherence empty
samevar_w acyclic
causality acyclic
```

[1] And there can be more than one type of fence – in the implementation, a string literal can be used to distinguish them, but this feature is not used here.

To adapt this memory model to the other two consistency families (*SC*, *TSO*), the only rule that needs to change is the *coherence* relation. The other rules might become unnecessary (e.g. the *fence* relation is unnecessary over *SC*), but they will never be too constraining.

The *coherence* relation should always forbid same-variable reads that would observe writes in a different order than what either ***ppo*** or ***mo*** allows. ***mo*** directly appears on the execution graphs and therefore using it to specify coherency is easy, and in the case of *weak* memory ***ppo*** is a subset of ***mo***. However as memory gets stricter, ***ppo*** outgrows ***mo***: in the case of *TSO*, a ***ppo*** edge appears between all *Write* events on the same thread, and in the case of *SC*, a ***ppo*** edge is present between all such events that can reach each other via ***po***.

The revised memory model is as follows (I omitted the unchanged rules):

```
ppo       = po(A --> A) // SC
ppo       = po(W --> W) // TSO

moppo     = mo union ppo

coherence = for_each_node R begin
              rf(source(moppo(W --> source(rf(W -> R[node])))) -> target(po(R[node] --> R)))
            end


coherence empty
```

6.1.1.1 Modeling Faulty Hardware

As discussed in Section 3.3, just because a hardware has a design flaw, we might be able to still use it, even for safety critical tasks – but we have to verify whether a specific program could become unsafe when run on the architecture. To achieve this, we must be able to model the flaws of the hardware.

Consider the ARM Cortex-A9 processor. It has a design flaw that allows incoherent reads from any given thread if there are no fences between two reads to the same location – this is called the *Read* $\rightarrow$ *Read hazard* [1]. Otherwise (being ARM), it employs a weak memory model similar to the previously constructed one.

In this case, removing the *coherence* relation from the memory model yields the (faulty) architecture's specification. Any two reads might read from writes *not* in ***mo***-order, but if we place a fence (such as the *DMB* instruction in ARM's ISA), the problem is solved, *coherency* is achieved yet again.

6.1.2 Mapping Programs to Formal Models

A concurrent program is most likely to be given in *source code* format to the verification tool, as that is a convenient location – formally verified compilers exist [25], and can produce binaries with certain guarantees towards safety, implying the source itself is written well. Furthermore, feedback to binaries might not be as straightforward to incorporate into the product as the same towards the source itself. However, source code must be transformed into the formal model a verification tool works on.

In this case, concurrent program sources must be translated into the *XCFA* formalism. This can either be done by hand by the software designer, or via an automated method. The first approach works best for cases where demonstrational purposes are important (e.g. models in this document), but is unreliable due to the human element in the verification

	relaxed	release	acquire	seq_cst
Memory access types	Store, Load	Store	Load	Store, Load
Fence before	no	yes	no	yes
Fence after	no	no	yes	yes

Table 6.1: Memory ordering types mapped to fence usage in XCFA

chain. The second is designed to produce a true abstraction of the program[2] reproducibly and verifiably.

Such a tool exists for the XCFA formalism using LLVM as a compiler, integrated into *Gazer*[3] [29], the sister project of Theta, in an *alpha* state. Right now it supports most necessary elements for the algorithm, but has not yet been tested for edge cases – elevating its state to stable is part of the immediate plans towards producing an end-to-end verification tool. The programs in this work have been in part translated by hand and in part by this tool, but in this latter case the models have been cross-checked to check if they are indeed correct.

Even though this tool enables the translation of concurrent programs (mainly written in C) to XCFA, there is still a discrepancy between the two languages needing to be addressed: the memory ordering rules. In C11 [22] (and most concurrent programming languages), memory ordering types are used in place of architecture-dependent implementations of elevated sequentiality for atomic memory accesses. For example, on an *SC* system, all atomic accesses (independent of their ordering) are sequential. For a *weak* memory system however, there is a difference between e.g. a *relaxed* and an *acquire* load – the latter providing synchronization capabilities not exposed by a simple memory instruction. The compiler will then map this to assembly instructions (such as fences around the memory access).

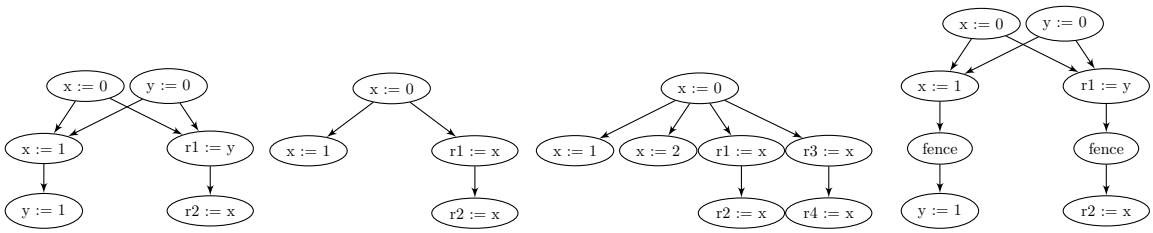
Memory ordering types are not yet supported by either the translation tool or the algorithm itself. Instead, *compiler mappings* can be used to mimic their behavior – this is an accepted procedure employed by other tools as well [32]. These compiler mappings can be found in Table 6.1.

6.1.3 Verifying Litmus Tests

To show the workflow of verifying hardware-software systems by the presented algorithm and tool, I use it to generate all consistent states to some litmus tests over the previously defined memory models. The set of litmus tests in question can be seen in Figure 6.2 and it contains the following litmus tests:

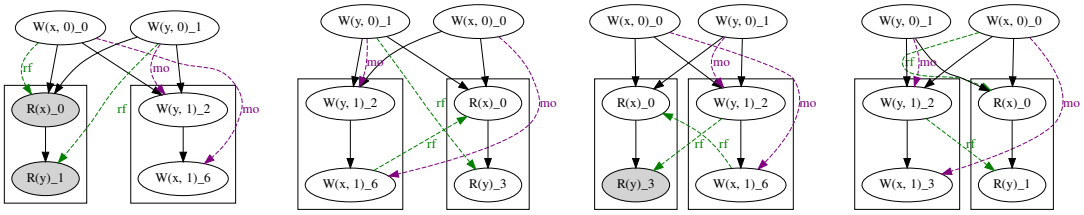
- 6.2a (*2w2r*): A simple litmus test containing two writes and two reads to different locations on two threads.
- 6.2b (*cow2r*): A litmus test for verifying coherency, one thread writes and another threads reads it back twice.
- 6.2c (*coww2r2r*): A complex litmus test for verifying coherency, two threads write two values and two threads read it back twice (each).
- 6.2d (*mp*): A message-passing litmus test for verifying fences.

[2] i.e. it shall not introduce any new behavior
[3] https://github.com/ftsrg/gazer

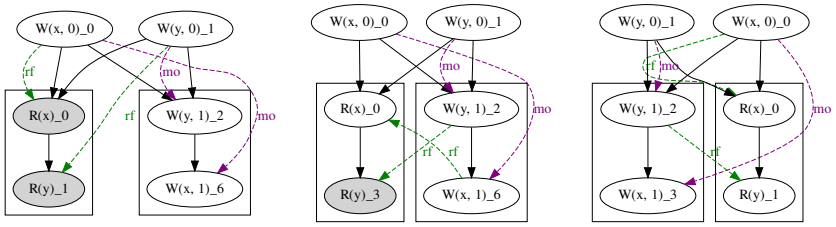


(a) *2w2r* **(b)** *cow2r* **(c)** *coww2r2r* **(d)** *mp*

Figure 6.2: Litmus tests



(a) *Weak*



(b) *SC*

Figure 6.3: Executions of the litmus test in Figure 6.2a

From the four memory models (*weak, TSO, SC, weak-a9*) and four litmus tests, a total of 16 configurations could be run. In the following sections, I show the outcome of the "interesting" runs among these test cases. In every case, I ran the algorithm with the `--all-states` and `--insitu-filtering` switches to return all consistent executions of the input programs. Note that the resulting graphs are the visualization of the raw output the tool produces, and therefore the lack of visual appeal can be justified by the use of an automated tool (graphviz dot[4]).

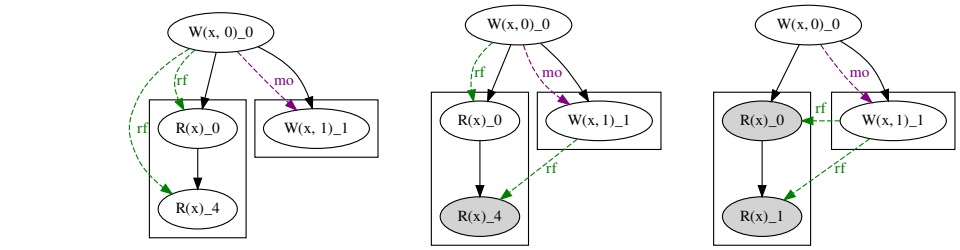
6.1.3.1 The *2w2r* Litmus Test

This litmus test aims to bring out the reordering capabilities of the weakly ordered systems. If we contrast the *weak* and *sc* runs of this litmus test in Figure 6.3, we can see that there is only one execution that is disallowed by the sequential model, where the first read reads 1, but the second read still reads the initial 0.

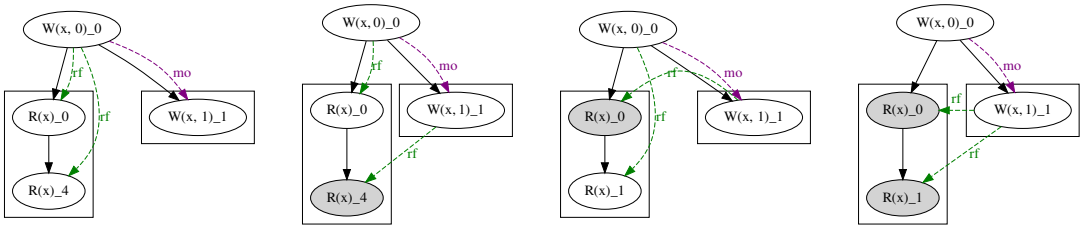
6.1.3.2 The *cow2r* Litmus Test

This litmus test aims to bring out incoherent behavior from a weakly ordered system. Over all correct memory models (*weak, SC, TSO*) the outcome $r_1 = 1 \wedge r_2 = 0$ is not allowed, as that would violate coherency. However, in the case of the ARM Cortex-A9 memory model, this outcome is present in the set possible executions, as seen in Figure 6.4.

[4]https://graphviz.org

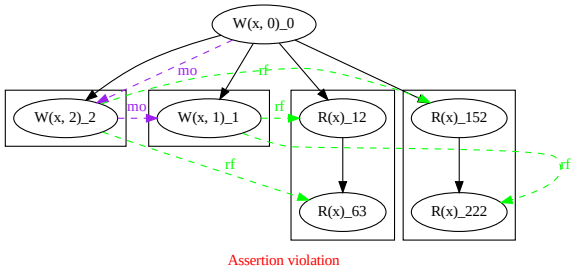


(a) *Correct MCMs*



(b) *Cortex-A9 MCM*

Figure 6.4: Executions of the litmus test in Figure 6.2b



(a) *Cortex-A9 MCM*: coherency violation

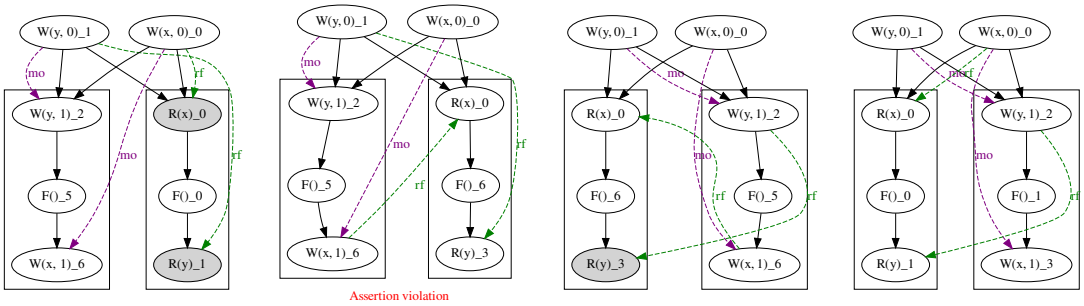
Figure 6.5: Assertion violating execution of the litmus test in Figure 6.2c

6.1.3.3 The *coww2r2r* Litmus Test

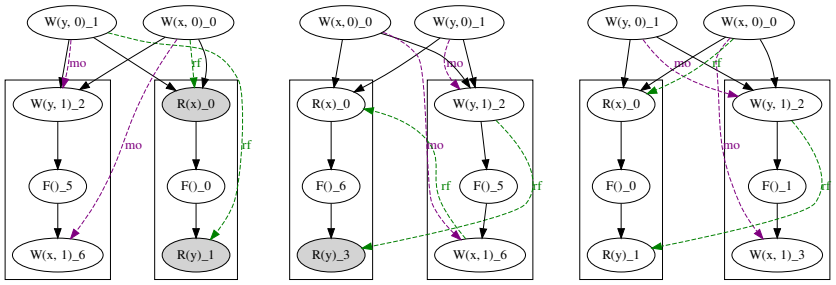
This litmus test (akin to the *cow2r* litmus test) tries to cause incoherent behavior. In this case, (in)coherency is harder to detect: the two writes execute arbitrarily, and therefore incoherency is not interpretable for a single reader thread. With two such threads however, incoherency would mean that one reads in a different order then the other. Over all correct memory models (*weak, SC, TSO*) this is not observable. However, in the case of the ARM Cortex-A9 memory model, threads *can* read incoherently, as seen in Figure 6.5 – this case the `--all-states` switch was omitted, as more than 100 consistent execution graphs were otherwise produced, and therefore the tool only reported whether there was a violation of the criterion above.

6.1.3.4 The *mp* Litmus Test

The *mp* litmus test uses fences to force the sequentiality of instructions, even on weakly ordered systems. The name is the abbreviation of the *message passing* expression, as the *acquire-release* semantics used here can make sure a complex (and possibly non-atomic) message ha been written entirely to memory when the reader reads an atomic "done" flag – in this case, the message is the variable x, and the flag is y. The reader shall never read $r_1 = 1 \wedge r_2 = 0$.



(a) *Nofence*



(b) *Other architectures*

Figure 6.6: Executions of the litmus test in Figure 6.2d

All previously modeled architectures pass this test – even the *weak-a9* model respects fences. Now consider another faulty memory model, *nofence*, where the *fence* relation is not respected by the architecture. The results can be seen in Figure 6.6.

6.2 Performance Evaluation

In order to get a picture on the scalability of the approach, I compare the execution time and memory consumption of three tools:

1. Nidhugg [2], because it can handle different memory models;
2. RCMC [24], because it serves as the basis of this work;
3. The PoC implementation of the proposed algorithm in four configurations (called *TUE* in short for Tool under Evaluation):
   (a) Verification mode (i.e. it stops at the first observable safety violation)
   (b) Filtering verification mode (i.e. it stops at the first observable safety violation, only explores consistent subexecutions)
   (c) State generation mode (i.e. it explores all subexecutions of the program)
   (d) Filtering state generation mode (i.e. it explores all consistent subexecutions of the program)

I have used the tool *benchexec* [11] to reliably measure the execution time and memory consumption of the above configurations on a virtual server running on the BME Füred cloud infrastructure[5]. The server was equipped with 8 cores and 64GB memory running Ubuntu 18.04.05 LTS. The benchmarks were run with a timeout of 10 minutes CPU time.

[5] https://fured.cloud.bme.hu/

TUE	Nidhugg	RCMC
0.895 s	0.170 s	0.066 s

Table 6.2: Comparison of the baseline execution times

The findings of the performance evaluation are as follows (I elaborate on the details and provide visualizations to the data in later sections):

- *Neither Nidhugg nor RCMC can handle infinite loops. The presented tool* can.
- *Due to the language disparities, the absolute minimum execution times of the presented tool (written in Java) is higher than the natively compiled Nidhugg and RCMC.*
- *The presented tool outperforms Nidhugg in the majority of the scenarios.*
- *The presented tool's performance is similar, but slightly worse than the algorithm with the hardcoded MCM in RCMC.*
- *The in-situ filtered verification mode of the presented tool performs significantly better in some scenarios, and marginally worse in other scenarios than the simple verification mode.*
- *The tool scales well by the number of available processor cores.*

6.2.1 Performance Evaluation Details

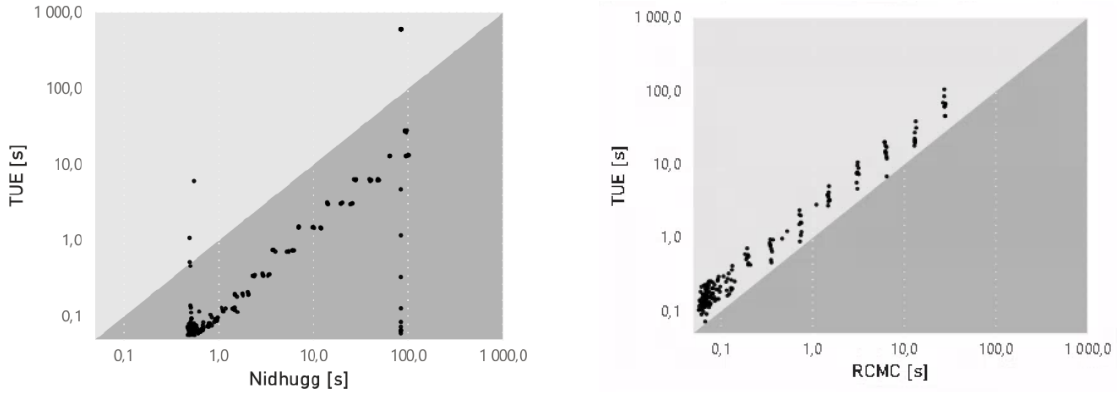
To support the claims above, I present the methodology of the benchmarking and its results. I used parameterizable benchmark templates to get scalability data from the tools under testing.

These benchmark templates are the following:

1. *Readers*: One thread writes a value with a given memory ordering, and N other threads read it back with another given memory ordering. Customizable are the number of threads $N + 1$, and the memory ordering types; in the benchmark run I used $N = 1..20$, *read accesses* = {relaxed, acquire, seq_cst}, *write accesses* = {relaxed, release, seq_cst} to yield 180 tests. (When necessary, the safety criterion can be: all threads read the same value written by the first thread)
2. *Loop*: One thread writes data in a loop, and N other threads read it back in a loop. In the benchmark run I used $N = 1..10$ to yield 10 tests. (When necessary, the safety criterion can be: a reader thread skips at least one value (e.g. jumps from 2 to 4))
3. *Coherence*: One thread writes data K times, and another thread reads it back K times. In the benchmark run I used $K = 1..20$ to yield 20 tests. (When necessary, the safety criterion can be: the reader reads the highest value possible $(K + 1)$)
4. *Empty*: To get a baseline of the time needed for starting, parsing and finalizing the executions, I used a completely empty input file to measure that. The outcomes can be seen in Table 6.2.

	TUE			TUE-insitu			Nidhugg			RCMC		
	Readers	**Loop**	**Coherence**	**Readers**	**Loop**	**Coherence**	**Readers**	**Loop**	**Coherence**	**Readers**	**Loop**	**Coherence**
OK	170	10	17	176	9	19	69	0	6	178	0	19
TIMEOUT	10	0	3	4	1	1	0	0	0	2	10	1
ERROR	0	0	0	0	0	0	111	10	14	0	0	0

Table 6.3: Outcomes of the tests. TUE and Nidhugg contain only the *weak (arm)* results



(a) Nidhugg vs. the presented tool

(b) RCMC vs. the presented tool

Figure 6.7: The *Readers* benchmark (without assertions, on a logarithmic scale)

As it can be deduced from the number of distinct tests above, the main benchmark I used was the *Readers* benchmark. This is taken from the paper by *Kokologiannakis et al.* [24], and they used it to show how poorly Nidhugg scaled in comparison with their tool, RCMC. The other benchmarks were mostly there for *feature* evaluation, as opposed to *performance* evaluation. The outcomes of these benchmark runs (complete with assertions) can be seen in Table 6.3.

The only contender capable of solving the *Loop* benchmark was the presented tool, the other two either timed out or ran into a problem while executing due to memory allocation problems. This is possibly due to its *BMC*-like behavior, as it can handle infinite loops by ignoring the rest of the execution after a certain number of iterations. Even though RCMC claims to handle infinite loops [24], this feature seems to be constrained to busy-wait loops, rather than a general solution. Of course, without an upper bound on the number of iterations, the state space is infinite, but there *was* an assertion in each test that was missed this way – a smarter exploration technique might have had helped even without BMC.

The other two families of benchmarks paint a similar picture of the three tools: Nidhugg fails to complete the tasks in the majority of the outcomes, while RCMC and the presented tool mostly solve them in time (and correctly). It is worth noting that RCMC and the presented tool behave in a very similar way across the bechmark runs – which is unsurprising, given that the basis of the proposed algorithm is implemented in RCMC. When we compare the two tools execution times (as seen in Figure 6.7b) in a benchmark run *without* assertions (but with in-situ filtering), the correlation is remarkable: even though the presented tool is always outperformed by RCMC, this only occurs by a small margin

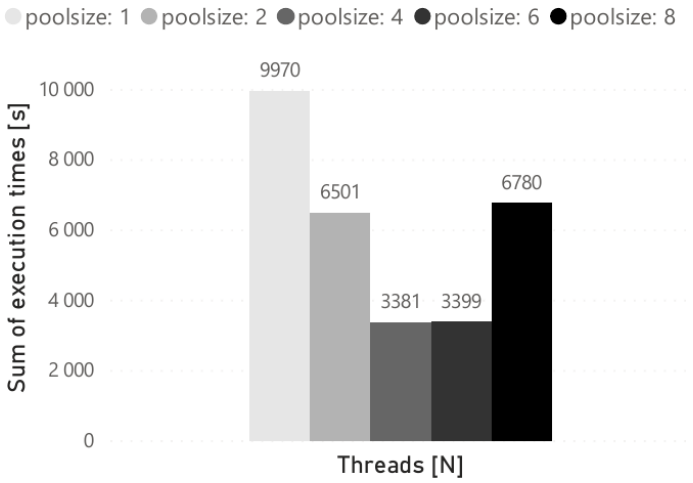
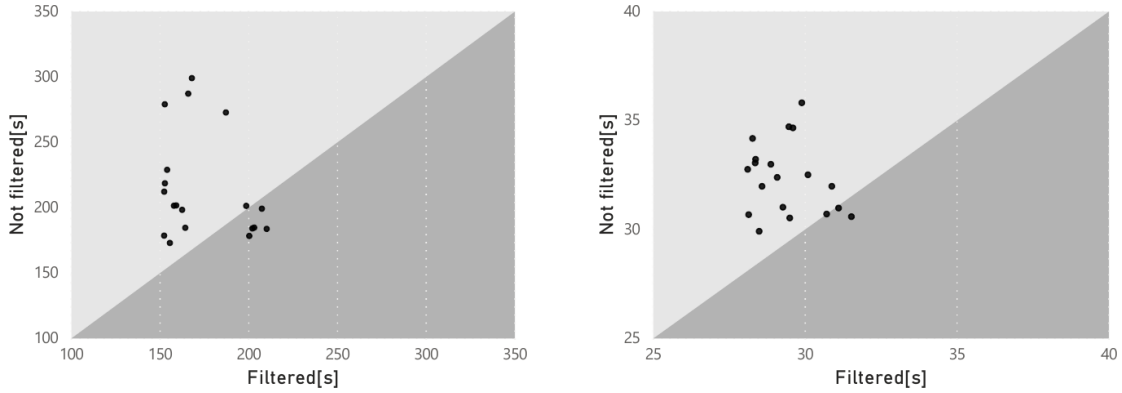


Figure 6.8: The multi-threaded performance of the presented tool



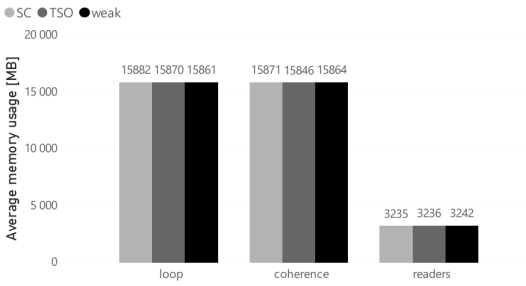
(a) Without assertions **(b)** With assertions

Figure 6.9: The *Coherence* benchmark (presented tool only)

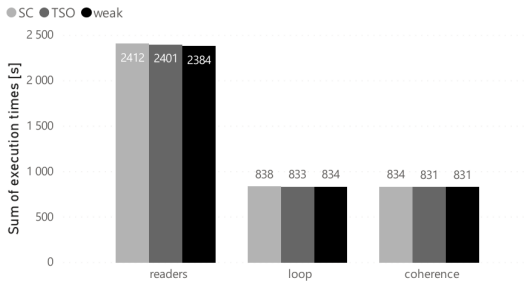
– which is acceptable, given the lack of customizability in RCMC and the general performance of Java programs compared to native ones. In comparison, when Nidhugg is compared to the tool in Figure 6.7a, the two tools' execution times do not show such a strong relationship. It is obvious that the tool outperforms Nidhugg in the majority of the scenarios, but there are some cases where it produces significantly worse execution times. (Note that only those tests were included where both tools produced a result in the given timeframe, and the *empty* execution times were subtracted from the tests).

The *Coherence* benchmark provides a great tool for vetting whether filtering is worth it, as the other benchmarks finish too early when assertions are enabled, and therefore their performance is not measurably different. In the case of the *Coherence* benchmark however, there is a significant difference between the tests' execution times, as seen in Figure 6.9 in both cases (with and without assertions). According to the charts, there are some cases when the filtering-induced overhead is slightly greater than its advantage, but in the majority of the tests the in-situ filtering approach won clearly – therefore it is worth using in the algorithm.

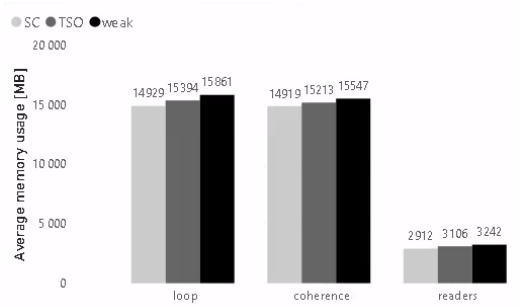
As for the customizability of the presented tool, the number of MCM-rules and their inner structure did not matter greatly in terms of execution time or utilized memory, *when the tool was running without in-situ filtering* – as seen on the charts in Figure 6.10, both metrics stayed almost constant among the different memory consistency models in the



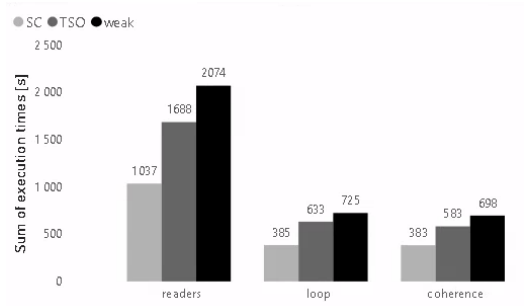
(a) Average memory usage in the benchmark families (without filtering)



(b) Sum of execution times in the benchmark families (without filtering)



(c) Average memory usage in the benchmark families (with filtering)



(d) Sum of execution times in the benchmark families (with filtering)

Figure 6.10: The effect of the *MCM* on the executions (presented tool only)

three families of the benchmark set. However, when *in-situ filtering* was enabled on the same benchmark set, the results changed quite a bit: the *SC*-model caused execution times to be almost half the *weak* ones, with *TSO* being in-between. This can be justified by the number of executions the runs had to enumerate – sequential consistency allows a lot fewer executions than weak memory does. Memory usage, however, stayed almost the same, which is caused by the *stateless* aspect of the tool: if the states are not stored globally, their number does not affect the memory usage greatly.

6.3 Future Work

Even though the presented approach works and scales well (as discussed earlier in the chapter), there is still work to be done to perfect the algorithm and the implementation. Some of the current limitations and future plans are the following:

- Stable translation tool for source code → XCFA transformation
- Simplify memory models for better performance
- Handling RMW (Read-modify-write) instructions (and other atomic expressions)
- Handling dynamic thread creation
- Changing the hard-coded generation of the *mo*-order to another method with fewer subexecutions
- Implementing a program slicing method, where the global state is handled by the presented algorithm, but locally a simpler model checking algorithm is used

The most interesting directions of development are the last two: the former is about the exponential growth of the state space when a new write is added due to all the new ***mo***-orders. This is taken entirely from RCMC's algorithm [24] – if we could get rid of this and still manage to stay sound (which is possible, as e.g. *herd* does not use such a relationship [6]), the result would be an even better performing algorithm, which could potentially outperform RCMC as well.

The last point of future plans is the most significant: right now, the implementation only handles local instructions if they are deterministic. There is no support for an uninitialized variable, for example – even though that can happen in a real-world scenario. However, if this approach was combined with a single-threaded model checking algorithm (such as CEGAR, as that is readily implemented in Theta [31]), the resulting tool could handle any combination of hard concurrent and thread-local problems.

Bibliography

[1] Cortex-A9 MPCore: Read-after-Read Hazards, 2011. URL https://documentation-service.arm.com/static/5ed75d07ca06a95ce53f93b9.

[2] Parosh Aziz Abdulla, Stavros Aronis, Mohamed Faouzi Atig, Bengt Jonsson, Carl Leonardsson, and Konstantinos Sagonas. Stateless Model Checking for TSO and PSO. *Tools and Algorithms for the Construction and Analysis of Systems Lecture Notes in Computer Science*, page 353–367, 2015. DOI: 10.1007/978-3-662-46681-0_28.

[3] Parosh Aziz Abdulla, Mohamed Faouzi Atig, Bengt Jonsson, and Carl Leonardsson. Stateless Model Checking for POWER. *Computer Aided Verification Lecture Notes in Computer Science*, page 134–156, 2016. DOI: 10.1007/978-3-319-41540-6_8.

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

[5] A. Adir, H. Attiya, and G. Shurek. Information-flow models for shared memory with an application to the powerPC architecture. *IEEE Transactions on Parallel and Distributed Systems*, 14(5):502–515, 2003. DOI: 10.1109/tpds.2003.1199067.

[6] 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.

[7] 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.

[8] A. Arvind and J.-W. Maessen. Memory Model = Instruction Reordering + Store Atomicity. *33rd International Symposium on Computer Architecture (ISCA'06)*, 2006. DOI: 10.1109/isca.2006.26.

[9] Levente Bajczi. HW-SW Co-verification of Concurrent Programs, 2018. URL https://tdk.bme.hu/VIK/ViewPaper/Konkurens-programok-HWSW-coverifikacioja.

[10] Levente Bajczi, András Vörös, and Vince Molnár. Will My Program Break on This Faulty Processor? *ACM Transactions on Embedded Computing Systems*, 18(5s):1–21, 2019. DOI: 10.1145/3358238.

[11] Dirk Beyer, Stefan Löwe, and Philipp Wendler. Reliable benchmarking: requirements and solutions. *International Journal on Software Tools for Technology Transfer*, 21 (1):1–29, 2017. DOI: 10.1007/s10009-017-0469-y.

[12] 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.

[13] Manfred Broy. A theory for nondeterminism, parallelism, communication, and concurrency. *Theor. Comput. Sci.*, 45(1):1–61, 1986. DOI: 10.1016/0304-3975(86)90040-X. URL https://doi.org/10.1016/0304-3975(86)90040-X.

[14] Gerald J. Burnett, L. J. Koczela, and Robert A. Hokom. A distributed processing system for general purpose computing. In *American Federation of Information Processing Societies: Proceedings of the AFIPS '67 Fall Joint Computer Conference, November 14-16, 1967, Anaheim, California, USA*, volume 31 of *AFIPS Conference Proceedings*, pages 757–768. AFIPS / ACM / Thomson Book Company, Washington D.C., 1967. DOI: 10.1145/1465611.1465710. URL https://doi.org/10.1145/1465611.1465710.

[15] E. M. Clarke, E. A. Emerson, and A. P. Sistla. Automatic verification of finite state concurrent system using temporal logic specifications. *Proceedings of the 10th ACM SIGACT-SIGPLAN symposium on Principles of programming languages - POPL '83*, 1983. DOI: 10.1145/567067.567080.

[16] Edmund Clarke, Daniel Kroening, Joël Ouaknine, and Ofer Strichman. Completeness and Complexity of Bounded Model Checking. *Lecture Notes in Computer Science Verification, Model Checking, and Abstract Interpretation*, page 85–96, 2004. DOI: 10.1007/978-3-540-24622-0_9.

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

[18] Michael S. Deutsch. *Software verification and validation*. Prentice-Hall, 1982.

[19] Cormac Flanagan and Patrice Godefroid. Dynamic partial-order reduction for model checking software. *Proceedings of the 32nd ACM SIGPLAN-SIGACT sysposium on Principles of programming languages - POPL '05*, 2005. DOI: 10.1145/1040305.1040315.

[20] Kourosh Gharachorloo. *Memory Consistency Models for Shared-Memory Multiprocessors*. Western Research Laboratory, 1995.

[21] Patrice Godefroid. Model checking for programming languages using VeriSoft. *Proceedings of the 24th ACM SIGPLAN-SIGACT symposium on Principles of programming languages - POPL '97*, 1997. DOI: 10.1145/263699.263717.

[22] ISO/IEC. ISO/IEC 9899:201x (N1548), Dec 2010. URL http://www.open-std.org/jtc1/sc22/wg14/www/docs/n1548.pdf.

[23] Ireneusz Karkowski and Henk Corporaal. Design of heterogenous multi-processor embedded systems: Applying functional pipelining. In *Proceedings of the 1997 Conference on Parallel Architectures and Compilation Techniques (PACT '97), San Francisco, CA, USA, October 11-15, 1997*, pages 156–165. IEEE Computer Society, 1997. DOI: 10.1109/PACT.1997.644012. URL https://doi.org/10.1109/PACT.1997.644012.

[24] Michalis Kokologiannakis, Ori Lahav, Konstantinos Sagonas, and Viktor Vafeiadis. Effective stateless model checking for C/C++ concurrency. *Proceedings of the ACM on Programming Languages*, 2(POPL):1–32, 2018. DOI: 10.1145/3158105.

[25] Robbert Krebbers, Xavier Leroy, and Freek Wiedijk. Formal C Semantics: CompCert and the C Standard. In Gerwin Klein and Ruben Gamboa, editors, *Interactive Theorem Proving - 5th International Conference, ITP 2014, Held as Part of the Vienna Summer of Logic, VSL 2014, Vienna, Austria, July 14-17, 2014. Proceedings*, volume 8558 of *Lecture Notes in Computer Science*, pages 543–548. Springer, 2014. DOI: 10.1007/978-3-319-08970-6_36. URL https://doi.org/10.1007/978-3-319-08970-6_36.

[26] Ori Lahav, Viktor Vafeiadis, Jeehoon Kang, Chung-Kil Hur, and Derek Dreyer. Repairing sequential consistency in C/C++11. 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 618–632. ACM, 2017. DOI: 10.1145/3062341.3062352. URL https://doi.org/10.1145/3062341.3062352.

[27] Daniel Lustig, Michael Pellauer, and Margaret Martonosi. PipeCheck: Specifying and Verifying Microarchitectural Enforcement of Memory Consistency Models. *2014 47th Annual IEEE/ACM International Symposium on Microarchitecture*, 2014. DOI: 10.1109/micro.2014.38.

[28] Brian Norris and Brian Demsky. CDSchecker. *Proceedings of the 2013 ACM SIGPLAN international conference on Object oriented programming systems languages; applications - OOPSLA '13*, 2013. DOI: 10.1145/2509136.2509514.

[29] Gyula Sallai. LLVM IR-alapú transzformációk szoftver modellellenőrzéshez. Master's thesis, 2019. URL https://diplomaterv.vik.bme.hu/hu/Theses/LLVM-IRalapu-transzformaciok-szoftver.

[30] 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.

[31] 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.

[32] Caroline Trippel, Yatin A. Manerkar, Daniel Lustig, Michael Pellauer, and Margaret Martonosi. TriCheck: Memory Model Verification at the Trisection of Software, Hardware, and ISA. *ACM SIGOPS Operating Systems Review*, 51(2):119–133, 2017. DOI: 10.1145/3093315.3037719.

[33] Antti Valmari. Stubborn sets for reduced state space generation. *Advances in Petri Nets 1990 Lecture Notes in Computer Science*, page 491–515, 1991. DOI: 10.1007/3-540-53863-1_36.

Appendix A

MCM Mapping Among Formalisms

The *cat* language has been shown to be capable of modeling complex memory architectures. By designing a simpler formalism instead, I only strived for syntactically easier representation of the same semantics, and therefore I show here how the basic building blocks of the language are mappable to the memory model introduced in Section 4.3 and to the DSL of the same model, introduced in Section 5.1.

cat	**MCM**	**DSL**
events	Writes, Reads, Fences	W, R, F
po, rf	po, rf	po, rf
tag	Current*	[tag]
definition	embedding (not necessary)	name = <expression>
Set algebra	Multiply, Subtract, Intersect, Union	*, \, intersect, union
loc	ForEachVar	for_each_var
ext	ForEachThread	for_each_thread
r?	Next	edgetype(A ->B)
*r**	Succ	edgetype(A –>B)
r+	Subtract(Succ, Next)	edgetype(A –>B) \edgetype(A ->B)
∼r	NA (no pre-defined universe)	NA (no pre-defined universe)
inverse (r^-1)	Next(B, A)	edgetype(B ->A)
sequence	ForEachNode	for_each_node